



Rational Rhapsody User Guide



Before using the information in this manual, be sure to read the “Notices” section of the Help or the PDF available from **Help > List of Books**.

This edition applies to IBM[®] Rational[®] Rhapsody[®] 7.5 and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright IBM Corporation 2000, 2009.

US Government Users Restricted Rights - Use, duplication, or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Introduction to Rational Rhapsody	1
Rational Rhapsody features	2
UML design essentials	3
UML diagrams	4
UML views	5
Diagrams in Rational Rhapsody	6
Specify a model with Rational Rhapsody	7
Development methodology	7
Analysis	7
Design	8
Implementation	9
The testing phase	9
Rational Rhapsody tools	10
The Rational Rhapsody browser	10
The Favorites browser	11
Diagram tools	12
Graphic editors	13
Code generator	15
Animator	15
Utilities	15
Third-party interfaces	16
Rational Rhapsody windows	17
View menu commands	19
Browser	22
Diagram drawing area	23
Diagram navigator	24
Output window	24
Active Code View window	33
Welcome window	33
Rational Rhapsody project tools	33
Browser filter	34
Standard tools	35
Edit menu commands	36

Table of Contents

Tools for Generating and running code.	37
Tools for managing and arranging windows	39
Tools for the Favorites browser.	40
Tools for the VBA interface options.	40
Tools for animation	40
Tools for creating and editing diagram elements	40
Tools for common annotations	41
Tools for zooming diagram views	41
Tools for formatting text	42
Tools for the layout of elements	42
Tools for free shapes	42
Creating diagrams	43
Tools for creating/opening diagrams.	44
Opening the main diagram	45
Locating in the browser.	45
Add new elements	46
Add New > Event	46
Add New > Interface	46
Add New > Actor	46
Add New > Tag	46
Add New > Use Case	46
Add New > Requirement.	46
Add New > Flow Item	47
The Features window	48
Open the Features window	48
Applying changes with the Features window	48
Canceling changes on the Features window.	49
General tab	49
Properties tab	49
Pinning the Features window	50
Hiding the buttons on the Features window	50
Docking the Features window	50
Undocking the Features window	51
Opening multiple instances of the Features window.	51
Displaying a tab on the Features window in a stand-alone window	51
Docking a stand-alone window for a Features window tab	52
Undocking a stand-alone window for a Features window tab	52
Hiding tabs on the Features window	53
Hyperlinks	55
Create hyperlinks	56
Following a hyperlink.	58
Edit a hyperlink	58
Deleting a hyperlink.	59

Hyperlink limitations	59
Create a diagram	60
Creating a diagram	60
Create a Rational Rhapsody project	60
Creating a Rational Rhapsody project	60
Import a Rational Rhapsody project	61
Importing a Rational Rhapsody project	61
Import source code	62
Importing source code	62
Search window	62
Graphic editors	62
Call stack and event queue	63
Classes and types	65
Creating a class	65
Class features	65
Defining the characteristics of a class	66
Defining the attributes of a class	67
Class operations	71
Primitive operations	72
Receptions	73
Triggered operations	75
Constructors	76
Destructors	79
Define class ports	80
Define relations	80
Showing all relations for a class, object, or package in a diagram	82
Defining class tags	83
Defining class properties	83
Adding a class derivation	84
Making a class an instance	85
Defining class behavior	85
Generating, editing, and roundtripping class code	85
Generating class code	85
Editing class code	86
Roundtripping class code	86
Opening the main diagram for a class	87
Display option settings	87

Table of Contents

General tab display options	88
Displaying attributes and operations	89
Removing or deleting a class	90
Ports	90
Partial specification of ports	91
Considerations	91
Creating a port	92
Specifying the features of a port	93
The Port General tab	93
The Port Contract tab	94
The Tags tab	96
The Properties tab	96
Viewing ports in the browser	97
Connecting ports	97
Using rapid ports	97
Selecting which ports to display in the diagram	100
Creating a new port for a class	100
Showing all ports	100
Showing new ports only	100
Hiding all ports	100
Deleting a port	101
Programming with the port APIs in C++	101
Port code generation in C	104
Port code generation in Java	105
Composite types	106
Creating enumerated types	107
Creating language types	108
Using %s	108
Creating structures	109
Creating Typedefs	110
Creating unions	110
Properties	111
Language-independent types	113
Changing the type mapping	114
Changing the order of types in the generated code	115
Using fixed-point variables	117
Defining fixed-point variables	117
Operations permitted for fixed-point variables	118
Restrictions on use of fixed-point variables	118
Fixed-point conversion macros	119

Java enums	120
Adding a Java enum to a model	120
Defining constants for a Java enum	120
Adding Java enums to an object model diagram	121
Code generation	121
Creating Java enums with the Rational Rhapsody API	121
Template classes and generic classes	122
Creating a template class	122
Using template classes as generalizations	124
Creating an operations template	124
Creating a functions template	125
Instantiating a template class	125
Code generation and templates	126
Template limitations	126
Eclipse platform integration	127
Platform integration prerequisites	128
Confirming your Rational Rhapsody Platform Integration within Eclipse	128
Rational Rhapsody Platform Integration within Eclipse	129
Rational Rhapsody perspectives in Eclipse	129
Rational Rhapsody Eclipse support for add-on tools	132
Eclipse projects	133
Creating a new Rational Rhapsody project within Eclipse	133
Opening a Rational Rhapsody project in Eclipse	133
Adding new elements	134
Filtering out file types	134
Exporting Eclipse source code to a Rational Rhapsody project	135
Importing Rational Rhapsody units	136
Importing source code (reverse engineering)	136
Search and replace in models	137
Accessing the Rational Rhapsody search facility in Eclipse	138
Generate and edit code	140
Checking the model	140
Generate code	141
Selecting Dynamic Model-Code Associativity	143
Edit code	143
Build, debug, and animate	145
Building your Eclipse project	145
Debugging your Eclipse project	146
Rational Rhapsody animation in Eclipse	146
Eclipse configuration management	149

Table of Contents

Parallel development	149
Configuration management and Rational Rhapsody unit view	150
Sharing a Rational Rhapsody model	151
Performing team operations	152
Rational Rhapsody DiffMerge facility in Eclipse	153
Generate Rational Rhapsody reports	153
Generating a report	153
Properties	155
Rational Rhapsody properties overview	155
Property groups and definitions	156
Subjects	157
Metaclasses	159
Regular expressions	160
Regular expression syntax	160
Parsing regular expressions	161
Property file format	162
Rational Rhapsody keywords	163
Predefined variables	163
Map custom properties to keywords	176
Rational Rhapsody properties	177
Using the Properties tab in the Features window	177
PRP files	187
Property inheritance	192
Concepts used in properties	193
Static architectures	193
IncludeFiles	193
Selective framework includes	194
Reactive classes	194
Units of collaboration	194
The Executer	195
Rational Rhapsody environment variables	196
Format properties	201
Defining default characteristics	202
Defining line characteristics	203
Rational Rhapsody projects	205
Project elements	205
Creating and managing projects	206

Creating a project	206
Profiles	207
Opening an existing Rational Rhapsody project	209
Search and replace facility	209
Locating and listing specific items in a model	213
File menu commands	214
Editing and changing a project	215
Using IDF for a Rational Rhapsody in C project	217
Saving a project	217
Renaming a project	219
Refactoring or renaming in the user code	219
Closing all diagrams	220
Closing a project	220
Closing Rational Rhapsody	220
Creating and loading backup projects	220
Archiving a project	221
Table and matrix views of data	222
Basic method to create views from layouts	222
Creating a table layout	223
Creating a table view	226
Creating a matrix layout	228
Creating a matrix view	230
Setting up an initial layout for table and matrix views	234
Managing table or matrix data	236
The Rational Rhapsody specialized editions	237
Creating projects in Rational Rhapsody Designer for Systems Engineers	237
Creating projects in Rational Rhapsody Architect for Systems Engineers	239
Creating projects in Rational Rhapsody Architect for Software	240
Components with variants for software product lines	241
Creating variation points	241
Defining variants	242
Selecting a variant	243
Generating code for software variations	243
Multiple projects	244
Inserting an existing project	244
Inserting a new project	245
Setting the active project	245
Copy and reference elements among projects	246
Moving elements among projects	248
Closing all open projects	248
Managing project lists	248
Project limitations	249

Table of Contents

Naming conventions and guidelines	251
Guidelines for naming model elements	251
Standard prefixes	252
Using project units	253
Unit characteristics and guidelines	254
Separating a project into units	255
Modifying units	256
Saving individual units	256
Loading and unloading units	256
Saving packages in separate directories	258
Using environment variables with reference units	260
Preventing unresolved references	261
Using workspaces	262
Creating a custom Rational Rhapsody workspace	262
Adding units to a workspace	262
Unloaded units	262
Opening a project with workspace information	263
Controlling workspace window preferences	263
Project files and directories	264
Parallel project development	266
Unit types	266
DiffMerge tool functions	267
Project migration and multi-language projects	268
Opening models from a different language version	268
Multi-language projects	269
Domain-specific projects and the NetCentric profile	272
Creating a NetCentric project	275
Creating a service contract to export as WSDL	275
Exporting a WSDL specification file	276
Importing a WSDL specification	276
Schedulability, Performance, and Time (SPT) profile	277
Manually adding the SPT profile to your model	277
Using the stereotypes and tagged values	278
Changing the profile	278
Rational Rhapsody with IDEs	279
IDE options	279
Locating Rational Rhapsody elements in an IDE	279
Opening the IDE	279
Creating an IDE project	279
Using the Rational Rhapsody Workflow Integration with Eclipse	280
Converting a Rational Rhapsody configuration to Eclipse	281

Importing Eclipse projects into Rational Rhapsody	281
Creating a new Eclipse configuration	282
Troubleshooting your Eclipse installation with Rational Rhapsody	283
Switching between Eclipse and Wind River Workbench	284
Rational Rhapsody tags for the Eclipse configuration	284
Configuring Rational Rhapsody for Eclipse	284
Eclipse workbench properties	285
Editing Rational Rhapsody code using Eclipse	286
Locating implementation code in Eclipse	286
Opening an existing Eclipse configuration	286
Disassociating an Eclipse project from Rational Rhapsody	287
Workflow integration with Eclipse limitations	287
Visual Studio IDE with Rational Rhapsody	288
Changing an existing Rational Rhapsody configuration to Visual Studio	288
Adding a new Visual Studio configuration	288
Creating a new Visual Studio project	288
Co-debugging with Tornado	289
Preparing the Tornado IDE	289
IDE operation in Rational Rhapsody	290
Co-debugging with the Tornado debugger	290
IDE properties	291
Creating Rational Rhapsody SDL blocks	292
Model elements	295
Browser techniques for project management	295
Opening the Rational Rhapsody browser	296
Browser display options	296
Basic browser icons	299
Rational Rhapsody browser menu options	302
Deleting items from the Rational Rhapsody browser	303
The Browse From Here browser	304
Opening a Browse From Here browser	304
Closing a Browse From Here browser	304
Navigating a Browse From Here browser	305
Deleting items from the Browse From Here browser	305
Browse From Here browser limitations	305
The Favorites browser	306
Favorites toolbar	307
Showing and hiding the Favorites browser	307
Creating your Favorites list	308
Creating a folder structure for your Favorites	309
Re-ordering the items on your Favorites list	310

Table of Contents

Removing items from your Favorites list	311
Favorites browser limitations	312
Elements	313
Adding elements	313
Naming new elements in the browser	314
Browser settings	314
Components	315
Configurations	315
Configuration files	315
Packages	316
Package design guidelines	316
Creating a package	317
Using functions	318
Using objects	318
Using variables	319
Dependencies	320
Constraints	320
Classes	320
Types	320
Receptions	320
Events	320
Actors	321
Use cases	321
Nodes	321
Files	321
Diagrams	322
Adding diagrams	322
Locating an element on a diagrams	323
Element identification and paths	326
Descriptive labels for elements	327
Setting properties for Asian languages	327
Adding a label to an element	329
Removing a label from an element	330
Label mode	330
Modify elements	331
Moving elements	331
Copying elements	332
Renaming elements	332
Deleting elements	333
Editing multiple elements	333
Re-ordering elements in the browser	334

Displaying stereotypes of model elements	335
Creating graphical elements	336
Smart drag-and-drop	339
Searching in the model	342
Finding element references	342
Advanced search and replace features	343
Using the auto replace feature	344
Searching for elements	345
Searching in field types	346
Previewing in the search and replace facility	348
Controlled files	349
Creating a controlled file	350
Browsing to a controlled file	351
Controlled file features	352
Troubleshooting controlled files	356
Controlled file limitations	357
Print Rational Rhapsody diagrams	360
Selecting which diagrams to print	360
Diagram print settings	362
Using page breaks	363
Exporting Rational Rhapsody diagrams	364
Annotations for diagrams	365
Creating annotations	366
Editing annotation text	368
Defining the features of an annotation	368
Converting notes to comments	370
Anchoring annotations	370
Changing the display options for annotations	372
Deleting an annotation	373
Using annotations with other tools	374
Annotation limitations	374
Profiles	375
Creating a project without a profile	376
Backward compatibility profiles	377
Types of profiles	378
Converting packages and profiles	378
Profile properties	378
Use a profile to enable access to your custom help file	379
Stereotypes	385
Associating stereotypes with an element	385
Associate a stereotype with a new term element	386

Table of Contents

Re-ordering stereotypes in a list	387
Associating a stereotype with a bitmap	387
Deleting stereotypes	388
Establishing stereotype inheritance	389
Special stereotypes	389
Use tags to add element information	390
Defining a stereotype tag	390
Defining a global tag	391
Defining a tag for an individual element	391
Adding a value to a tag	391
Deleting a tag	393
The Internal code editor	395
Window properties	395
The Color/Font tab	395
The Language/Tabs tab	398
The Keyboard tab	399
The Misc tab	403
Mouse actions	405
Using Undo and Redo	405
Using the search feature of the internal code editor	406
Bookmarks	407
Printing from the internal code editor	408
Graphic editors	409
Create new diagrams	409
Creating new statecharts	409
Creating new activity diagrams	409
Creating all other diagram types	410
Opening existing diagrams	411
Navigating forward from opened diagram to opened diagram	411
Navigating backwards from opened diagram to opened diagram	411
Deleting diagrams	412
Automatically populating a diagram	412
Relation type styles	412
Creating and populating a new diagram	413
Automatically populating existing diagrams	415
Property settings for the diagram editor	416
Setting diagram fill color	417

Create elements	417
Repetitive drawing mode	418
Drawing boxes	418
Drawing arrows	419
Naming boxes and arrows	421
Draw freestyle shapes	422
Placing elements using the grid	428
Setting the grid properties	428
Snapping to the grid	428
Displaying the rulers	428
Autoscroll	429
Select elements	430
Selecting elements using the mouse	430
Selecting elements using the edit menu	430
Selection handles	431
Selecting multiple elements	431
Edit elements	433
Resizing elements	434
Moving control points	434
Moving elements	435
Maintain line shape when moving or stretching elements	435
Change the format of a single element	436
Copying formatting from one element to another	440
Changing the format of a metaclass	441
Making the format for an element the default formatting	445
Copy an element	445
Arranging elements	448
Removing an element from the view	451
Deleting an element from the model	451
Editing text	452
Display compartments	453
Selecting items to display	453
Display stereotype of items in list	454
Zoom	455
Zoom toolbar	455
Zooming in and zooming out	456
Refreshing the display	456
Scaling a diagram	457
Panning a diagram	457
Undoing a zoom	457
Specifying the specification or structured view	457

Table of Contents

The Bird's Eye (diagram navigator)	459
Showing and hiding the Bird's Eye window	459
Navigating to a specific area of a diagram	459
Using the Bird's Eye to enlarge and shrink the visible area	459
Scrolling and zooming in drawing area	460
Changing the appearance of the viewport	460
General characteristics of the Bird's Eye window	460
Complete relations	461
Use IntelliVisor	462
Activating IntelliVisor	462
IntelliVisor information	463
Customizations for Rational Rhapsody	469
Helpers	470
Creating a link to a helper application	472
Adding a VBA macro	479
Visual Basic for applications	481
VBA and Rational Rhapsody	481
The VBA project file	481
VBA versus VB programs	481
Writing VBA macros	482
Creating and editing macros	483
Exporting and importing VBA macros	485
Creating a customized profile	486
Creating a new stereotype for the new profile	487
Re-using your customized profile	487
Adding new element types	489
New terms and their properties	489
Availability of out-of-the-box model elements	490
Creating a customized diagram	493
Adding customized diagrams to the diagrams toolbar	494
Creating a customized diagram element	494
Adding customized diagram elements	496
Diagram types	496
Diagram elements	497
Customize the Add New menu	501
Re-organizing the common list section of the Add New menu	501
Re-organizing the bottom section of the Add New menu	502
Customizing the Add New menu completely	504
Re-using property changes to the Add New menu	505

Creating a Rational Rhapsody plug-in	507
Writing a Java plug-in for Rational Rhapsody	507
Creating a .hep file for the plug-in	510
Attaching a .hep file to a profile	512
Troubleshooting Rational Rhapsody plug-ins	512
Debugging Rational Rhapsody plug-ins	513
The simple plug-in sample	515
 Use case diagrams	 517
Use case diagrams overview	517
Opening an existing use case diagram	518
Create use case diagram elements	519
Use case diagram drawing tools	519
System boundary box	520
Use cases	520
Actors	524
Creating packages	526
Creating associations	527
Creating generalizations	527
Creating dependencies	528
Sequences	528
 Object model diagrams	 529
Object model diagrams overview	529
Object model diagram elements	530
Object model diagram drawing tools	530
Objects	531
Opening an existing object model diagram	532
Creating an object	532
Object characteristics	532
Parts in an object model diagram	533
Object features	534
Converting object types	535
Converting classes to objects	535
Code generation for objects	536
Editing the declaration order of objects	537
Changing the value of an instance	537
Creating a vacuum pump model as an example	540
Creating classes	545
Class compartments	545

Table of Contents

Creating composite classes	546
Creating a package	547
Package features	547
Inheritance	548
Realization	550
Associations	551
Bi-directional associations	551
Creating a bi-directional association	552
Association features	553
Directed associations	558
Aggregation associations	559
Composition associations	561
Links	565
Dependencies	572
Dependency arrows	572
Drawing the dependency	573
Actors	577
Creating an actor	577
The actor menu	577
Flows and flowitems	578
Creating a flow	579
Features of a flow	581
Conveyed information	582
Flow menu	583
Flowitems	583
Embedded flows	586
Files	587
Creating a file	588
Converting files	592
Associations and dependencies	592
Code generation for files	593
Files with other tools	595
Attributes, operations, variables, functions, and types.	596
Adding details to the object model diagram	596
Flow ports	597
External elements	599
Reverse engineering	600
External elements created by modeling	605
Converting external elements	608
Implementation of the base classes	613

Implicit invocation	613
Explicit invocation	614
Namespace containment	618
Activity diagrams	621
Activity diagram features	621
Advanced features of activity diagrams	622
Actions	622
Activity diagram elements	623
Activity diagram drawing tools	623
Drawing an action	625
Modify the features of an action	625
Displaying an action	626
Activity frames	626
Action blocks	628
Subactivities	630
Creating a final activity	631
Object nodes	632
Adding call behaviors	634
Activity flows	634
Connectors	636
Join or fork bars	638
Swimlanes	642
Adding calls to behaviors	646
Add action pins/activity parameters to diagrams	648
Local termination semantics	650
Code generation	652
Functor classes	652
Limitations and specified behavior	654
Flow charts	655
Define algorithms with flow charts	655
Flow charts similarity to activity diagrams	656
Create flow chart elements	657
Tools for drawing flow charts	657
Actions	658
Action blocks	660
Activity final	662
Activity flows	663
Connectors	664

Table of Contents

Code generation	666
Flow chart limitations and specified behavior	666
Sequence diagrams	669
Sequence diagram layout	670
Names pane	671
Message pane	672
Analysis versus design mode	672
Showing unrealized messages	673
Realizing a selected element	673
Creating sequence diagram elements	674
Sequence diagram drawing tools	674
Creating a system border	676
Creating an instance line	677
Creating a message	681
Creating a reply message	690
Drawing an arrow	693
Creating a destroy arrow	693
Creating a condition mark	693
Creating a timeout	694
Creating a cancelled timeout	694
Creating an actor line	695
Specifying a time interval	695
Creating a dataflow	696
Creating a partition line	698
Creating an interaction occurrence	698
Creating interaction operators	701
Creating execution occurrences	703
Shifting diagram elements with the mouse	705
Display options	706
Sequence diagrams in the browser	706
Animation for selected classes	707
Sequence diagram comparison	707
Sequence comparison algorithm	707
Comparing sequence diagrams	708
Sequence comparison options	710
The Instance Groups tab	715
The Message Groups tab	720
Statecharts	725
States	726

Opening an existing statechart	727
Statechart drawing tools	727
Drawing a state	728
State name guidelines	728
Features of states	729
Display options for states	731
Termination states	732
Local termination code with the reusable statechart implementation	732
Local termination code with flat statechart implementation	733
Transitions	735
Creating a statechart transition	735
Features of transitions	736
Types of transitions	738
Selecting a trigger transition	740
Transition labels	741
Triggers	741
Guards	747
Actions	749
Initial connectors	750
Events and operations	751
Sending events across address spaces	752
Properties for sending events across address spaces	752
API for sending events across address spaces	753
Functions for serialization/unserialization	754
Send action elements	756
Defining send action elements	756
Display options for send actions	757
Graphical behavior of send actions	757
Code generation for send actions	757
And lines	758
Drawing And lines	758
Connectors	759
Decision nodes	760
History connectors	761
Merge nodes	762
Diagram connectors	762
Termination connectors	763
EnterExit points	763
Submachines	765

Table of Contents

Creating a submachine	765
Opening a submachine or parent statechart	765
Deep transitions	765
Merging a sub-statechart into its parent statechart	766
Statechart semantics	767
Single message run-to-completion processing	767
Active transitions	768
Transition selection	768
Transition execution	770
Active classes without statecharts	770
Single-action statecharts	770
Inherited statecharts	771
Types of inheritance	772
Inheritance color coding	772
Inheritance rules	773
Overriding inheritance rules	776
Overriding textual information	777
Refining the hierarchy of reactive classes	778
IS_IN Query	781
Message parameters	783
Modeling of continuous time behavior	785
Interrupt handlers	785
Inlining of statechart code	786
Tabular statecharts	787
Format of statechart tables	787
Modifying statecharts from tabular view	788
Panel diagrams	791
Panel diagram features	792
Creating a panel diagram	794
Create panel diagram elements	795
Panel diagram drawing tools	795
Drawing a bubble knob control	796
Drawing a gauge control	797
Drawing a meter control	798
Drawing a level indicator control	799
Drawing a matrix display control	800
Drawing a digital display control	801
Drawing an LED control	802
Drawing an on/off switch control	803

Drawing a push button control	804
Drawing a button array control	805
Drawing a text box control.	806
Drawing a slider control.	807
Bind a control element to a model element	808
Binding a control element	809
More about binding a control element.	809
Change the settings for a control element	811
Changing the settings for a control	811
Change the properties for a control element	812
Properties for a bubble knob control	812
Properties for a gauge control.	814
Properties for a meter control	817
Properties for a level indicator control.	820
Properties for a matrix display control.	822
Properties for a digital display control	822
Properties for a LED control	823
Properties for a on/off switch control.	824
Properties for a slider control	825
Setting the value bindings for a button array control	827
Changing the display name for a control element	827
Panel diagram limitations.	828
Structure diagrams	829
Structure diagram drawing Tools	830
Composite classes	830
Objects	831
Creating an object.	831
Features of objects	832
Actual Call window for objects.	833
Changing the order of objects	833
Supported Rational Rhapsody functionality in objects	834
Structure diagram ports	835
Links and associations.	835
Dependency uses	835
Flows mechanism	835
External files in C	836

Collaboration diagrams	837
Collaboration diagrams overview	837
Collaboration diagram tools	839
Classifier roles	840
Multiple objects	840
Actors	841
Creating an actor	841
Links	841
Creating a link	842
Features of links	843
Changing the underlying association	844
Link messages and reverse link messages	844
Creating a link message or reverse link message	845
Component diagrams	847
Component diagram uses	848
Component diagram drawing Tools	849
Elements of a component diagram	850
Components	850
Files	852
Folders	855
Dependencies	857
Component interfaces and realizations	857
Flows	858
Component configurations in the browser	859
Component options	859
Active component	860
Configurations	860
Configuration menu	861
Setting the active configuration	861
Features of configurations	862
Using selective instrumentation	866
Making permanent changes to the main file	868
Creating components under a package	869
Deployment diagrams	871
Opening an existing deployment diagram	872
Deployment diagram drawing tools	872

Nodes	873
Creating a node	873
Changing the owner of a node	874
Designating a CPU type	874
Features of nodes	874
Component instances	875
Adding a component instance	875
Moving a component instance	877
Features of component instances	877
Dependencies	878
Adding a dependency	878
Flows	879
Assigning a package to a deployment diagram	880
Checks	881
Checker features	881
The Checks tab	882
Specifying which checks to run	884
Checking the model	885
Checks tab limitations	885
User-defined checks	886
Creating user-defined checks	886
Removing user-defined checks	887
Deploying user-defined checks	888
External checks limitations	888
List of Rational Rhapsody checks	889
Basic code generation concepts	909
Code generation overview	909
The Code Toolbar	912
Generating Code	912
Incremental Code Generation	912
Smart Generation of Packages	913
Generating Code Guidelines	914
Dynamic Model-Code Associativity	914
Generating Makefiles	914
Stopping Code Generation	915
Targets	916

Table of Contents

Building the Target	916
Deleting Old Objects Before Building Applications	917
Running the Executable	918
Shortcut for Creating an Executable	918
Instrumentation	918
Stopping Model Execution	918
Generating Code for Individual Elements	919
Using the Code Menu	919
Using the Browser	919
Using an Object Model Diagram	919
Results of Code Generation	920
Output Messages	920
Locating and Fixing Compilation Errors	920
Viewing and Editing the Generated Code	921
Setting the Scope of the Code View Editor	921
Adding Line Numbers	922
Editing Code	923
Locating Model Elements	923
Regenerating Code in the Editor	924
Associating Files with an Editor	924
Using an External Editor	925
Viewing Generated Operations	925
Deleting Redundant Code Files	926
Generating Code for Actors	926
Selecting Actors Within a Component	927
Limitations on Actor Characteristics	927
Generating Code for Component Diagrams	928
Cross-Package Initialization	930
Class Code Structure	932
Class Header File	932
Implementation Files	938
Changing the Order of Operations/Functions in Generated Code	942
Using Code-Based Documentation Systems	944
Template Properties	944
Sample Usage	945
Wrapping Code with #ifdef-#endif	949
Overloading Operators	949
Using Anonymous Instances	954
Creating Anonymous Instances	954

Deleting Anonymous Instances.	955
Deleting Components of a Composite.	955
Using Relations	956
To-One Relations	956
To-Many Relations	956
Ordered To-Many Relations	957
Qualified To-Many Relations.	957
Random Access To-Many Relations.	958
Support for Static Architectures	959
Properties for Static Memory Allocation	960
Static Memory Allocation Algorithm.	962
Static Memory Allocation Conditions.	963
Static Memory Allocation Limitations.	963
Using Standard Operations	964
Applications for Standard Operations	964
Creating Standard Operations	966
Statechart Serialization.	970
Generating Methods for Serialization	970
Serialization Properties	970
Methods Provided for Implementing Serialization.	971
Generating Classes as Structs in C++.	972
Components-based Development in C	973
Action Language for Code Generation	974
C Optimization.	975
Backward Compatibility.	976
Limitations.	976
Customize C code generation	977
Code customization concepts	977
Customizing code generation	978
Viewing the simplified model	979
Customize the generation of the simplified model	979
Properties used for simplification	979
Customizing the code writer	980
Customizing the C rules	981
Deploying the changed rules.	984
Reverse engineering	985
Reverse engineering restrictions.	985

Table of Contents

Reverse engineering legacy code	986
Reverse engineering tool features	986
Displaying files in a tree view	987
Displaying files in a flat view	989
Reverse engineering messages in the Output window	990
Initializing the Reverse Engineering window	991
Excluding particular files	992
Analyzing makefiles	992
Visualization of external elements	994
Defining preprocessor symbols	995
Adding a preprocessing symbol	996
Analyzing #include files	1003
Mapping classes to types and packages	1008
Specifying directory structures	1014
Specifying reference classes	1016
Reference classes	1018
Locating a directory that contains reference classes	1019
Miscellaneous reverse engineering options	1020
Modeling classes as Rational Rhapsody types	1023
Reflect data members	1028
Reverse engineering error handling	1031
Creating flow charts during reverse engineering	1032
Updating existing packages	1033
Command-line interface for populate object model diagrams	1034
Populate object model diagrams limitations	1034
Reverse engineering message reporting	1035
Code respect and reverse engineering for Rational Rhapsody Developer for C and C++ ..	1037
Reverse engineering for C++	1037
Reverse engineering for Rational Rhapsody in Java	1037
Reverse engineering other constructs	1038
Unions	1038
Enumerated types	1038
Comments	1039
Limitations for comments	1040
Macro collection	1041
Collected macro file	1041
Code Generation	1042

Controlling macro collection	1042
Code generation of imported macros	1043
Limitations for imported macros	1043
Backward compatibility issues.	1044
Results of reverse engineering	1044
Lost constructs	1045
Roundtripping	1047
Supported elements	1048
Roundtripping limitations.	1048
Dynamic Model-code Associativity (DMCA)	1049
The roundtripping process.	1050
Automatic and forced roundtripping	1050
Roundtripping classes.	1050
Modifying code segments for roundtripping	1051
Recovering lost roundtrip annotations.	1052
Roundtripping classes.	1053
Roundtripping packages	1055
Roundtripping deletion of elements from the code	1057
Roundtripping for C++.	1058
Roundtripping for Java	1059
Roundtripping properties.	1059
Code respect	1063
Activating the code respect feature.	1064
Where code respect information is defined	1065
Making SourceArtifacts display in the browser	1066
Manually adding a SourceArtifact	1067
Reverse engineering and SourceArtifacts.	1067
Roundtripping and SourceArtifacts	1067
Code generation and SourceArtifacts	1068
Configuration management and SourceArtifacts.	1068
Code-centric mode	1069
Entering code-centric mode.	1069
Leaving code-centric mode	1070
Roundtripping in code-centric mode.	1072
Code generation in code-centric mode.	1073
Diagrams for which code not generated	1074

Table of Contents

Code regeneration in code-centric mode	1074
Animation in code-centric mode	1075
Scope for code-centric models	1076
Properties modified by code-centric settings	1077
Animation	1079
Animation Overview	1080
Animation Features	1080
Preparing for Animation - General Procedure	1080
Create a Component	1081
Creating a component	1081
Setting the Component Features	1082
Creating a Configuration	1083
Setting the Instrumentation Mode	1084
Running the Animated Model	1086
Running on the Host	1086
Running on a Remote Target	1087
Opening a Port Automatically	1088
Testing an Application on a Remote Target	1088
Testing a Library	1089
Partially Animating a Model (C/C++)	1089
Setting Elements for Partial Animation	1090
Partial Animation Considerations	1090
Partially Animated Sequence Diagrams	1091
Ending an Animation Session	1092
Animation Toolbar	1093
Creating Initial Instances	1094
Break Command	1095
Command Prompt	1095
Generating Events Using the Animation Command Bar	1095
Events with Arguments	1096
Generating Events Using the Command History List	1097
Threads	1098
Thread View	1098
Setting the Thread Focus	1098
Names of Threads	1099
Notes on Multiple Threads	1099
Active Thread Properties	1100

Creating Breakpoints	1101
Defining Breakpoints	1102
Enabling and Disabling Breakpoints	1104
Deleting Breakpoints	1105
Event Generator	1106
Generating Events	1106
Events History List	1107
Calling Animation Operations	1108
Scheduling and Threading Issues	1110
Using Partial Animation	1110
Scheduling and Threading Restrictions	1110
Animation Modes	1112
Silent Mode	1112
Watch Mode	1112
Viewing the Model	1113
Call Stack	1114
Event Queue	1114
Animated Browser	1115
Animated Sequence Diagrams	1115
Animating Statecharts	1124
Instance Names	1125
Names of Class Instances	1125
Names of Component Instances	1125
Navigation Expressions	1126
Names of Special Objects	1126
Animation Scripts	1126
Sample Script	1127
Running Scripts Automatically	1128
Black-Box Animation	1130
Animation Properties	1130
Example	1131
Using the Properties for Black-Box Testing	1134
Instance Line Menu	1135
Behavior and Restrictions	1135
Animation Hints	1136
Exception Handling	1136
If Animation and Application are Out of Sync	1136
Passing Complex Parameters	1137
Combining Animation Settings in the Same Model	1137
Animation Feature Limitations	1137
Guidelines for Writing Serialization Functions	1138

Table of Contents

AnimSerializeOperation	1138
AnimUnserializeOperation.	1140
Running an Animated Application Without Rational Rhapsody.	1141
Tracing	1143
Tracer Capabilities	1143
Starting a Trace Session.	1144
Controlling Tracer Operation	1145
Accessing Tracer Commands.	1145
Tracer Commands and an Input File.	1145
Threads in Tracing	1147
Tracer Commands	1148
break	1148
CALL	1151
display	1153
GEN	1153
go	1154
help	1155
input	1155
LogCmd.	1156
output	1157
quit	1157
resume	1158
set focus	1158
show	1159
suspend.	1162
timestamp	1162
trace	1162
watch.	1166
Tracer Messages by Subject	1167
Ending a Trace Session	1169
Managing Web-enabled devices	1171
Use of Web-enabled Devices	1171
Setting Model Elements as Web-Manageable.	1172
Limitations on Web-Enabling Elements.	1173
Selecting Elements to Expose to the Internet.	1174
Connecting to the Web Site from the Internet	1176
Navigating to the Model through a Web Browser	1176
The Web GUI Pages	1178

Viewing and Controlling of a Model via the Internet	1183
Customizing the Web Interface	1184
Adding Web Files to a Rational Rhapsody Model.	1184
Accessing Web Services Provided with Rational Rhapsody.	1185
Adding Rational Rhapsody Functionality to Your Web Design	1189
Customizing the Rational Rhapsody Web Server.	1193
Reports	1195
ReporterPLUS	1195
Launching ReporterPLUS.	1196
ReporterPLUS templates	1196
Generating reports using existing templates.	1200
Viewing reports online.	1201
Generating a list of specific items	1201
Using the system model template.	1201
The internal reporting facility.	1203
Producing an internal report	1203
Setting the RTF character set	1205
Using the internal report output.	1205
Java-specific issues	1207
Generation of Javadoc comments.	1207
Including Javadoc comments in Rational Rhapsody-generated code.	1207
Changing the appearance of Javadoc comments in generated code	1208
Enabling/disabling Javadoc comment generation.	1208
"Built-in" keywords.	1209
Description templates in JavaDocProfile.	1209
Multiple appearance of Javadoc tags	1209
Adding new Javadoc tags	1210
Javadoc handling in reverse engineering and roundtripping.	1211
Javadoc troubleshooting	1211
Static import.	1212
Adding static imports to a model.	1212
Reverse engineering/roundtripping and static import statements	1212
Code generation checks	1213
Static blocks	1213
Adding static blocks to classes in a model	1213
Changing a static block to an operation	1213
Reverse engineering/roundtripping and static blocks	1214
Generating JAR files	1214
Java 5 annotations	1215

Table of Contents

Creating a JavaAnnotation type	1215
Using a JavaAnnotation type	1216
Using a JavaAnnotation within a model	1217
Code generation and Java 5 annotations	1220
Reverse engineering and Java 5 annotations	1220
Limitations for Java 5 annotations	1221
Java reference model	1221
Systems engineering with Rational Rhapsody	1223
Installing and launching systems engineering	1223
Creating a SysML profile project	1224
SysML profile features	1225
SysML profile packages	1226
Views and viewpoints	1227
Adding elements	1228
Harmony process and toolkit	1230
Harmony process summary	1230
Creating a Harmony project	1232
Harmony profile features	1234
Systems engineering requirements in Rational Rhapsody	1242
Analysis and requirements using the Rational Rhapsody Gateway	1243
Searching requirements	1245
Creating Rational Rhapsody requirements diagrams	1245
Creating specialized requirement types	1249
Requirements tabular view	1250
Creating use case diagrams	1251
Boundary box and the environment	1252
Actors and systems design in use cases	1252
Use case features for systems engineering	1253
Associating actors with use cases	1254
Defining requirements in use case diagrams	1255
Tracing requirements in use case diagrams	1255
Dependencies between requirements and use cases	1255
Defining flow in a use case diagram	1256
Defining the stereotype of a dependency	1256
Activity modeling in SysML	1257
Action types in SysML	1257
SysML activity diagrams	1257
Creating an activity diagram	1258
Setting activity diagram properties	1258
Activity diagram drawing tools for systems engineering	1259

Drawing action states	1260
Drawing a initial flow	1261
Drawing a subactivity	1261
Drawing activity flows	1261
Drawing activity flows between states	1262
Drawing swimlanes	1262
Drawing a fork node	1263
Drawing a join node	1263
Creating a sequence diagram from an activity diagram	1263
Creating a design structure	1264
Block properties	1264
Blocks and behaviors	1265
Creating a block definition diagram	1265
Block definition diagram drawing tools	1266
Adding graphics to block definition diagrams	1268
Creating an internal block diagram	1269
Internal block diagram drawing tools	1270
Drawing the parts	1270
Drawing standard ports and links	1271
Specifying the port contract and attributes	1271
Parametric diagrams	1272
Parametric diagram drawing tools	1273
Creating the constraint block	1274
Creating the parametric diagram	1275
Binding constraint properties together	1276
Adding equations	1276
Implementation using the action language	1277
Basic syntax rules	1277
Frequently used statements	1278
Reserved words	1278
Assignment and arithmetic operations	1279
Defining an action using the action language	1279
Checking action language entries	1280
Action language reference	1281
System validation	1286
Creating a component	1286
Setting the component features	1287
Creating a configuration	1287
Preparing to Web-enable the model	1288
Creating a Web-enabled configuration	1288
Selecting elements to Web-enable	1290

Table of Contents

Connecting to the Web-enabled model	1291
Navigating to the model through a Web browser	1291
Viewing and controlling a model	1292
Sending events to your model	1292
Importing DoDAF diagrams from Rational System Architect	1293
Mapping the import scope	1293
Importing the Rational System Architect elements	1295
Converting imported data into a Rational Rhapsody diagram	1296
Post processing mechanism for Rational System Architect users	1297
Generating a Imported Elements report	1297
Integration with Teamcenter systems engineering	1298
UML or SysML	1298
Prerequisites for working with Rational Rhapsody	1300
Importing a Rational Rhapsody model into Teamcenter	1300
Creating a Rational Rhapsody model from existing Teamcenter Project	1301
Modifying shared elements from within Teamcenter	1301
Limitations	1302
The MicroC profile	1303
The extended execution model	1303
MicroC code generation	1303
UI changes	1303
The mxr	1304
Modeling network ports	1304
Optimizations for static systems	1305
Direct flow ports	1305
Direct relations	1306
Monitoring of application running on target	1307
Using target monitoring	1307
Viewing MicroC properties	1309
IBM Rational Rhapsody DoDAF Add On	1311
Rational Rhapsody for DoDAF Add On and profile	1311
DoDAF views	1312
Operational view	1312
Systems view	1313
Technical view	1313
All views	1313
Products included in the Rational Rhapsody for DoDAF Add On	1314

Rational Rhapsody for DoDAF Add On helper utilities	1318
Setup DoDAF packages	1320
Create OV-2 from Mission Objective	1320
Create OV-6c from Mission Objective	1320
Update OV-2 from OV-6c	1320
Generate Service Based OV-3 Matrix	1320
Generate SV-3 Matrix	1320
Generate SV-5 Summary Matrix	1320
Generate SV-5 Full Matrix	1321
Rational Rhapsody for DoDAF Add On Report Generator	1321
Rational Rhapsody project for Rational Rhapsody for DoDAF Add On configuration	1322
Creating a Rational Rhapsody for DoDAF project	1322
Diagrams toolbar for a Rational Rhapsody for DoDAF project	1325
DoDAF tags	1327
Generating the OV-3 Operational Information Exchange Matrix	1329
Generating the DoDAF report from the architecture model	1331
Limitations	1333
Troubleshooting	1334
Verifying the Rational Rhapsody for DoDAF Add On installation	1334
Manually adding the Rational Rhapsody for DoDAF Add On helpers	1335
Correcting messages that appear as mission objectives	1337
View, caption, or table of figures is missing from document	1339
IBM Rational Rhapsody MODAF Add On	1341
Rational Rhapsody for MODAF Add On	1342
MODAF viewpoints	1343
All Views viewpoint	1345
Strategic viewpoint	1345
Operational viewpoint	1345
Systems viewpoint	1346
Acquisition viewpoint	1346
Technical viewpoint	1346
Views Included in the Rational Rhapsody for MODAF Add On	1347
Configure a Rational Rhapsody project for MODAF	1355
Creating a Rational Rhapsody for MODAF project	1355
Customize the Rational Rhapsody table and matrix views for MODAF	1360
Creating stereotypes and using tags	1361
About creating table/matrix views in MODAF	1362
Create documentation for Your MODAF project with ReporterPLUS	1368
Setting up ReporterPLUS	1368

Table of Contents

Document structure	1369
Generating a MODAF document	1370
Troubleshooting ReporterPLUS and Rational Rhapsody for MODAF	1371
The Dependencies Linker	1372
Using the Dependencies Linker	1372
Troubleshoot the Dependencies Linker	1373
General troubleshooting	1374
Verify the Rational Rhapsody for MODAF Add On installation	1374
Find icons missing from diagram tools	1374
Check your Rational Rhapsody for MODAF model	1374
The Rational Rhapsody automotive industry tools	1377
AUTOSAR modeling	1377
The AUTOSAR workflow	1378
Creating an AUTOSAR project	1378
Creating AUTOSAR diagrams	1378
Checking an AUTOSAR model	1379
Import/export from/to AUTOSAR XML format	1379
The AutomotiveC profile	1380
Automotive-specific adaptor	1380
Automotive-specific stereotypes	1382
Simulink and StateMateBlock integration capabilities	1383
Fixed-point variable support	1383
AutomotiveC properties	1383
StateMateBlock in Rational Rhapsody	1385
Preparing a Rational StateMateBlock for Rational Rhapsody	1385
Creating the Rational StateMateBlock in Rational Rhapsody	1386
Connecting and synchronizing Rational StateMate and Rational Rhapsody	1388
Troubleshooting Rational StateMate with Rational Rhapsody	1389
IBM Rational DOORS interface	1391
Installation requirements	1392
Rational DOORS version 7.0	1392
Solaris-specific information	1392
Using Rational Rhapsody with Rational DOORS	1393
Configuring Rational Rhapsody and Rational DOORS with the Gateway wizard	1394
Requirements synchronization in Rational DOORS and Rational Rhapsody	1395
Navigating from Rational DOORS to Rational Rhapsody	1396

Rational DOORS projects	1396
Invoking the Rational DOORS interface	1396
Set export options	1397
Identify which formal modules to create	1397
Selecting Rational DOORS export options	1398
Linking the Rational DOORS data	1400
Information stored in Rational DOORS	1402
Rational DOORS information stored in Rational Rhapsody	1404
Data checking	1404
Problem Description window	1404
Mapping Requirements to imported elements	1406
Ending a Rational DOORS session	1407
Rational DOORS with Rational Rhapsody summary	1407
Rational Rose models	1409
Importing a Rational Rose model	1410
Setting up the XML map file for importing Rational Rose properties	1413
Incremental import of Rational Rose models	1414
Before the import process starts	1415
About processing time and project size	1416
Code import	1417
Merging imported code to the imported Rational Rose model	1418
How Rational Rose constructs and options map into a Rational Rhapsody model	1419
Imported association classes	1424
XMI exchange tools	1425
Using XMI in Rational Rhapsody development	1425
Exporting a model to XMI	1426
Examining the exported file	1428
Importing an XMI file to Rational Rhapsody	1429
More information	1430
Integrating Simulink components	1431
Importing Simulink components	1432
Integration of the Simulink-generated code	1433
Troubleshooting Simulink integration	1434

Creating Simulink S-functions with Rational Rhapsody	1435
Using Rational Rhapsody in conjunction with Simulink	1435
Creating a Simulink S-function	1435
S-function creation: behind the scenes	1436
Timing and S-Functions	1436
Limitations	1437
 The Rational Rhapsody command-line interface (CLI)	 1439
RhapsodyCL	1439
Interactive mode	1440
Socket mode	1440
Command-line syntax	1441
Switches	1441
Commands	1441
Order of commands	1442
Include commands in a script file	1442
Exit after use of command-line options	1442
Return codes	1443
Examples	1443
Command-line switches	1444
Command-line commands	1446
 Rational Rhapsody shortcuts	 1453
Accelerator keys	1453
Mnemonics	1454
Keyboard modifiers	1454
Standard Windows keyboard interaction	1455
Rational Rhapsody accelerator keys	1455
Application accelerators	1455
Accelerators and modifier usage in diagrams	1457
Code editor accelerators	1458
Useful Rational Rhapsody Windows shortcuts	1459
Changing settings to show the mnemonic underlining	1460
 Technical support	 1461

Contacting IBM Rational Software Support 1461

Prerequisites 1461

Contacting Support 1462

 About Rational Rhapsody 1464

 License Details 1464

Reporting Rational Rhapsody Problems from the Software 1465

Rational Rhapsody glossary 1467

Index 1531

Introduction to Rational Rhapsody

Welcome to IBM® Rational® Rhapsody®!

Systems engineers and software developers use Rational Rhapsody to create either embedded or real-time systems. However, Rational Rhapsody goes beyond defining requirements and designing a software solution. Rational Rhapsody actually implements the solution from design diagrams and automatically generating ANSI-compliant code that is optimized for the most widely used target environments.

With Rational Rhapsody, you have the ability to analyze the intended behavior of the application much earlier in the development cycle by generating code from UML and SysML diagrams and testing the application as you create it. Rational Rhapsody can be used for any of the following items:

- ◆ Reactivity for statecharts and events
- ◆ Time-based behavior for timeouts
- ◆ Multi-threaded architectures for active classes and protected classes
- ◆ Real-time environments for direct support for several real-time, operating systems (RTOS)

Rational Rhapsody is semantically complete. Most items that you draw in Rational Rhapsody UML diagrams, such as objects or events, have precise meaning in the underlying model. *Objects* are the structural building blocks of a system.

Rational Rhapsody translates these diagrams into source code in one of four high-level languages: C++, C, Ada, or Java. Rational Rhapsody then allows you to edit the generated code and dynamically roundtrip the changes back into the model and its graphical views. Rational Rhapsody supplies four editions to create specific types of projects depending on your job requirements.

- ♦ Rational Rhapsody Developer edition (C, C++, Java, and Ada are available, and this edition required for Eclipse users.)
- ♦ Rational Rhapsody Architect for Software edition (described in [Creating projects in Rational Rhapsody Architect for Software](#))
- ♦ Rational Rhapsody Architect for Systems Engineers edition (described in [Creating projects in Rational Rhapsody Architect for Systems Engineers](#))
- ♦ Rational Rhapsody Designer for Systems Engineers edition (described in [Creating projects in Rational Rhapsody Designer for Systems Engineers](#))

Only one Rational Rhapsody edition can be selected during installation.

Rational Rhapsody features

Rational Rhapsody includes the following features:

- ♦ UML[®], SysML[™], and Functional C design modeling environment with Domain-Specific Language (DSL) support including DoDAF*, MODAF*, and AUTOSAR*.
- ♦ Predefined *profiles* supplying a coherent set of tags, stereotypes, and constraints for a specific type of project. For more information, see [Profiles](#).
- ♦ MathWorks Simulink[®] Interface, SDL Interface, and StateMate[®] Interface available with the Interfaces Add On can be used to validate your complete architecture while using the best-in-class tools for control engineering, protocol development, and functional system design.
- ♦ Model verification with full model simulation and execution.
- ♦ Static checking to ensure that the design is consistent.
- ♦ Full application generation of C, C++, Java, and Ada in an integrated design environment.
- ♦ Easily customizable real-time framework that separates high-level application design from platform-specific implementation details with numerous adapters available such as VxWorks, Windows CE, and Integrity. A full list is available in the Rational Rhapsody release notes, and in addition you can create your own adapter.
- ♦ Requirements modeling and traceability features with integration to leading requirements management tools such as DOORS^{®*} or text-based tools such as Microsoft[®] Word.
- ♦ Easily integrate and create models from your existing C, C++, Java, and Ada code into the modeling environment using reverse engineering and code visualization.

- ♦ Integration with leading IDEs such as Eclipse, Wind River[®] Workbench, and Green Hills[®] Multi[®].
- ♦ Dynamic model-code associativity enabling design to be done using either code or diagrams providing maximum flexibility while ensuring the two remain synchronized.
- ♦ Improved test productivity and early detection of defects using Rational Rhapsody TestConductor[™] to automate tedious testing tasks, define tests with code and graphically with sequence diagrams, statecharts, activity diagrams and flowcharts; and execute the tests interactively or in batch mode.
- ♦ XMI* (XML Metadata Interchange) and IBM[®] Rational Rose[®]* importing for integration of legacy systems and reuse of existing code.
- ♦ Full Configuration Management Interface* support with advanced graphical difference and merging capabilities for use with tools such as IBM[®] Rational[®] Synergy[™] or IBM[®] Rational[®] ClearCase[®].
- ♦ Support for software product lines using class and object variants for components
- ♦ Customization to meet your specific development needs using the Java API.
- ♦ Generation of documentation using a range of tools, from a simple RTF report generator to the full customization with Rational[®] Rhapsody[®] ReporterPLUS[™].

* Capabilities are provided by optional add-ons.

UML design essentials

The Developer edition for C++, C, Java, and Ada and [The Rational Rhapsody specialized editions](#) support UML to design your models.

The Unified Modeling Language (UML) is a third-generation modeling language for describing complex systems. The Object Management Group[®] (OMG[®]) adopted the UML as the industry standard for describing object-oriented systems in the fall of 1997. For more information on the OMG, see their Web site at <http://www.omg.org>.

UML defines a set of diagrams by which you can specify the objects, messages, relationships, and constraints in your system. Each diagram emphasizes a different aspect or view of the system elements. For example, a UML sequence diagram focuses on the message flow between objects during a particular scenario, whereas an object model diagram defines classes, their operations, relations, and other elements.

UML diagrams

The UML specification includes the following diagrams:

- ♦ **Use case diagram** shows typical interactions between the system being designed and external users or actors. Rational Rhapsody can generate code for actors in use case diagrams to be used for testing a model.
- ♦ **Object model diagram** shows the static structure of a system: the objects in the system and their associations and operations, and the relationships between classes and any constraints on those relationships.
- ♦ **Sequence diagram** shows the message flow of objects over time for a particular scenario.
- ♦ **Collaboration diagram** provides the same information as a sequence diagram but emphasizes structure, whereas a sequence diagram emphasizes time.
- ♦ **Statechart** defines all the states that an object can occupy and the messages or events that cause the transition of the object from one state to another.
- ♦ **Activity diagram** specifies a workflow or process for classes, use cases, and operations. Activity diagrams provide similar information to statecharts, but are better for linear step-by-step processes, whereas statecharts are better suited for non-linear or event-driven processes.
- ♦ **Component diagram** describes the organization of the software units and the dependencies among these units.
- ♦ **Deployment diagram** depicts the nodes in the final system architecture and the connections between them. Nodes include processors that execute software components, and the devices that those components control.
- ♦ **Structure diagram** models the structure of a composite class; any class or object that has an OMD can have a structure diagram. Object model diagrams focus more on the specification of classes, whereas structure diagrams focus on the objects used in the model.

In addition, a Flow Chart is available in the Rational Rhapsody product. You can use a flow chart to describe a function or class operation and for code generation.

UML views

You can use Rational Rhapsody to draw UML diagrams that provide different views of your system. By editing the UML diagrams in Rational Rhapsody to create increasingly complex views, you can add layers of perspective, detail, and specificity to your model until you have a complete solution.

Structural views

Structural views show model elements and their relationships to each other. Model elements include classes, use cases, components, and actors; their relationships include dependencies, inheritances, associations, aggregation, and composition.

The following UML diagrams provide structural views:

- ◆ Use case diagram
- ◆ Object model diagram
- ◆ Structure diagrams
- ◆ Component diagram
- ◆ Deployment diagram

Dynamic behavior views

Dynamic behavior views describe the system behavior. This includes state behavior, such as the different states a class occupies, state transitions, forks and joins, and actions within a state; and interactions, such as the collaborations occurring between classes during a particular scenario.

The following UML diagrams provide dynamic behavior views of the model:

- ◆ Statechart
- ◆ Activity diagram
- ◆ Sequence diagram
- ◆ Collaboration diagram

Model management views

Model management views show the hierarchical organization of the model. Object model diagrams provide a model management view.

Diagrams in Rational Rhapsody

Rational Rhapsody includes a graphic editor for each of the UML diagrams, enabling you to create detailed views of your model. The graphic editors not only capture the design of your system, but also generate implementation code.

Note

Rational Rhapsody diagrams have varying levels of code generation ability. Model elements and implementation code can also be created from the browser.

Because Rational Rhapsody maintains a tight model-code associativity, you can easily generate updated code when you make changes to the model. You can also edit code directly and bring those changes into the model via the roundtrip feature. For more information about model-code associativity, see [Basic code generation concepts](#).

Partially constructive diagrams

Partially constructive diagrams generate code for some, but not all of the elements in the diagram. Partially constructive diagrams include the following diagrams:

- ◆ Use case diagrams
- ◆ Sequence diagrams
- ◆ Collaboration diagrams

Fully constructive diagrams

Fully constructive diagrams generate code for every element in the diagram. Fully constructive diagrams include the following diagrams:

- ◆ Object model diagrams
- ◆ Component diagrams
- ◆ Statecharts
- ◆ Activity diagrams

Specify a model with Rational Rhapsody

To create a working model, you must create at a minimum an object model diagram. An object model diagram generates the code necessary for a minimally functioning model.

A properly designed implementation, however, includes at a minimum object model diagrams, statecharts or activity diagrams, and component diagrams. Object model diagrams and statecharts can be considered to be design diagrams, because they are most often used in the design phase of a project. Other diagrams are more helpful in other phases. For example, use case and sequence diagrams are useful in the requirements analysis phase, where use case diagrams document structural requirements and sequence diagrams document behavioral requirements.

Development methodology

A development methodology is a combination of a *process*, a tool, and a modeling language. Rational Rhapsody is a UML-compliant modeling tool that is process-neutral and supports the most common phases of any good development methodology. However, Rational Rhapsody is particularly well-suited to an iterative process in which you build a number of model prototypes, test, debug, reanalyze, and then rebuild the model any number of times, all within a single development environment.

The ROPES[™] process is an example of an iterative process that illustrates the use of Rational Rhapsody and the UML across all typical process phases and activities. The following sections provide a general overview of the phases involved in the ROPES process (including the subtasks involved in general analysis, design, implementation, and test phases) and the Rational Rhapsody tools appropriate for each phase. The Web site for IBM Rational modeling products contains detailed information on ROPES.

Analysis

In the analysis phase, you define a problem, its possible solutions, and their characteristics.

Requirements analysis

Begin with the requirements analysis to identify the system requirements. What are the primary system functions or system usages? Use case diagrams can capture these along with the external actors that interact with the system.

Describe the expected behavior of the system as a whole by creating a series of “black-box” sequence diagrams. In these, you will define the sequence of messages between external actors and the system as a whole. You can create a number of sequence diagrams for each use case, where each sequence diagram represents one scenario that could occur while carrying out that use case. You can also use collaboration diagrams to specify the expected behavior of the system.

Use the black-box sequence diagrams as the basis for creating statecharts, which realize all possible scenarios. Statecharts specify the behavior of each object, or object implementation, as opposed to sequence diagrams, which concentrate on requirements-based scenarios. Sequence diagrams also serve as the primary test data in the testing phase; in later stages, use them to test whether your system as a whole responds properly to the external messages that come into it. You can also use activity diagrams to realize all possible scenarios. To review and analyze the requirements in your project, you can use the advanced search facility and the table and matrix tools.

Object analysis

While you are capturing system requirements, you should also define the entities and structural relationships that will exist in the application you are creating and its domain or environment. This should result in a structural (static) model of the system (a logical object model of the system).

Determine the subsystems of your system. What are their responsibilities and relationships? These subsystems become the basis of the packages, or collections of classes, within your system.

Determine the key objects or classes in these subsystems and define their responsibilities, descriptions, and their relations to other classes. Use object model diagrams to create these classes and their relations. Using sequence diagrams and statecharts, define the behavior and interactions of these essential objects.

You can also use the code generation and animation tools to execute and debug these higher-level analysis models.

Design

In the analysis phase, you came up with several possible solutions to your problem. In the design phase, you choose one of those solutions and define how it will be implemented. As with the analysis phase, the design phase has more than one component. Just as the analysis phase should conclude with some result, a full set of use cases and a logical object model of the system, the design phase should also provide results: task and deployment models, and more refined logical object models.

Architectural design

Define the major architectural pieces of the system. What are the high-level parts? Define what the system domains are and which key classes fit in each domain. Which are your composite classes? In this analysis, you should also map classes, packages, and components to the relevant physical parts of your system (the processors and devices). Define which libraries and executables are necessary in your model. You are creating the task and deployment models for your system. To help with this, you can apply UML design patterns as appropriate for your system.

Mechanistic design

Continue to detail the internal workings of your system, breaking it down into smaller pieces and more classes, if necessary. Use “white-box” sequence diagrams to depict the class interactions within the system. Define the collaborations that are required to realize certain core cases by creating collaboration diagrams. Add to your model the “glue” objects that are used in the UML design patterns that you use. Again, you can use the code generation and animation tools to debug and test the model at this point. Your end result should be a more refined set of logical object models.

Detailed design

Continue to fill in the details of your design. Get your individual classes working; fully define their constraints, internal data structures, and message passing behavior. Use activity diagrams and statecharts to define correct behavior. At this stage, you will probably begin typing in extra code in the implementation boxes in various diagrams. Use component diagrams to define the physical artifacts of your system and to include the libraries, executables, or legacy code you have deemed necessary for your model. Make low-level decisions about implementations, such as choosing static or dynamic instantiations. This should result in a more refined logical object model (or models) of your system.

Implementation

The implementation phase is essentially the code generation and unit testing phase. Using Rational Rhapsody, write the code that is not generated automatically, such as the bodies of non-statechart operations. These include constructors, destructors, object methods, and global functions. Use the animation and tracing tools to test and debug sections of code and to make decisions about any optimization trade-offs.

The testing phase

In the testing phase, you determine not only whether your model is working, but whether it meets the requirements that you set in the analysis phase. Your end result should be a working system.

Rational Rhapsody includes the following features to assist with the testing phase:

- ♦ **Animator** creates test scripts to apply external test stimuli to the system.
- ♦ **Tracer** performs white, gray, and black-box regression testing. It also provides performance testing based on timing annotations or on a simulated time facility.
- ♦ **Sequence diagram comparison** Automatically compares requirement sequences with implementation sequences.

Rational Rhapsody tools

Rational Rhapsody consists of the following set of tools that interact with each other to give you a complete software design environment:

- ♦ [The Rational Rhapsody browser](#)
- ♦ [The Favorites browser](#)
- ♦ [Graphic editors](#)
- ♦ [Code generator](#)
- ♦ [Animator](#)
- ♦ [Utilities](#), including reverse engineering, Web-enabling devices, XMI generation, COM and CORBA[®] support, and Visual Basic[®] for Applications
- ♦ [Third-party interfaces](#) such as Eclipse, Rational Rose import, and CM tools

The Rational Rhapsody browser

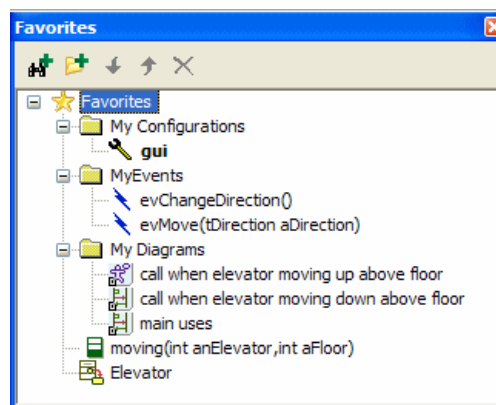
The Rational Rhapsody browser shows a comprehensive display of the system with a clear overview of your entire model. Views filter the display to optimize usability for a particular task. During an animation session, the browser dynamically displays object instances as the model executes.

For more details about the uses of the browser, see [Browser](#) and [Browser filter](#); and for details on the model display features for the browser, see [Browser techniques for project management](#).

The Favorites browser

You can use the Favorites browser to create a favorites list, which is a list of items (model elements) that you are most interested in for the opened Rational Rhapsody model. This is analogous to the favorites functionality for a Web browser. You might find the Favorites browser most useful with Rational Rhapsody models that are very large, which can make it difficult to find commonly used model elements in the Rational Rhapsody browser. The Favorites browser should help you manage large and complex projects by making it easier to focus on and easily access model elements of particular interest to you.

The following figure shows a sample Favorites browser:



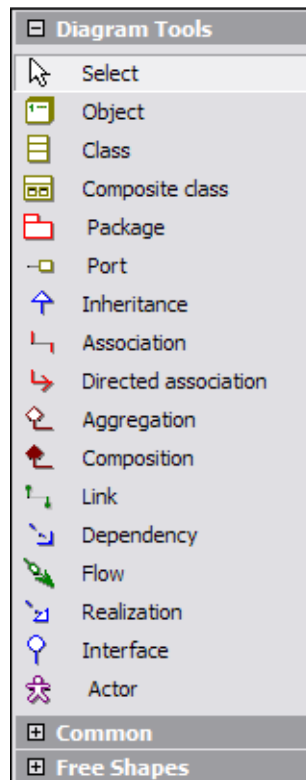
Your favorites list is saved in the <projectname>.rpw file, as well as the position and visibility of the Favorites browser, so that when you open the project the next time, your settings are automatically in place. Note that when multiple projects are loaded, the Favorites browser shows the favorites list for the active project, as described in [Setting the active project](#).

For more information about the Favorites browser, see [The Favorites browser](#).

Diagram tools

When you have a diagram open on the drawing area in Rational Rhapsody, a panel of diagram drawing tools for the currently displayed diagram type also appears. You can move this panel to different locations and close it.

Different tools are available depending on the type of diagram displayed in the drawing area.



This accordion menu also contains a **Common** section for tools for common additions to diagrams and a **Free Shapes** section for tools that let you draw elements freehand in a diagram.

Graphic editors

You can use the graphic editors to analyze, design, and construct the system using UML diagrams. Diagrams enable you to observe the model from several different perspectives, like turning a cube in your hand to view its different sides. Depending on its focus, a diagram might show only a subset of the total number of classes, objects, relationships, or behaviors in the model. Together, the diagrams represent a complete design.

Rational Rhapsody adds the objects created in diagrams to the Rational Rhapsody project, if they do not already exist. Conversely, Rational Rhapsody removes elements from the project when they are deleted from a diagram. However, you can also add existing elements to diagrams that do not need to be added to the project, and remove elements from a diagram without deleting them from the model repository.

- ♦ **Use case diagram editor** provides tools for creating use case diagrams, which show the use cases of the system and the actors that interact with them. See [Use case diagrams](#).
- ♦ **Object model diagram editor** provides tools for creating object model diagrams, which are logical views showing the static structure of the classes and objects in an object-oriented software system and the relationships between them. See [Object model diagrams](#).
- ♦ **Sequence diagram editor** provides tools for creating sequence diagrams, which show interactions between objects in the form of messages passed between the objects over time. If you run animated code with the Animator, you can watch messages being passed between objects as the model runs. See [Sequence diagrams](#).
- ♦ **Collaboration diagram editor** provides tools for creating collaboration diagrams, which describe how different kinds of objects and associations are used to accomplish a particular task. Collaboration diagrams and sequence diagrams are both interaction diagrams that show sequences. Sequence diagrams have a time component, whereas collaboration diagrams do not.

Like sequence diagrams, collaboration diagrams show the message flow between different classes. However, collaboration diagrams emphasize object relationships whereas sequence diagrams emphasize the order of the message flow. For a particular task or interaction, a collaboration diagram can also show the individual objects that are created, destroyed, or exist continuously for the duration of the task. See [Collaboration diagrams](#).

- ♦ **Statechart editor** provides tools for creating statecharts, which define the behaviors of individual classes in the system.

Statecharts show the states of a class in a given context, events that can cause transitions from one state to another, and actions that result from state transitions. Rhapsody generates function bodies from information entered into statecharts. If you run animated code with the animator, you can watch an object change states as it reacts to various messages, events, and triggered operations that you generate. See [Statecharts](#)

- ♦ **Activity diagram editor** provides tools for creating activity diagrams. Activity diagrams show the lifetime behavior of an object, or the procedure that is executed by an operation in terms of a process flow, rather than as a set of reactions to incoming events. When a system is not event-driven, use activity diagrams rather than statecharts to specify behavior. See [Activity diagrams](#).
- ♦ **Component diagram editor** provides tools for creating component diagrams, which show the dependencies among software components, such as library or executable components. Component diagrams can also show component dependencies, such as the files (or other units) that are contained by a component, or the connections or interfaces among components. See [Component diagrams](#).
- ♦ **Deployment diagram editor** provides tools for creating deployment diagrams, which show the run-time physical architecture of the system. The physical architecture of a running system consists of the configuration of run-time processing elements and the software components, processes, and objects that live on them. A deployment diagram graphs the nodes in the system, representing various processors, connected by communication associations. See [Deployment diagrams](#).
- ♦ **Structure diagram editor** provides tools for creating structure diagrams, which model the structure of a composite classes. See [Structure diagrams](#).

Note

All the diagrams use UML notation.

The FunctionalC profile has these diagrams available to construct a C model:

- ♦ Use case diagram
- ♦ Statechart
- ♦ Build diagram
- ♦ Call Graph diagram
- ♦ Flow Chart
- ♦ Message diagram
- ♦ File diagram

Code generator

The code generator synthesizes complete production-quality code from the model to free you from low-level coding activities. Rational Rhapsody generates code primarily from OMDs and statecharts, but also from activity and other diagrams. Allowing the tool to generate code automatically for you lets you concentrate on higher-level system analysis and design tasks. For more information, see [Basic code generation concepts](#).

Animator

The animation facility lets you debug and verify your software at the design level rather than the compiler level. For more information, see [Animation](#).

Utilities

In addition to the core UML-based design features, Rational Rhapsody provides a number of utilities to assist with development including the following utilities:

- ♦ Dynamic reverse engineering (see [Reverse engineering](#))
- ♦ Roundtrip (see [Basic code generation concepts](#))
- ♦ Check model (see [Checks](#))
- ♦ Web-enabling of Rational Rhapsody models (see [Managing Web-enabled devices](#))
- ♦ Standard and customizable report generation with Rational Rhapsody Reporter and ReporterPLUS (see [Reports](#))
- ♦ Import of model elements from libraries and external source files including [XML exchange tools](#), [StateMateBlock in Rational Rhapsody](#), and [Importing DoDAF diagrams from Rational System Architect](#).
- ♦ **Add to Model** and multiuser collaboration
- ♦ Component download
- ♦ File comparison and merging, as described in [Parallel project development](#) and [DiffMerge](#)
- ♦ Web Collaboration, as described in [Viewing and Controlling of a Model via the Internet](#)

Third-party interfaces

The following third-party software interfaces can be accessed through the Rational Rhapsody interface:

- ◆ Configuration management tools including support for the Microsoft Source Code Control (SCC) standard
- ◆ Visual Studio standard or professional edition (see [Visual Studio IDE with Rational Rhapsody](#))
- ◆ Integrated VBA Interface for development and macros
- ◆ IDE interface to the Tornado™ development environment (see [Co-debugging with Tornado](#))
- ◆ Code editors (such as CodeWright™)
- ◆ Source debuggers (in addition to IDEs)
- ◆ Eclipse (for information about the Eclipse implementations, see [Eclipse platform integration](#) and [Using the Rational Rhapsody Workflow Integration with Eclipse](#))

Rational Rhapsody windows

When creating or editing a project, the Rational Rhapsody workspace has the following windows:

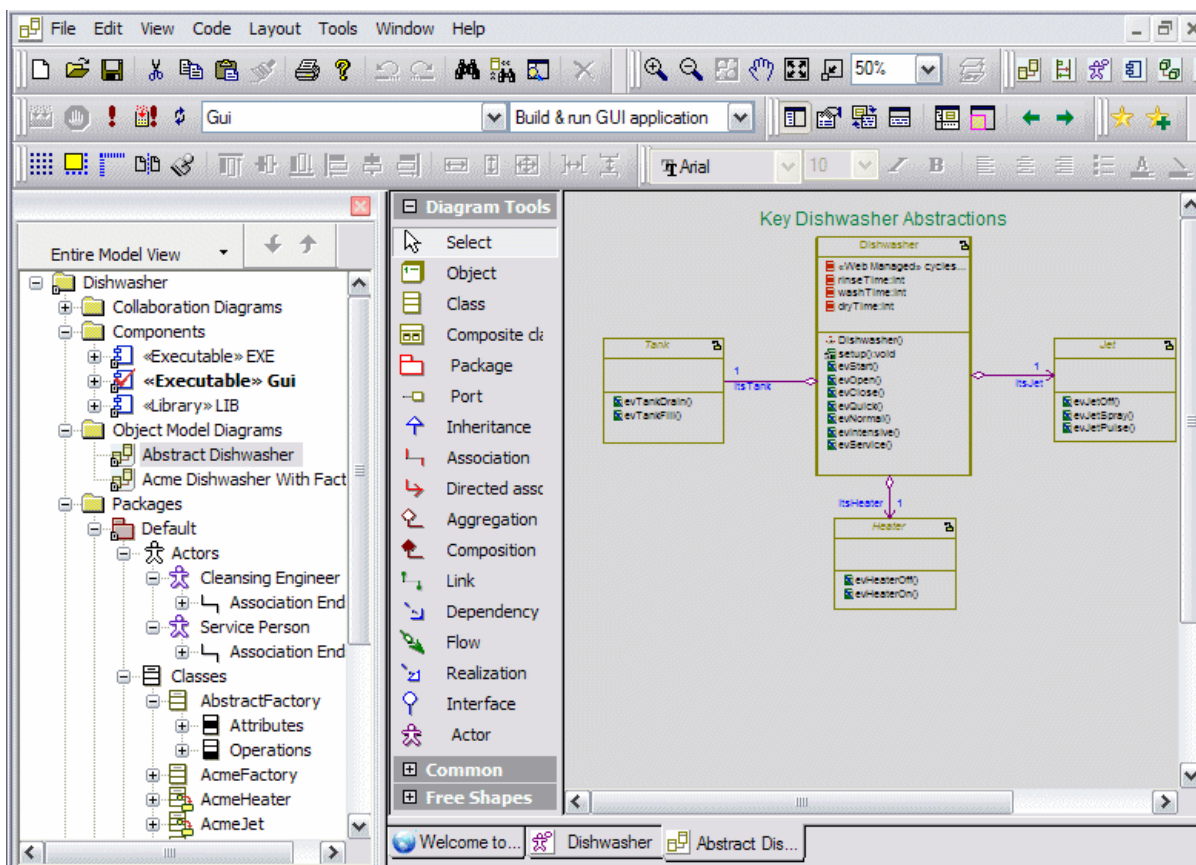
- ♦ **Menu bar** lists the primary functions as File, Edit, View, Code, Tools, Layout, Windows, and Help. Many of the Rational Rhapsody functions available on the menus are also accessible from [Rational Rhapsody shortcuts](#) and buttons across the top of the Rational Rhapsody interface, as described in [Rational Rhapsody project tools](#).
- ♦ **Browser** displays the contents of the project and has several views to choose from. See [Browser](#).
- ♦ **Diagram drawing area** contains the diagram editor windows, which can be moved and resized. See [Diagram drawing area](#).
- ♦ **Diagram Tools** contains the drawing tools for each diagram type and opens in a panel next to the diagram open in the drawing area in Rational Rhapsody. See [Diagram tools](#). Different buttons are displayed in the panel depending on the type of diagram displayed in the drawing area. See the descriptions of the diagrams for explanations of each diagram drawing tool. This accordion menu also contains a **Common** section for tools for common additions to diagrams and a **Free Shapes** section for tools that let you draw elements freehand in a diagram.
- ♦ **Diagram Navigator** provides a bird's eye view of the diagram that is currently displayed. See [Diagram navigator](#).
- ♦ **Output window** has several tabs, each displaying different types of Rational Rhapsody output including search results. See [Output window](#).

In addition, you can open two windows from the View menu:

- ♦ **Features window** displays details of selected model element. By default, it displays as a floating window, but you can dock it to the main window in any position. See [The Features window](#).
- ♦ **Active code view** generates and displays code for the selected model element. See [Active Code View window](#).

When you open Rational Rhapsody for the first time, the [Welcome window](#) displays.

The following figure shows the default arrangement of the Rational Rhapsody windows.



Note the following information:

- ♦ You can reposition each window within the Rational Rhapsody workspace to suit your preferred work style.
- ♦ To dock or undock a window quickly, double-click the title bar.
- ♦ To reposition a window, click the title bar and drag-and-drop the window to the intended location.

View menu commands

The Rational Rhapsody View menu allows you to customize the display of the Rational Rhapsody interface areas.

View > Status Bar

The status bar at bottom of the main window displays the current mode (for example, GE MODE) and the date and time. Use this menu command to toggle the status display on and off.

View > Favorites

Use this menu command to display [The Favorites browser](#) for your project.

View > Features

Use this menu command to display [The Features window](#) for a selected project element.

View > Description

Use this menu command to display and edit the description of a selected element as it is on the Description tab of the Features window.

View > Tags

Use this menu command to display and edit the tags of a selected element as it is on the Tags tab of the Features window.

View > Relations

Use this menu command to display and edit the relations of a selected element as it is on the Relations tab of the Features window.

View > Properties

Use this menu command to display the properties associated with a selected project element. For more information, see [Properties tab](#).

View > Browser

The browser displays a tree structure of your project. You can also use this area to edit and restructure your model. For more information, see [Creating hyperlinks on the Rational Rhapsody browser](#).

View > Label Mode

If you want to work exclusively with the label names of the elements, choose the **Label Mode** command on the View menu. For more information about this work mode, see [Label mode](#).

View > Workbar Mode

The tabs above the diagram drawing area are displayed when the Workbar Mode is selected. To switch off the tabs in the drawing area, clear the check mark next to **Workbar Mode** on the View menu.

View > Gradient Mode

The Gradient Mode displays the project diagrams with a shaded background.

View > Full Screen Mode

The Full Screen Mode displays Rational Rhapsody as the only application on your computer screen. To redisplay the other programs you have running and end this mode, press the **Esc** key.

View > Maintain Window Content

The part of the diagram displayed in the drawing area is called the *viewport*. To specify whether to display the viewport regardless of any sizing or reposition of the diagram drawing window, choose the **Maintain Window Content** on the View menu. For more information, see [Maintaining the window content](#).

View > Output Window

Use this menu command to open the Output window manually. The Output Window opens automatically when a process produces output for display. For more information, see [Output window](#).

View > Active Code View

To display code for an element selected in the browser, choose View Active Code View. For more information, see [Active Code View window](#).

View > Bird's Eye

Use this menu command to show or hide the Bird's Eye window, a view of the entire diagram with a rectangular focus area showing the portion of the diagram is currently displayed in the drawing area. For more information about this feature, see [The Bird's Eye \(diagram navigator\)](#)

View > Pop Context

To return to the origin point of a hyperlink, choose this View menu command or press **Ctrl+P**.

View > Toolbars > Diagrams

Use this menu command to show or hide the diagram control toolbar.

View > Toolbars > Code

Use this menu command to show or hide the code access toolbar.

View > Toolbars > Browser Filter

Use this menu command focus the browser display on the portion of the project related to your current task.

View > Toolbars > Start Target Monitoring

Target monitoring is background animation of a C application from a target with unknown or limited resources or limited monitoring for application execution on the target. This menu command displays the tools used to set up and start monitoring a C application created using the MicroC profile.

View > Toolbars > Target Monitoring

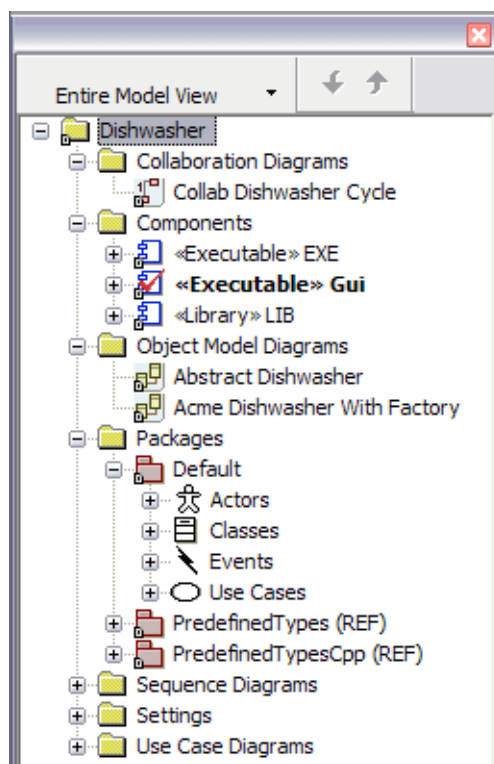
Use this toolbar to watch the C application execution on the target. However, this feature cannot be used to control application execution.

View > Toolbars > VBA

Use this menu command to display the Rational Rhapsody Visual Basic for Applications (VBA) Interface editor and macro creation facility.

Browser

The browser displays a hierarchy of your project and provides easy access to the elements and diagrams it contains. You can filter the display with several view options. [Browser techniques for project management](#) provides detailed descriptions of browser elements and views.



By default, the browser is docked at the upper, left corner of the Rational Rhapsody main window.

- ♦ To open the browser, choose **View > Browser**.
- ♦ Choose **Tools > Browser** open multiple instances of the browser to simplify the process of navigating between elements. This feature is particularly useful during animation.

In addition, you can select multiple elements in the browser and perform any of these operations on all of them:

- ♦ Move them to another browser element by dragging and dropping them over the target element
- ♦ Copy them to another browser element by dragging and dropping them over the target element
- ♦ Delete all of them at the same time

Diagram drawing area

The drawing area displays the graphic editors and code editors. Editors can be moved and resized within the drawing area. When you open more than one editor, tabs are displayed at the bottom of the drawing area so you can easily move between the open diagrams or generated code files. In addition, you can tile or cascade the windows that contain the different diagrams.

Each diagram includes a title bar, which contains the name of the diagram and its type. A modified diagram has an asterisk (*) added to the end of its name in the title bar.

For a description of the graphic editor windows, see [Graphic editors](#). For a description of the default code editor, see [The Internal code editor](#).

Maintaining the window content

When you resize the drawing area (for example, to increase the available drawing space), some of the diagram might move out of the visible area of the window. The part of the diagram displayed in the window is called the *viewport*. You can specify whether to display the viewport regardless of window manipulation using two different methods:

- ♦ Choose **View > Maintain Window Content**.
- ♦ Set the `General::Graphics::MaintainWindowContent` property to Checked.

Using this functionality, the elements are scaled according to the zoom factor so you see the same elements in the window regardless of scaling.

Changing the drawing area window display

Rational Rhapsody uses these standard Microsoft Windows features to change the shape and display design of the diagram drawing area:

- ♦ Manual resizing by dragging the edge of the window
- ♦ Maximize/Minimize and Restore buttons
- ♦ **Window > Tile**
- ♦ **Window > Cascade**

Diagram navigator

The Diagram Navigator provides a bird's eye view of the diagram that is currently being viewed. This can be very useful when dealing with very large diagrams, allowing you to view specific areas of the diagram in the drawing area, while, at the same time, maintaining a view of the diagram in its entirety.

The Diagram Navigator contains a depiction of the entire diagram being viewed, and a rectangle viewport that indicates which portion of the diagram is currently visible in the drawing area.

For detailed information about using the Diagram Navigator window, see [The Bird's Eye \(diagram navigator\)](#).

Output window

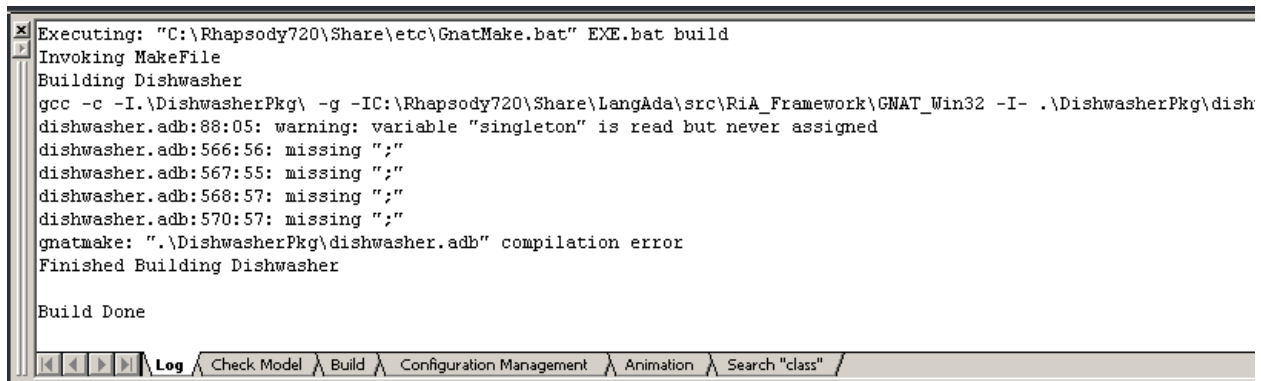
The Output window is where Rational Rhapsody displays various output messages. You can use the tabs on the Output window to navigate easily among the different types of output messages:

- ♦ The [Log tab](#) shows all the messages from all the other tabs of the Output window (except for **Search Results**) in text (meaning non-tabular) format.
- ♦ The [Build tab](#) shows the messages related to building an application in tabular format.
- ♦ The [Check Model tab](#) shows the messages related to checking the code for a model in tabular format.
- ♦ The [Configuration Management tab](#) shows the messages related to configuration management actions for a model in text format.
- ♦ The [Animation tab](#) shows the message related to animating a model in text format.
- ♦ The [Search Results tab](#) shows the results from searches of your model in tabular format. Note that this tab might not appear until you perform a search.

By default, the Output window is located at the bottom portion of the main Rational Rhapsody window. Also by default, when you generate, build, or run an application; do a search, a CM action, or a check model, Rational Rhapsody opens the Output window.

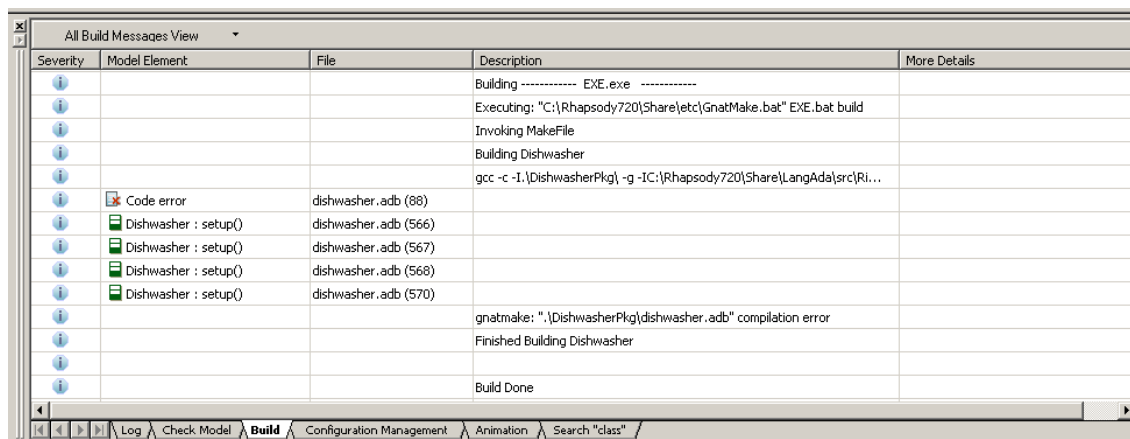
Log tab

The **Log** tab serves as a console log. It shows all the messages from all the other tabs of the Output window (except for **Search Results**) in text (meaning non-tabular) format. The messages that appear on the **Check Model**, **Build**, **Configuration Management**, and **Animation** tabs appear on the **Log** tab too, but always in text format. For the check model and build functions, you can view their messages on the **Check Model** and **Build** tabs in tabular format or on the **Log** tab in text format, depending on your preference. The **Log** tab displays messages in text format after a build function is performed.



Note that you can right-click on the **Log** tab to use the **Clear**, **Copy**, **Paste**, and **Hide** commands.

The following figure shows the **Build** tab for the same build function. As you can see, the messages provide the same type of information, though the presentation is in a tabular format.

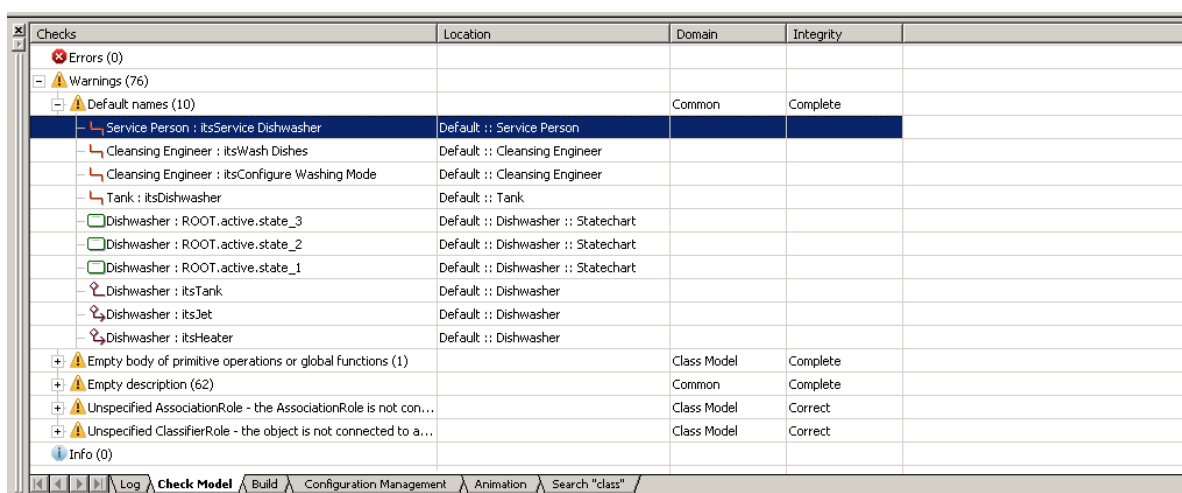


On the **Log**, **Check Model**, and **Build** tabs, you can double-click an item on the tab and, if possible, Rational Rhapsody opens either the relevant model element (for example, the Features window for an association that might be causing an error) or to the code source. From whichever opens, you can make corrections or view the item more closely.

Check Model tab

Rational Rhapsody analyzes and organizes the results of checking the code for a model and displays the results on the **Check Model** tab. Check messages are grouped by a severity hierarchy, and provide you with the location, domain, and integrity for an item, where possible.

Before generating code, Rational Rhapsody automatically performs certain checks for the correctness and completeness of the model. In addition, you can perform selected checks at any time during the design process. To check the code for a model, choose **Tools > Check Model** and then the name of the configuration for the model. For more information about checking the code for a model, see [Checks](#).

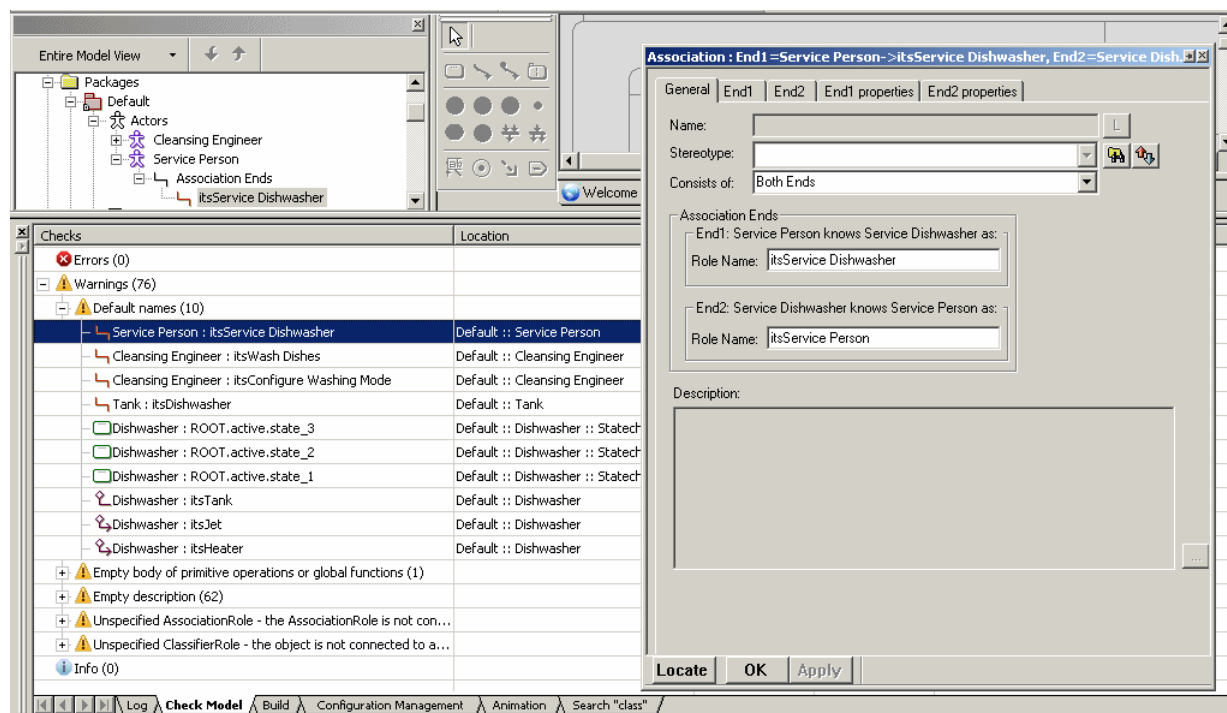


Check	Location	Domain	Integrity
Errors (0)			
Warnings (76)			
Default names (10)		Common	Complete
Service Person : itsService Dishwasher	Default :: Service Person		
Cleansing Engineer : itsWash Dishes	Default :: Cleansing Engineer		
Cleansing Engineer : itsConfigure Washing Mode	Default :: Cleansing Engineer		
Tank : itsDishwasher	Default :: Tank		
Dishwasher : ROOT.active.state_3	Default :: Dishwasher :: Statechart		
Dishwasher : ROOT.active.state_2	Default :: Dishwasher :: Statechart		
Dishwasher : ROOT.active.state_1	Default :: Dishwasher :: Statechart		
Dishwasher : itsTank	Default :: Dishwasher		
Dishwasher : itsJet	Default :: Dishwasher		
Dishwasher : itsHeater	Default :: Dishwasher		
Empty body of primitive operations or global functions (1)		Class Model	Complete
Empty description (62)		Common	Complete
Unspecified AssociationRole - the AssociationRole is not con...		Class Model	Correct
Unspecified ClassifierRole - the object is not connected to a...		Class Model	Correct
Info (0)			

The following table explains what type of information available on the **Check Model** tab.

Column	Explanation
Checks	Shows the check results tree grouped by three levels and in the hierarchy shown as follows: Level 1, Severity: Where the elements listed are grouped under the three optional severity levels as follows:  Errors  Warnings  Info The grouping is determined by the severity level of each added check. If there is a + icon next to a label, click it to expand or collapse the list. In addition, to the right of each severity level name is the number of problems in the security level. Level 2, Checks: Where each check that produces errors/problems when it is executed is shown in the list indented under its relevant severity level (for example, Default Names under Warnings). A check is considered a group that holds under it all its related problems. Each check in the list shows also its domain and integrity properties. In addition, to the right of each check name is the number of problems the check contains. There are 10 Default Name warnings. Level 3: Check Elements, where each problem found when performing a specific check is shown as a list item under the relevant check name. Problems are represented by model elements and shown with the relevant name, type icon, and model location path (for example, ServicePerson: itsServiceDishwasher, the first item in the Default Names check group in the Warnings severity level).
Location	Shows, for each problem found, the location of an element in the model.
Domain	Shows, for each check in the list, its domain property. This includes domains that are from user-defined checks.
Integrity	Shows, for each check in the list, its integrity property value.

You can double-click an item on the **Check Model** tab and if possible, Rational Rhapsody brings you to the relevant model element (for example, the Features window for an association) or to the code on which you can make corrections or view the item more closely. Note that if you click a level heading, you expand or collapse the list for the level.



Note that you can right-click on the Check Model tab to use the **Copy All** and **Clear All** commands.

Build tab

The **Build** tab shows the messages related to building an application.

Severity	Model Element	File	Description	More Details
			Building ----- EXE.exe -----□	
			Executing: "C:\Rhapsody720\Share\etc\msmake.bat" EXE.mak build□	
			Setting environment for using Microsoft Visual C++ tools.□	
			AbstractFactory.cpp□	
			AcmeFactory.cpp□	
			AcmeHeater.cpp□	
			AcmeJet.cpp□	
			AcmeTank.cpp□	
			Dishwasher.cpp□	
	Dishwasher : setup()	Dishwasher.cpp (266)	C2146: syntax error : missing ';' before identifier 'setWashTime'□	
	Dishwasher : setup()	Dishwasher.cpp (267)	C2146: syntax error : missing ';' before identifier 'setDryTime'□	
	Dishwasher : setup()	Dishwasher.cpp (268)	C2146: syntax error : missing ';' before identifier 'cycles'□	
	Code error	Dishwasher.cpp (270)	C2143: syntax error : missing ';' before '}'□	
			NMAKE : fatal error U1077: 'cl' : return code '0x2'□	
			Stop.□	
			□	
			Build Done□	

This tabular view shows the following types of information:

Column	Explanation
Severity	Error messages. Errors appear when the model fails to build. You must fix errors before the model can be built. Rational Rhapsody parses the information provided by the compiler to develop the list of error messages. Note that there can be two types of error (and warning) messages: model element and code error. Model element-type errors (and warnings) are those that Rational Rhapsody can correspond to specific model elements in a project. Code error-type errors are those that Rational Rhapsody cannot find any corresponding model element. Warning messages. Warnings have no effect on whether the model is built, but you should review them and address them if necessary as they might have an effect on whether the model builds as expected. Rational Rhapsody parses the information provided by the compiler to develop the list of warning messages. Informational messages. These messages are messages that are not warnings or errors and they have no effect on the building of the model.
Model Element	Applicable to error and warning messages only, shows the Rational Rhapsody model element and its applicable Rational Rhapsody icon. If no related model element is found, the error is assumed to be a code error-type error.
File	Applicable to error and warning messages only, shows the file name and line number where an error/warning was found.
Description	Descriptions are provided by the compiler.
More Details	Applicable to error and warning messages only, and only if available, show more details as provided by the compiler.

By default, you see all the messages for a build. If this is not the case, you can select **All Build Messages View** from the menu in the upper-left corner of the Output window for the **Build** tab.

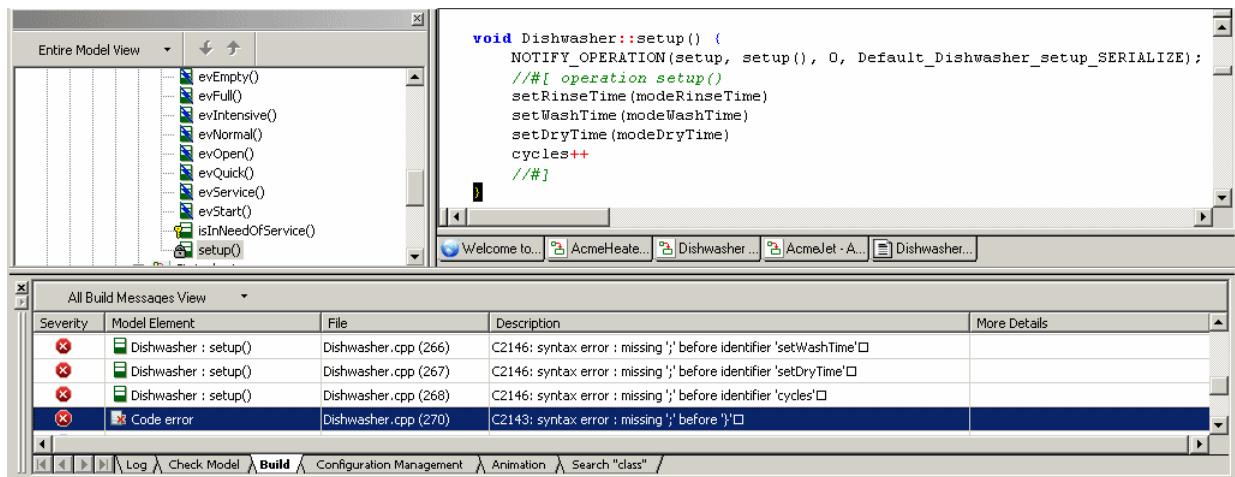
You can use the menu in the upper-left corner of the Output window for the **Build** tab to choose other views. The following figure shows the **Build** tab with **Model Element Messages** selected, which shows only items with a severity of Error or Warning that also have references to a model element. In addition, information messages, code error-type errors and warnings are filtered out.

Severity	Model Element	File	Description	More Details
Error	Dishwasher : setup()	Dishwasher.cpp (266)	C2146: syntax error : missing ';' before identifier 'setWashTime'	
Error	Dishwasher : setup()	Dishwasher.cpp (267)	C2146: syntax error : missing ';' before identifier 'setDryTime'	
Error	Dishwasher : setup()	Dishwasher.cpp (268)	C2146: syntax error : missing ';' before identifier 'cycles'	

The following figure shows the **Build** tab with **All Errors and Warnings View** selected, which shows only error and warning messages (so that information messages are filtered out).

Severity	Model Element	File	Description	More Details
Error	Dishwasher : setup()	Dishwasher.cpp (266)	C2146: syntax error : missing ';' before identifier 'setWashTime'	
Error	Dishwasher : setup()	Dishwasher.cpp (267)	C2146: syntax error : missing ';' before identifier 'setDryTime'	
Error	Dishwasher : setup()	Dishwasher.cpp (268)	C2146: syntax error : missing ';' before identifier 'cycles'	
Error	Code error	Dishwasher.cpp (270)	C2143: syntax error : missing ';' before '}'	

Note that you can double-click an item on the tab and, if possible, Rational Rhapsody opens either the relevant model element (for example, the Features window for an association that might be causing an error) or to the code source. From whichever opens, you can make corrections or view the item more closely. In the following figure, double-clicking the “Code Error” item on the **Build** tab in the lower portion of the figure, opens the code for that item in the upper-right portion of the figure.



You can right-click the **Build** tab to use the **Copy All** and **Clear All** commands.

Note

By default, the **Build** tab displays after you run a build. If you want the **Log** tab to automatically display instead after a build, you can set the `CG::General::ShowLogViewAfterBuild` property to Checked.

Supported compilers

The compilers from Microsoft, Java, and Cygwin are fully supported, which means that Rational Rhapsody is able to analyze the output from their compilers and show the correct severity levels for their messages. For all other compilers, their output will only show Informational messages.

Configuration Management tab

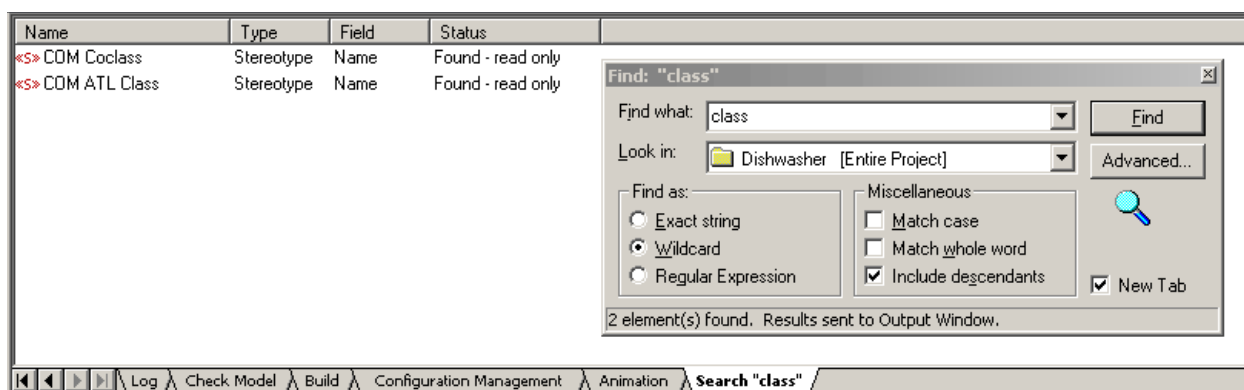
The **Configuration Management** tab shows messages related to configuration management actions for a model. You can right-click on the **Configuration Management** tab to use the **Clear**, **Copy**, **Paste**, and **Hide** commands.

Animation tab

The **Animation** tab shows messages related to animating a model. For more information about animation, see [Animation](#). You can right-click on the **Animation** tab to use the **Clear**, **Copy**, **Paste**, and **Hide** commands.

Search Results tab

The **Search Results** tab shows results from searches of your model. Note that this tab might not appear until you perform a search (for example, choose **Edit > Search** and select the **New Tab** check box). For more information about doing searches, see [Searching models](#).



Active Code View window

The Active Code View window displays code for an element selected in the browser. Whenever you make changes to the model, Rational Rhapsody regenerates the code and updates it in the Active Code View window.

To open the Active Code View window, choose **View > Active Code View**.

Specification tab

The Active Code View window has two tabs, **Specification** and **Implementation**. The **Specification** tab displays the specification code.

Implementation tab

The Active Code View window has two tabs, **Specification** and **Implementation**. The **Implementation** tab displays the implementation code.

Welcome window

Rational Rhapsody typically starts up with the Welcome window open. The Welcome window provides links to help you get started quickly. The Welcome Screen appears each time you open Rational Rhapsody unless you clear the **Show Welcome Screen at startup** check box at the bottom of the window. You can view the Welcome window at any time by choosing **Help > Welcome Screen**.

Restoring the Welcome window

To restore the display-on-startup setting for the Welcome window, do either of the following actions depending on your situation:

- ♦ If Rational Rhapsody starts up without opening the Welcome window, choose **Help > Welcome Screen**. Notice that doing this automatically selects the **Show Welcome Screen at startup** check box at the bottom of the window.
- ♦ If you have cleared the **Show Welcome Screen at startup** check box but have not yet shut down Rational Rhapsody, return to the Welcome window (select the Welcome tab in the drawing area or choose **Help > Welcome Screen**) and select the **Show Welcome Screen at startup** check box.

Rational Rhapsody project tools

The Rational Rhapsody project tools allow you to perform model design and development tasks using groups of toolbar buttons in the browser, drawing area, and output window. If a button is

disabled or not displayed, the operation represented by the button is unavailable for the currently displayed project items.

Browser filter

The browser filter lets you display only the elements relevant to your current task. Click the down arrow at the top of the browser to display the menu of filter options.

The filter is set to **Entire Model View** (default). For detailed information about the other options, see [Rational Rhapsody browser menu options](#). You can also display the browser filter using **View > Toolbars > Browser Filter**.

Standard tools

The **Standard** toolbar provides quick access to the standard tools in Rational Rhapsody. To display or hide this toolbar, choose **View > Toolbars > Standard**.

The **Standard** toolbar includes the following tools:

Tool Button	Name	Description
	New	Creates a project. This button executes the same command as File > New .
	Open	Opens an existing project. This button executes the same command as File > Open .
	Save	Saves the current project. This button executes the same command as File > Save .
	Cut	Cuts the selection to the clipboard. This button executes the same command as Edit > Cut .
	Copy	Copies the selection to the clipboard. This button executes the same command as Edit > Copy .
	Paste	Pastes the contents of the clipboard. This button executes the same command as Edit > Paste .
	Format Painter	Used for copying formatting from one element to another element in the same diagram.
	Print	Prints the active view. This button executes the same command as File > Print .
	About	Opens the About Rational Rhapsody window, which displays the product version information. You can also choose Help > About Rhapsody to open the window. In addition, when you have the About Rhapsody window open, you can click the License button to open the License Details window.
	Undo	Undoes the last operation you performed in the model. This button executes the same command as Edit > Undo .
	Redo	Reverses the undo command. This button executes the same command as Edit > Redo .
	Search	Opens the Search window for a term in the model. This button executes the same command as Edit > Search .

Tool Button	Name	Description
	References	Opens a list of elements that reference the selected element. This button executes the same command as right-clicking the selected element in the browser and selecting References .
	Locate in Browser	Locates the selected diagram element in the browser. This button executes the same command as the Locate button in the Features window and Edit > Locate in Browser .
	Delete	Deletes the current selection from model. This button executes the same command as Edit > Delete .

Edit menu commands

The Rational Rhapsody Edit menu lets you access and change text and diagrams using the following menu commands. Many of these menu commands are also represented as toolbar buttons, as described in [Standard tools](#).

Edit > Undo

Rational Rhapsody allows you to undo the last 20 operations performed on the project.

Edit > Redo

You can redo the operation that was most recently undone using **Edit > Undo** or the Undo button. **Redo** is not active until you have used **Undo** at least once.

Edit > Cut

This menu command removes the selection from the model and puts it into the Windows clipboard for possible pasting in another location.

Edit > Copy

This menu command makes a copy of the selection and puts it into the Windows clipboard for possible pasting in another location.

Edit > Paste

This menu command copies and previously cut or copied selection from the Windows clipboard into a different location.

Edit > Delete

The **Delete** menu command removes the selected item from the entire active project. In the browser, it always deletes an object from the entire model including all diagrams. Before the delete operation is completed, a confirmation message displays.

Edit > Search

Use this menu command to perform a quick search of the model or a more advanced search for text or model elements. This facility displays the results in the Output window. See [Searching models](#).

Edit > Advanced Search and Replace

Use this menu command to perform more complex searches, such as identifying only the units in the model or locating unresolved elements. See [Advanced search and replace features](#).

Edit > Search Inside Selected

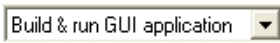
Use this menu command to perform a search within the item you selected on the Rational Rhapsody browser. See [Searching models](#).

Tools for Generating and running code

The **Code** toolbar provides quick access to frequently used Code menu commands.

The **Code** toolbar includes the following tools:

Tool Button	Name	Description
	Make	Builds the active configuration. You must generate code before you can build the configuration. This button executes the same command as Code > Generate > Build run XXX.exe .
	Stop Make/Execution	Stops the make process or the execution while it is in progress. This button executes the same command as Code > Stop .
	Run Executable	Runs the executable image. This button executes the same command as Code > Run XXX.exe .
	Generate/Make/Run (GMR)	Generates code, builds the configuration, and runs the executable image. This button executes the same command as Code > Generate/Make/Run .
	Disable/Enable Dynamic Model Code Associativity	Disables or enables dynamic model-code associativity. The button displays as two connected arrows when DMCA is active, and two disconnected arrows when DMCA is inactive. See Deleting Redundant Code Files .

Tool Button	Name	Description
	Current Component	Contains a list of all components in the project. To change the active component, select it from the drop-down list.
	Current Configuration	Contains a list of all configurations in the active component. To change the active configuration, select it from the drop-down list.

Tools for managing and arranging windows

The **Windows** toolbar provides quick access to Rational Rhapsody windows, such as the browser and the Features window. You can also access these commands from the View menu. To display or hide this toolbar, choose **View > Toolbars > Windows**.

The **Windows** toolbar includes the following tools:

Tool Button	Name	Description
	Browser	Toggles between showing and hiding the browser.
	Show/Hide Features	Toggles between showing and hiding the Features window for the current element.
	Show/Hide Active Code View	Toggles between showing and hiding the Active Code View window.
	Show/Hide Output Window	Toggles between showing and hiding the Output window.
	Toggle Arrange Options	Toggles between two standard desktop arrangements. Alternatively, choose Window > Arrange Options . Use Windows > Arrange Icons to manipulate the arrangement of the desktop.
	Bird's Eye Window	Toggles between showing and hiding the Bird's Eye window. You can also press Alt+F5 to perform the same operation. For more information about this feature, see The Bird's Eye (diagram navigator) .
	Back	Displays the previously displayed window. This operation is also available using Window > Back .
	Forward	Displays the window in the opposite direction from Back . This operation is also available using Window > Forward .

Note

The Back and Forward navigation is unavailable on Linux. This is for Windows systems only.

Tools for the Favorites browser

The **Favorites** toolbar provides tools for the Favorites browser. To display or hide this toolbar, choose **View > Toolbars > Favorites**.

For more information about the Favorites browser and about the **Favorites** toolbar, see [The Favorites browser](#).

Tools for the VBA interface options

The **VBA** toolbar provides quick access to Rational Rhapsody VBA Interface options. To display or hide this toolbar, choose **View > Toolbars > VBA**.

The **VBA** toolbar includes the following tools:

Tool Button	Name	Description
	VBA Editor	Opens the VBA editor.
	Show Properties	Opens the VBA properties window.
	Macros	Opens the Macros window so you can create VBA macros.
	Design Mode	Runs VBA in design mode.

Tools for animation

When you run an executable model with instrumentation set to **Animation**, Rational Rhapsody displays the **Animation** toolbar. This toolbar automatically appears during an animation session. To display or hide this toolbar during an animation session, choose **View > Toolbars > Animation**.

For more information about animation and about this toolbar, see [Animation](#).

Tools for creating and editing diagram elements

The **Diagram Tools** panel provides access to tools used in creating and editing diagrams in the graphic editors. Each graphic editor has a unique set of diagram tools. To display or hide the drawing tools for the current diagram, choose **View > Toolbars > Drawing**.

For more information about modeling toolbars, see [Graphic editors](#).

Tools for common annotations

Use the **Common** tools, which are displayed in its own section on the **Diagram Tools** panel, to add annotations (constraints, comments, and requirements) to a diagram. To display or hide the **Diagram Tools** panel, choose **View > Toolbars > Drawing**.

For information about annotations for diagrams, see [Annotations for diagrams](#).

The **Common** toolbar includes the following tools:

Tool Button	Name	Description
	Note	Creates a documentation note. Click the button and use the mouse to draw the note in the diagram. This is the type of note available with previous versions of Rational Rhapsody. The note displays in the diagram, but not in the browser.
	Constraint	Creates a constraint. This button executes the same command as Edit > Add New > Constraint .
	Comment	Creates a comment. This button executes the same command as Edit > Add New > Comment .
	Requirement	Creates a requirement. This button executes the same command as Edit > Add New > Requirement .
	Anchor Constraint/ Comment/ Requirement to Item	Creates an anchor for a constraint, comment, or requirement.

Tools for zooming diagram views

The **Zoom** toolbar contains the zoom tools you can use with all the different diagram types. These tools are also available in **View > Zoom/Pan**. To display or hide this toolbar, choose **View > Toolbars > Zoom**.

For more information about zoom function and the Zoom toolbar, see [Zoom](#).

Tools for formatting text

The **Format** toolbar provides tools that affect the display of text in your diagrams, such as font, size, color, and so on. In addition, you can access these options by selecting **Edit > Format > Change**. To display or hide this toolbar, choose **View > Toolbars > Format**.

For more information about the **Format** toolbar, see [Format text on diagrams](#).

Tools for the layout of elements

The **Layout** toolbar provides quick access to tools that help you with the layout of elements in your diagram, including a grid, page breaks, rulers, and so on. To display or hide this toolbar, choose **View > Toolbars > Layout**.

For more information about the **Layout** toolbar, see [Layout toolbar](#).

Tools for free shapes

Use the **Free Shapes** tool, which are displayed in its own section on the **Diagram Tools** panel, to draw elements freehand in a diagram. To display or hide the **Diagram Tools** panel, choose **View > Toolbars > Drawing**.

For more information about the graphic editors and about these tools, see [Graphic editors](#).

Creating diagrams

The **Diagrams** drawing buttons access the graphic editor to create and edit diagrams. The [Profiles](#) create a starting point structure for new projects and control which diagrams are available for those projects. The available diagrams are represented as buttons on the toolbar across the top of the Rational Rhapsody window.

To create a new diagram:

1. Choose **Tools > Diagrams** and select the type of diagram you want to create, or click the diagram button at the top of the Rational Rhapsody window.
2. The Open window for the selected diagram displays. Highlight the portion of the project with which the diagram will be associated.
3. Click **New**.
4. In the New Diagram window, enter the **Name** of the new diagram.
5. If you want to populate the new diagram automatically with existing model elements, click the **Populate Diagram** check box.
6. Click **OK**.

Tools for creating/opening diagrams

The **Diagrams** toolbar includes the following tools:

Diagram Button	Name	Description
	Object Model Diagram	This diagram shows the logical views of the static structure of the classes and objects in an object-oriented software system and the relationships between them. This diagram is available for the majority of profiles. Click this button to be able to open an existing object model diagram or to create one.
	Sequence Diagram	This diagram shows the interactions between objects in the form of messages passed between the objects over time. This diagram is available for the majority of profiles. Click this button to be able to open an existing sequence diagram or to create one.
	Use Case Diagram	This diagram shows the use cases of the system and the actors that interact with them. This diagram is available for the majority of profiles and also the FunctionalC profile. Click this button to be able to open an existing use case diagram or to create one.
	Component Diagram	This diagram shows the dependencies among software components, such as library or executable components. This diagram is available for the majority of profiles. Click this button to be able to open an existing component diagram or to create one.
	Deployment Diagram	This diagram shows the run-time physical architecture of the system. This diagram is available for the majority of profiles. Click this button to be able to open an existing deployment diagram or to create one.
	Collaboration Diagram	This diagram describes how different kinds of objects and associations are used to accomplish a particular task. This diagram is available for the majority of profiles. Click this button to be able to open an existing collaboration diagram or to create one.
	Structure Diagram	This diagram shows the architecture of the composite classes that define the model structure. This diagram is available for the majority of profiles. Click this button to be able to open an existing structure diagram or to create one.
	Open Statechart	A statechart defines the behaviors of individual classes in the system. This diagram is available for the majority of profiles and also the FunctionalC profile. Click this button to be able to create one a statechart.
	Open Activity Diagram	This diagram shows the lifetime behavior of an object, or the procedure that is executed by an operation in terms of a process flow, rather than as a set of reactions to incoming events. This diagram is available for the majority of profiles. Click this button to be able to create an activity diagram.
	Panel Diagram	This diagram provides you with a convenient way to demonstrate a user device. During animation or Webify, you can use a panel diagram to activate and monitor your user application. This diagram is available for Rational Rhapsody in C, Rational Rhapsody in C++, and Rational Rhapsody in Java projects. Click this button to be able to open an existing panel diagram or to create one.
	Build Diagram	This diagram shows how the software is to be built. This diagram is primarily associated with the FunctionalC profile. Click this button to be able to open an existing build diagram or to create one.

Diagram Button	Name	Description
	Call Graph	A call graph shows the relationship of function calls as well as the relationship of data. This diagram is primarily associated with the FunctionalC profile. Click this button to be able to open an existing call graph or to create one.
	File Diagram	This diagram shows how files interact with one another (typically how the <code>#include</code> structure is created). This diagram is primarily associated with the FunctionalC profile. Click this button to be able to open an existing file diagram or to create one.
	Message Diagram	This diagram shows how the files functionality might interact through messaging (synchronous function calls or asynchronous communication). This diagram is primarily associated with the FunctionalC profile. Click this button to be able to open an existing message diagram or to create one.
	Open Flowchart	For a function or class the chart, a flow chart shows the operational flow and code generation. This diagram is primarily associated with the FunctionalC profile. Click this button to be able to open an existing flow chart.

Opening the main diagram

To open the diagram for an element in the browser, select the element and choose **Tools > Main Diagram**. This is commonly used to see the diagrams associated with classes. If no diagram is identified as the main diagram for the selected class, no diagram is displayed.

Locating in the browser

If you want to locate an element from a diagram in the Rational Rhapsody browser, select the diagram element and choose **Edit > Locate in Browser**, click the Locate in Browser button, or press **Ctrl+L**. The browser opens and highlights the intended element.

Add new elements

To create elements for your project, use either the **Edit > Add New** submenu commands or right-click a browser item and select the intended addition from the **Add New** submenu. The options displayed on the submenu depend on the selected item in the browser or diagram. After you create the element, you might need to define it further using the [The Features window](#).

Add New > Event

This menu command creates an event, which is a specification of a significant occurrence that has a location in time and space.

Add New > Interface

This menu command creates a set of operations that publicly define a behavior or way of handling something so knowledge of the internals is not needed or interface. Component diagrams define interfaces between components only.

Add New > Actor

This menu command creates an actor element for a use case diagram.

Add New > Tag

Tags add information to the model relating to the domain or platform. To create a tag using the browser, select the element and choose **Edit > Add New > Tag**.

Add New > Use Case

Use cases represent the externally visible behaviors or functional aspects of the system, but the content of use cases is not used for code generation. To create a use case in the browser, select a package or other element that might contain a use case and choose **Edit > Add New > Use Case**.

Add New > Requirement

To add an intended feature, property, or behavior of a system component as a requirement, right-click the component in the browser and select **Add New > Requirement**.

Add New > Flow Item

To describe the kinds of information that can be exchanged between objects, add a flow item to represent either pure data, data instantiation, or commands (events). Flow items can represent classes, types, events, relations, parts, objects, attributes or variables. To add the flow item, select the element being represented and choose **Edit > Add New > Flow Item**.

The Features window

The Features window enables you to edit the features of each element in the Rational Rhapsody model. The Features window contains a number of tabs that are common to almost all types of elements:

- ◆ [General tab](#)
- ◆ **Description** (see [Creating hyperlinks on the Description tab](#))
- ◆ **Relations** (see [Define relations](#))
- ◆ **Tags** (see [Use tags to add element information](#))
- ◆ [Displaying a tab on the Features window in a stand-alone window](#)

Open the Features window

To define and change model elements, use any of these methods to launch the Features window for the element that needs to be modified:

- ◆ Double-click an element in the browser (except a diagram).
- ◆ Double-click an element on a diagram.
- ◆ Right-click an element and select **Features**.
- ◆ Select an element in the browser and press **Alt + Enter**. This is unavailable for a Rational Rhapsody project within the Eclipse platform.
- ◆ Select an element and choose **View > Features**. This is unavailable for a Rational Rhapsody project within the Eclipse platform.

The Features window lists different fields depending on the element type.

Applying changes with the Features window

When you have made changes that need to be applied to the project, an asterisk (*) is displayed in the title bar of the Features window. Use one of these methods to save the changes.

- ◆ Press **Ctrl + Enter**.
- ◆ Click the **Apply** button on the Features window.
- ◆ Change focus to another window or tab.
- ◆ Initiate an external activity, such as generating code, saving the project, or generating a report.

To apply changes and close the Features window, click **OK**.

Canceling changes on the Features window

To cancel changes made to the Features window, press the **Esc** key. Alternatively, you can close the window without applying changes.

Note that changes cannot be canceled once they have been applied to the model.

General tab

The **General** tab of the Features window enables you to define the characteristics of the selected element. The fields for the General tab vary depending on the characteristics of the selected element. The most common fields on the **General** tab are as follows:

- ♦ In the **Name** box you specify the name of the element.
- ♦ You use the **L** button to open the Name and Label window to specify the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ In the **Stereotype** list you specify the stereotype of the element, if any. For information on creating stereotypes, see [Stereotypes](#).
 - To select from a list of current stereotypes in the project, click the Select Stereotype button .
 - To sort the order of the selected stereotypes, click the Change Stereotype Order button .
- ♦ **Default Package** specifies a group for the selected element.

Properties tab

The **Properties** tab of the Features window displays the properties for the currently selected item (selected in the browser or in a diagram) or all of the properties for a model (choose **File > Project Properties**).

For more information about the Properties tab, see [Rational Rhapsody properties](#).

Pinning the Features window

The Features window can be “pinned” to a specific element to keep the information for that element displayed while examining other features for your element.

To pin a Features window to an element:

1. Right-click an element in the browser or a diagram and select **Features** to open the Features window for that element.
2. Click the Pin button  (horizontal orientation) in the upper right corner of the window.
Note that the button changes to a vertical orientation  to indicate that the window for this element will now remain displayed.

In pinned mode, the features displayed in the pinned window remain displayed and accessible from all of the window tabs even when a different element is selected. Therefore, you can display and pin two or three Features windows to compare the information for the elements.

When you no longer need to see the features of the element displayed, you can click the Pin button again to disconnect it from the element or simply close the window.

Hiding the buttons on the Features window

At the bottom of the Features window, there are three buttons: **Locate**, **OK**, and **Apply**.

To remove these buttons from view, right-click the title bar for the Features window and uncheck **Features Toolbar**.

Docking the Features window

By default, the Features window is a floating window. It can be positioned anywhere on the window, or docked to the Rational Rhapsody work area.

To dock the Features window in the Rational Rhapsody window, do one of the following actions:

- ♦ Double-click the title bar of the Features window. The window will jump to the location where it was last docked. To dock the window in a different location, click the title bar and drag the window to the intended location.
- ♦ Right-click the title bar and select **Enable Docking by Drag** to display a check mark and drag the window to the intended location.

Undocking the Features window

To undock the Features window, do one of the following actions:

- ◆ Double-click the title bar of the Features window.
- ◆ Click the title bar and hold down **Ctrl** while dragging to a new location.
- ◆ Right-click the title bar and select **Enable Docking by Drag** to remove the check mark and drag the window to the intended location. The window is no longer docked with the main window.

Opening multiple instances of the Features window

You can open multiple Features windows in the Rational Rhapsody workspace. Using this functionality, you can easily compare the features of two different elements and quickly copy text from one Features window to another.

To open more than one Features window, right-click an element and select **Features in New Window**.

When opened as a new window, the Features window remains focused on the same element, even when you change the browser or graphic editor selection. Any changes made to that element from another view (such as the browser or a diagram editor) are automatically updated in the Features window. This enables you to keep track of the features for a particular element while working with other parts of the model.

When you have an open Features window that is focused on a particular element, you can locate that element in the browser by clicking the **Locate** button at the bottom of the window. Alternatively, you can locate the item by selecting the **Locate in Browser** tool from the standard toolbar.

Displaying a tab on the Features window in a stand-alone window

For each of the Features tabs, you have the option to display the information in a dockable stand-alone window.

To do display a tab on the Features window in a stand-alone window:

1. Select an element in the Rational Rhapsody browser or in a diagram.
2. Choose the relevant menu item in the View menu, for example, **View > Description**.

Docking a stand-alone window for a Features window tab

Once you have a stand-alone window open for a tab of the Features window (as well as the Features window), you can dock it.

To dock a the Features window or a stand-alone window for one of its tabs:

1. Right-click the title bar for the window and select **Enable Docking by Drag**. Notice that a check mark displays to the left of the command on the pop-up menu.
2. Drag the window to one of the borders or other docking locations in the Rational Rhapsody window. Notice that upon reaching one of these locations, the outline of the window changes to reflect the area the window occupies when docked.

Note

When one of these windows is docked, it continues to display the information in the same manner as it does when it is “pinned,” as described in [Pinning the Features window](#).

Undocking a stand-alone window for a Features window tab

To undock a the Features window or a stand-alone window for one of its tabs, do one of the following actions:

- ♦ To undock without disabling the docking capability, drag the window to one of the non-docking locations in the Rational Rhapsody window.
- ♦ To disable docking and undock:
 - a. Right-click the title bar for the window and select **Enable Docking by Drag**. Notice that the check mark to the left of the command no longer displays.
 - b. Drag the window anywhere in the Rational Rhapsody window.

Hiding tabs on the Features window

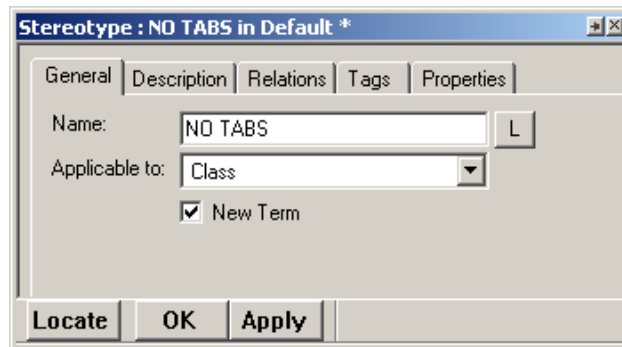
If you normally do not use one of the tabs on the Features window for a particular metaclass, you can hide it. To do this, you have to create a **New Term** stereotype that sets the `HideTabsInFeaturesDialog` property to hide one or more tabs on the Features window. Then you would apply this stereotype to a model element of that metaclass.

Note

This feature is used exclusively for elements with the **New Term** stereotype.

To hide one or more tabs on the Features window for a particular metaclass:

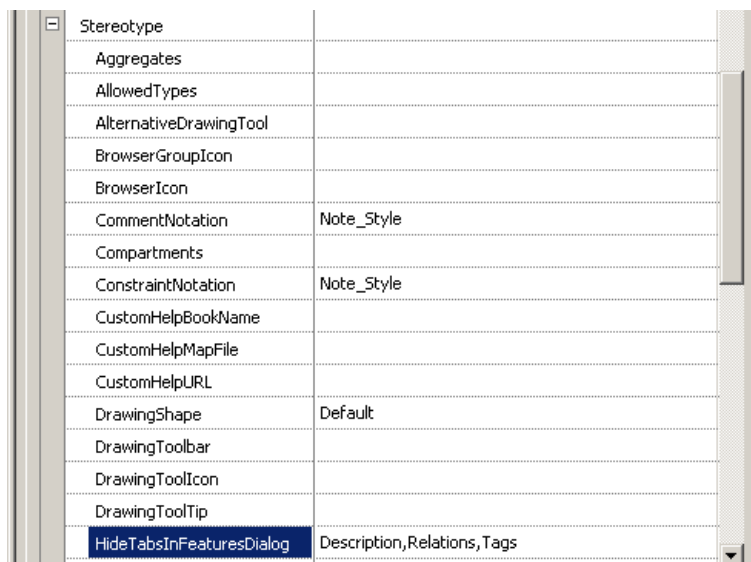
1. Display your model in Rational Rhapsody.
2. In the Rational Rhapsody browser, create a stereotype as a **New Term**.
 - a. To learn how to create a stereotype, see [Stereotypes](#).
 - b. Be sure to select one metaclass in the **Applicable To** box and select the **New Term** check box on the **General** tab of the Features window for your stereotype.



- c. On the **Properties** tab for the stereotype, locate the `Model::Stereotype::HideTabsInFeaturesDialog` property.

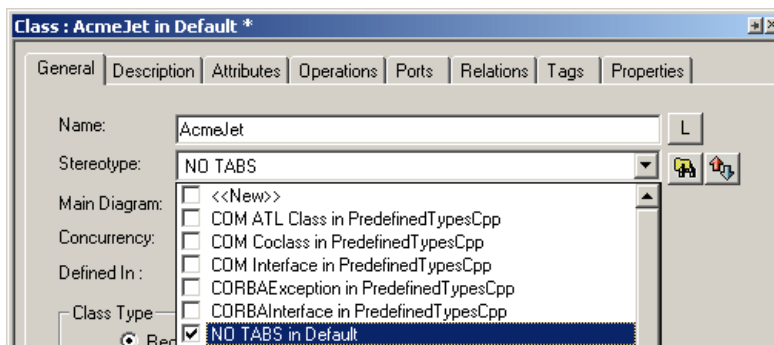
- d. Click the box to the right of the property name and type the names of the tabs, separated by a comma, that you want to hide; for example, Description, Relations, Tags.

Note: You cannot hide the **General** tab.



- e. Click **OK**.

3. Apply the stereotype to a model element of the metaclass. For example, if you selected the **Class** metaclass (in the **Applicable To** box on the **General** tab of the Features window for the stereotype), then you can apply this stereotype (select it from the **Stereotype** drop-down menu on the **General** tab of the Features window for the class) to any classes you currently have in your model or any that you create.



Note: When you define a stereotype as a **New Term**, it is given its own category in the Rational Rhapsody browser, and any elements to which this stereotype is applied are displayed under this category.

If you want to display a previously hidden tab, delete the name of that tab from the list you entered in the `Model::Stereotype::HideTabsInFeaturesDialog` property.

Hyperlinks

Rational Rhapsody supports both *internal hyperlinks*, which point to Rhapsody model elements, and *external hyperlinks*, which point to a URL or file.

In addition, you can:

- ◆ Use the DiffMerge tool to compare models to locate differences in diagrams and to merge models that contain hyperlinks.

Note: You can edit a description that uses hyperlinks or RTF format in the DiffMerge tool if it is from the left or right side of the comparison, but you cannot edit a description from a merge.

- ◆ Export hyperlinks using the Rhapsody COM API.

Note: You cannot create or modify hyperlinks using the COM API.
- ◆ Report on hyperlinks using ReporterPLUS.
- ◆ Find references to hyperlinks using the Show References feature.

Create hyperlinks

You can create hyperlinks inside the description of an element, or with the Rational Rhapsody browser.

A typical use for the **Description** tab of the Features window is to enter a description for whatever Rhapsody element you currently have open. For example, if you have the Features window open for a class, you can enter a detailed description for the class on the **Description** tab. You can do the same on the **Description** tab for an attribute, an event, a package, and so on.

Note

Hyperlinks created in the **Description** are not model elements and can neither be viewed in the browser nor accessed by the API.

Creating hyperlinks on the Description tab

To create hyperlinks on the Description tab:

1. Open the Features window for the element.
2. On the **Description** tab, right-click in the open field and select **Hyperlink**.

Note: If you want to replace pre-existing text with a hyperlink, select the text before right-clicking.

3. On the Hyperlink window, specify the hyperlink text and target.
 - ♦ The **Text to display** group specifies the text for the hyperlink. The possible values are as follows:
 - **Free text** displays the specified text as the hyperlink text.
 - **Target name** displays the full path of the target as the hyperlink text.
 - **Target label** displays the label of the target as the hyperlink text. This option is available only for internal hyperlinks that have labels.
 - **Tag value** displays the value for the tag. Note that this value is available only when you select a tag as the hyperlink target. For an example that uses this field, see [Using tag values in hyperlinks](#).
 - ♦ The **Link target** group specifies the target file, Web page, or model element. You can specify the target by typing the target in the text field, using the list to select the model element in the model, or clicking the Ellipses button  to open a new window so you can navigate to the target file.

Note: You can include a relative path in the hyperlink target. If you use a relative path, the base directory is the one where the <Project name>.rpy file is located.

- Click **OK**.
The hyperlink is displayed on the **Description** tab as blue, underlined text. This type of hyperlink is not displayed in the browser.

Creating hyperlinks on the Rational Rhapsody browser

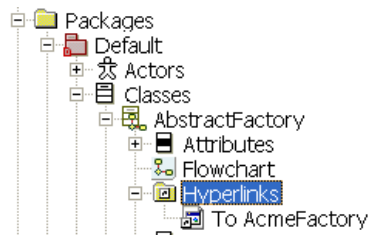
To create a hyperlink in the browser:

- Right-click the element to which you want to add the hyperlink and select **Add New > Relations > Hyperlink**.
Rhapsody creates a hyperlink in the browser.

Note: **Add New > Relations** is the default menu command structure in Rhapsody. It can be changed by users. This topic assumes that all defaults are in place.

- Open the Features window for the new hyperlink.
- Specify the hyperlink display text in the **Text to display** group.
- Specify the hyperlink target in the **Link target** group by typing the path, using the drop-down list, or using the navigation window.
- Optionally, specify a stereotype or description.
- Click **OK**.

The hyperlink is added to the Hyperlinks category under the owner element.



To improve readability, there are different icons for the different targets, such as the following targets:

- ◆ Word files
- ◆ Classes
- ◆ URLs

You can drag-and-drop hyperlinks from the **Hyperlinks** category of one element to that of another. Similarly, you can copy hyperlinks from the Hyperlinks category of one element to that of another by dragging-and-dropping and pressing **Ctrl**, or using the Copy and Paste shortcuts.

Following a hyperlink

To follow a hyperlink, double left-click it. The corresponding file, window, or URL is displayed.

Alternatively, you can use the **Open Hyperlink** option in the menu.

Edit a hyperlink

You can edit a hyperlink using the Features window or from within the Description area, depending on the type of hyperlink.

Note

You cannot rename a hyperlink directly from the browser. You must open the Features window.

Use the Features window to change the features of the hyperlink, including its text display and target.

A hyperlink has the following features:

- ♦ **Name** specifies the name of the element. The default name is `hyper link_n`, where `n` is an incremental integer starting with 0.
- ♦ **Text to display** specifies the text for the hyperlink. The possible values are **Free text**, **Target name**, **Target label**, and **Tag value**. For more information on these options, see [Creating hyperlinks on the Description tab](#).
- ♦ **Link target** specifies the target file, Web page, or model element.
- ♦ **Description** describes the hyperlink.

Editing the hyperlink in the Description area

To edit a hyperlink in the Description area:

1. Open the Features window for the element.
2. On the **Description** tab, right-click the hyperlink in the text and select **Edit Hyperlink**.
3. In the Hyperlink window (see [Creating hyperlinks on the Description tab](#)), edit the link.
4. Click **OK**.

Using tag values in hyperlinks

You can display the value of a tag in a hyperlink.

To add a tag value to a hyperlink:

1. Wherever you want to create the hyperlink, right-click and select **Features** to open the Features window.
2. On the **Tags** tab, use the **Quick Add** group to enter the name of the hyperlink and its value. If the tag does not have a value, the value «empty» is displayed.
3. Click **OK**.

Changing the tag value

To change the value of the tag:

1. Click the tag value hyperlink or click the New button  on the **Tags** tab to open this Features window.
2. Replace the existing value with the new value.
3. Click **OK**.

Deleting a hyperlink

Delete a hyperlink using one of the following methods:

- ♦ In the text area of the **Description** tab, right-click the link and select **Remove Hyperlink**, or use the backspace key or **Delete** icon.
- ♦ In the browser, select the hyperlink and select **Delete from Model** or click the **Delete** icon.

Hyperlink limitations

Note the following limitations:

- ♦ You can select tags as hyperlink targets, which are available in the Rhapsody browser. For example, if you have the tag `color` in a profile that is applicable to all classes, you cannot see the tag `color` under a given class instance in the browser. The Rhapsody browser shows only local or overridden tags; however, these tags are shown in the **Tags** tab of the Features window for the class.
- ♦ If you override a tag value in a package, the tag is considered to be local because it is tied to that specific element. If you have a hyperlink to the local tag and subsequently delete the tag, the reference will be unresolved.

Create a diagram

This topic is for Eclipse users.

Besides being able to import a Rational Rhapsody project and all its diagrams, you can create diagrams in the Rational Rhapsody Platform Integration.

Creating a diagram

This procedure is for Eclipse users.

To create a Rational Rhapsody diagram in Eclipse:

1. On the New Diagram window, select a diagram type from the drop-down list.
2. Enter a name for the diagram.
3. If available, select a location for the diagram from the drop-down list.
4. If available, if you want to populate the new diagram automatically with existing model elements. Click the **Populate Diagram** check box.
5. Click **Finish**.

Create a Rational Rhapsody project

Besides being able to import a Rational Rhapsody project in the Rational Rhapsody Platform Integration, you can create a Rational Rhapsody project.

Creating a Rational Rhapsody project

This procedure is for Eclipse users.

To create a Rational Rhapsody project:

1. On the New Rhapsody Project window, enter a name for your Rational Rhapsody project.
2. Optionally, enter a name for your first object model diagram.
3. Select the language for your project from the drop-down list.
4. If available, select the Rational Rhapsody project type from the drop-down list.
5. If you want to designate a location for your project other than your default location, clear the Use default location check box and browse to your preferred location.

6. Click **Finish**.
7. If the directory for your project is not already created, click **Yes** when you are asked if you want to create it.

Import a Rational Rhapsody project

Besides being able to create a Rational Rhapsody project in the Rational Rhapsody Platform Integration, you can import an existing Rational Rhapsody project.

Importing a Rational Rhapsody project

This procedure is for Eclipse users.

To import a project:

1. On the Import window, browse to your select root directory.
2. Select the projects you want to import.
 - ♦ Click **Select All** to select all the projects listed.
 - ♦ Click **Deselect All** to clear the check boxes for all the selected projects.
 - ♦ Click **Refresh** to refresh your list.
3. If available, select your options selection:
 - ♦ **With All Subunits**. Select this radio button to load all units in the project, ignoring workspace information. For information on workspaces, see [Using workspaces](#).
 - ♦ **Without Subunits**. Select this radio button to prevent loading any project units. All project units will be loaded as stubs.
 - ♦ **Restore Last Session**. Select this radio button if you would like to load only those units that were open during your last Rational Rhapsody session.
4. Click **Finish**.

Import source code

Importing source code involves the Rational Rhapsody Reverse Engineering tool.

Importing source code

This procedure is for Eclipse users.

To import source code:

1. On the Importing Source Code window, click the **Finish** button.
2. On the message box that displays click **Continue** to open the Reverse Engineering window.
3. See [Reverse engineering legacy code](#).

Search window

This topic is for Eclipse users.

The **Search** window shows results from searches of your model. Note that this window might not appear until you perform a search (for example, choose **Search > Search** and select the **Rhapsody** tab). For more information about doing searches in Rational Rhapsody, see [Searching models](#).

Graphic editors

This topic is for Eclipse users.

You can use the graphic editors to analyze, design, and construct the system using UML diagrams. Diagrams enable you to observe the model from several different perspectives, like turning a cube in your hand to view its different sides. Depending on its focus, a diagram might show only a subset of the total number of classes, objects, relationships, or behaviors in the model. Together, the diagrams represent a complete design.

Rational Rhapsody adds the objects created in diagrams to the Rational Rhapsody project, if they do not already exist. Conversely, Rational Rhapsody removes elements from the project when they are deleted from a diagram. However, you can also add existing elements to diagrams that do not need to be added to the project, and remove elements from a diagram without deleting them from the model repository.

For more information about the graphic editors, see [Graphic editors](#).

Call stack and event queue

This topic is for Eclipse users.

The call stack view describes the current stack of calls for the focus thread. The event queue view describes the current state of the event queue for the focus thread.

Classes and types

Classes provide a specification (blueprint) for *objects*, which are self-contained, uniquely identified, run-time entities that consist of both data and operations that manipulate this data. Classes can contain attributes, operations, events, relations, components, super classes, types, actors, use cases, diagrams, and other classes. The Rational Rhapsody browser icon for a class is a three-compartment box with the top, or name, compartment filled in. For an example of this icon, see [Defining the attributes of a class](#).

Creating a class

To create a class, in the Rational Rhapsody browser:

- ◆ Right-click the **Classes** category to which you want to add a class and select **Add New Class**.
- ◆ Right-click a package and select **Add New > Class**.
- ◆ Select a package and choose **Edit > Add New > Class**.

Rational Rhapsody creates a new class and names it `class_n`, where `n` is greater than or equal to 0. The new class is located in the browser under the **Classes** category, and is selected so that you can rename it.

For information on creating classes in OMDs, see [Object model diagrams](#).

Class features

Use the Features window to define and modify a class. You can also use it to re-arrange the order of attributes and operations, control the display of attributes and operations, create templates, and so on. To open the Features window for a class, double-click it on the Rational Rhapsody browser, or right-click it and select **Features**.

Defining the characteristics of a class

Use the **General** tab of the Features window to define the characteristics of a class.

On the **General** tab, you define the general features for a class through the various controls on the tab.

- ♦ In the **Name** box you specify the name of the element. The default name is `class_n`, where `n` is an incremental integer starting with 0. To enter a detailed description of the class, use the **Description** tab.
- ♦ You use the **L** button to open the Name and Label window to specify the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ In the **Stereotype** list you select the stereotype of the element, if any. For information on creating stereotypes, see [Stereotypes](#).
 - To select from a list of current stereotypes in the project, click the Select Stereotype button .
 - To sort the order of the selected stereotypes, click the Change Stereotype Order button .

Note: The COM stereotypes are constructive; that is, they affect code generation.

- ♦ In the **Main Diagram** list you specify the diagram (from the ones available) that contains the most complete view of the class.
- ♦ In the **Concurrency** drop-down list box you specify the concurrency. The possible values are as follows:
 - **Active** means the class runs on its own thread.
 - **Sequential** means the class runs on the system thread.
- ♦ In the **Defined In** drop-down list box you specify the owner for the class. Every class lives inside either a package or another class.

Note: A class not explicitly drawn in a package belongs to the default package of the diagram. If the diagram is not explicitly assigned to a package, the diagram belongs to the default package of the project.

- ♦ In the **Class Type** area you specify the class type. The possible values are as follows:
 - **Regular** creates a class.
 - **Template** creates a template. To specify the necessary arguments, use the **Template Parameters** tab that displays once you select the **Template** radio button. For more information, see [Creating a template class](#).
 - **Instantiation** creates an instantiation of a template. To specify the necessary arguments, use the **Template Instantiation** tab that displays once you select the **Instantiation** radio button. For more information, see [Instantiating a template class](#).

Note: To create an instance of a class, select the **Instantiation** radio button and select the template that the instance is from. For example, if you have a template class A and create B as an instance of that class, this means that B is created as an instance of class A at run time.

Selecting nested classes in windows

Every primary model element is uniquely identified by a path in the following form:

`<ns1>::<ns2>::...<nsn>::<name>`

In this syntax, ns can be either a package or a class. Primary model elements are packages, classes, types, and diagrams. Classes can contain only other classes, stereotyped classes (such as actors), and types.

You can select a nested element in a window by entering its name in either of the following formats:

- ◆ `<name> in <ns1>::<ns2>::...<nsn>`
- ◆ `<ns1>::<ns2>::...<nsn>::<name>`

Defining the attributes of a class

Attributes are the data members of a class. Rational Rhapsody automatically generates accessor (get) and mutator (set) methods for attributes, so you do not need to define them yourself.

The Rational Rhapsody browser icon for attributes is a three-compartment class box with the middle compartment filled in:



The icon for the **Attributes** category is black.



The icon for an individual attribute is red.



The icon for a protected attribute is overlaid with a key.



The icon for a private attribute is overlaid with a padlock.

The **Attributes** tab of the Features window contains a list of the attributes that belong to the class. It allows you to perform the following tasks:

- ◆ Add a new attribute.

To create a new attribute, either click the <New> row in the list of attributes, or click the New button  in the upper, right corner of the window. The new row is filled in with the default values.

- ◆ **Modify an existing attribute.**

To modify an attribute, you can use any of the following methods:

- Select the attribute and change the value name and/or change its parameters from the drop-down list boxes.
- Select the attribute and click the open Feature Dialog button  to open the Features window for the attribute and make your changes there. You can also double-click the attribute name or icon next to the name to open the Features window.

- ◆ **Delete an attribute.**

To delete an attribute from the model, select the attribute and click the Delete button .

- ◆ **View the attribute values.**

To view the values for an attribute, open the Features window for it.

You can use the following keyboard shortcuts within an editable list:

- ◆ **Arrow** keys to move between rows and columns.
- ◆ **Enter** key to start or stop editing in a text box, or to make a selection in a combo box.
- ◆ **Insert** key to insert a new element below the selected element.
- ◆ **Delete** key to delete the selected element.
- ◆ **Esc** key to cancel editing.

Defining the features of an attribute

When you click the Invoke Features window button  or double-click an attribute, the Attribute window opens. This window is also displayed when you select an attribute in the browser and might have different options than shown in here.

On the **General** tab, you define the general features for a attribute through the various controls on the tab.

- ♦ In the **Name** box you specify the name of the attribute. The default name is `attribute_n`, where `n` is an incremental integer starting with 0. To enter a detailed description of the attribute, use the **Description** tab.
- ♦ You use the **L** button to open the Name and Label window to specify the label for the element, if any. For information on creating tables, see [Descriptive labels for elements](#).
- ♦ In the **Stereotype** list you specify the stereotype of the attribute, if any. For information on creating stereotypes, see [Stereotypes](#) for information on creating stereotypes.
 - To select from a list of current stereotypes in the project, click the Select Stereotype button .
 - To sort the order of the selected stereotypes, click the Change Stereotype Order button .

Note: The COM stereotypes are constructive; that is, they affect code generation.

- ♦ In the **Attribute type** area you specify the attribute type. There are two ways to specify the type:
 - Select the **Use existing type** check box to select a predefined or user-defined type or class. Use the **Type** list to select from among the Rational Rhapsody predefined types, and any types and classes you have created in this project. Or to define a new type, delete the value in the **Type** drop-down list box to enable the Invoke Features window button and click it to open the Type window.

For more information on creating types, see [Composite types](#).
 - Clear the **Use existing type** check box if there is no defined type. A **C++[Java] Declaration** box displays in which you can give the attribute a declaration appropriate for your language edition. See [Modifying data types](#).
- ♦ In the **Visibility** area you specify the type of access (visibility) for the accessor/mutator generated for the attribute: **Public**, **Protected**, or **Private**.

When you generate code, each attribute is generated into three entities:

- The data member itself
- An accessor (get) method for retrieving the data value
- A mutator (set) method for setting the data value

Note: The visibility setting affects only the visibility of the accessor and mutator methods, not of the data member itself. The data member is always protected, regardless of the access setting.

- ♦ In the **Multiplicity** box (displayed when appropriate) you specify the multiplicity of the attribute. If this is greater than 1, use the **Ordered** check box to specify whether the order of the reference type items is significant. The modifier choices are as follows:
 - **Constant** specifies whether the attribute is read-only (check box is selected) or modifiable (check box is cleared).
 - **Reference** specifies whether the attribute is referenced as a reference, such as a pointer (*) or an address (&) in C++.
 - **Static** creates a static attribute, which belongs to the class as a whole rather than to individual objects. See [Initializing static attributes](#).
- ♦ In the **Initial Value** box you specify the initial value for the attribute.

Launching a text editor

To access the text editor, click the Ellipses button . Throughout the Rational Rhapsody interface, the Ellipses button opens a text editor.

Modifying data types

To create or edit a user-defined data type:

1. Open the Features window for the attribute.
2. In the **Attribute type** area, clear the **Use existing type** check box.
3. Type a declaration for the new type in the **C++[Java] Declaration** box using the proper syntax. Note the following information:
 - ♦ You can omit the semicolon at the end of the line; Rational Rhapsody automatically adds one if it is not present.
 - ♦ Substitute %s for the name of the type in the declaration. For example:

```
typedef unsigned char %s[100]
```

This translates to the following declaration in the generated code:

```
typedef unsigned char msg_t[100];
```
4. Add a description for the type on the **Description** tab.
5. Click **OK**.

Rational Rhapsody adds it to the **Types** category under the package to which the class belongs, rather than under the class itself.

Initializing static attributes

If you select the **Static** check box on the Features window for an attribute, use the **Initial Value** box to enter an initial value. You can open a text editor for entering initialization code by clicking the Ellipses button  associated with the box.

For information on code generation for static attributes, see [Generating Code for Static Attributes](#).

Class operations

The **Operations** tab of Features window for a class enables you to add, edit, or remove operations from the model or from the current OMD view. It contains a list of all the operations belonging to the class.

Rational Rhapsody enables you to create sets of standard operations for classes and events. For more information, see [Using Standard Operations](#).

You can create the following types of operations:

- ◆ [Primitive operations](#)
- ◆ [Receptions](#)
- ◆ [Triggered operations](#)
- ◆ [Constructors](#)
- ◆ [Destructors](#)

Primitive operations

A *primitive operation* is one whose body you define yourself instead of letting Rational Rhapsody generate it for you from a statechart.

Creating a primitive operation

To create a primitive operation using the Features window for a class:

1. On the Rational Rhapsody browser, double-click a class to open its Features window.
2. On the **Operations** tab, either click the <New> row in the list of operations or click the New button in the upper, right corner of the window and select **PrimitiveOperation**. The new row is filled in with the default values.
3. By default, Rational Rhapsody names the new primitive operation Message_n, where n is greater than or equal to 0. Type the new name for the operation in the **Name** column.
4. Change the other default values as necessary.
5. Click **OK**.

Alternatively, you can create a primitive operation through the use of the Rational Rhapsody browser, as follows:

1. In the Rational Rhapsody browser, right-click the class, actor, operation, or use case node to which you want to add the operation and select **Add New > Operation**. Alternatively, you can select the item and choose **Edit > Add New > Operation** from the menu bar.
2. Rename the operation.

Defining the features of a primitive operation

The Features window for a primitive operation enables you to change the features for it, including its return values, arguments, and modifiers. On the **General** tab, you define the general features for a primitive operation through the various controls on the tab. Notice that the signature for the primitive operation is displayed at the top of the **General** tab of the Features window.

- ♦ In the **Name** box you specify the name of the element. The default name is Message_n, where n is an incremental integer starting with 0. To enter a detailed description of the operation, use the **Description** tab.
- ♦ You use the **L** button to open the Name and Label window to specify the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#) for information on creating labels.

- ◆ In the **Stereotype** list you specify the stereotype of the attribute, if any. For information on creating stereotypes, see [Stereotypes](#).
 - To select from a list of current stereotypes in the project, click the Select Stereotype button .
 - To sort the order of the selected stereotypes, click the Change Stereotype Order button .

Note: The COM stereotypes are constructive; that is, they affect code generation.

- ◆ In the **Visibility** list you specify the visibility of the primitive operation: **Public**, **Protected**, or **Private**.
- ◆ In the **Type** drop-down list box you specify the operation type. For primitive operations, this box is set to **Primitive Operation**. If this is a template class, select the **Template** check box. To specify the necessary arguments for the template, use the **Template Parameters** tab that displays once you select the **Template** check box. For more information, see [Creating a template class](#).
- ◆ In the **Returns** area you specify the return type of a function.
 - If the function will return a defined type, select the **Use existing type** check box and select the return type from the **Type** drop-down list box that displays once you select the check box. Or to define a new type, delete the value in the **Type** drop-down list box to enable the Invoke Features window button and click it to open the Type window.

For more information on creating types, see [Composite types](#).

 - If you want to use a type that is not defined, clear the **Use existing type** check box. A **C++[Java] Declaration** box displays. Enter the code as you want it to appear in the return statement. To access the text editor, click the Ellipses button .
- ◆ In the **Modifiers** area you specify the modifiers of the operation. The possible values are **Virtual**, **Static**, **Inline**, **Constant**, or **Abstract**, but the available modifier types vary according to the type of operation.

Receptions

A *reception* specifies the ability of a class to react to a certain event (called a *signal* in the UML). The name of the reception is the same as the name of the event; therefore, you cannot change the name of a reception directly. Receptions are displayed under the **Operations** category for the class.

Creating a reception using the Features window

To create a reception using the Features window:

1. On the Rational Rhapsody browser, double-click a class to open its Features window.
2. On the **Operations** tab, either click the <New> row in the list of operations or click the New button in the upper, right corner of the window and select **Reception** from the pop-up menu. The new row is filled in with the default values.
3. Type the name of the reception in the **Event** box on the New Reception window. If Rational Rhapsody cannot find an event with the given name, a confirmation box opens, prompting you to create a new event. Click **Yes** to create a new event and the specified reception.
4. Open the Features window for the new reception operation you just created and set its other values as necessary.
5. Click **OK**.

Creating a reception using the browser

To create a reception using the Rational Rhapsody browser:

1. In the browser, right-click a class, actor, operation, or use case node and select **Add New > Reception**. The New Reception window opens.
2. Type the name of the new reception and click **OK**.
3. The following action happens depending on what Rational Rhapsody finds:
 - ♦ If Rational Rhapsody finds an event with the specified name, it creates the new reception and displays it in the browser.
 - ♦ If Rational Rhapsody cannot find an event with the given name, a confirmation box opens, prompting you to create a new event. Click **Yes** to create a new event and the specified reception.

Note the following information:

- ♦ When you add a new reception with a new name to a class, an event of that name is added to the package. If you specify an existing event name, the reception simply points to that event.
- ♦ Receptions are inherited. Therefore, if you give a trigger to a transition with a reception name that does not exist in the class but does exist in its base class, Rational Rhapsody does not create a new reception.

Reception features

The Features window for a reception enables you to change the features of a reception, including its type and the event to which the reception reacts. On the **General** tab, you define the general features for a reception through the various controls on the tab. Notice that the signature for the reception is displayed at the top of the **General** tab of the Features window.

- ♦ In the **Name** box you specify the name of the reception. The default name is event_n, where n is an incremental integer starting with 0. To enter a detailed description of the reception, use the **Description** tab.
- ♦ If the **Name** box is inaccessible, click the **L** button to open the Name and Label window to change the name, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ In the **Stereotype** list you specify the stereotype of the attribute, if any. For information on creating stereotypes, see [Stereotypes](#).
 - To select from a list of current stereotypes in the project, click the Select Stereotype button .
 - To sort the order of the selected stereotypes, click the Change Stereotype Order button .

Note: The COM stereotypes are constructive; that is, they affect code generation.

- ♦ In the **Visibility** list you specify the visibility of the reception (**Public**, **Protected**, or **Private**), if available.
- ♦ In the **Type** drop-down list box you specify the operation type. For receptions, this box is set to Reception.
- ♦ In the **Event** list you specify the event to which the reception reacts. To view or modify the features of the event itself, click the Invoke Features window button.

Deleting receptions

You cannot delete receptions in the following cases:

- ♦ The reception is used by the statechart of the class.
- ♦ The reception is used by a derived statechart of a class that does not have its own reception.

Triggered operations

A *triggered operation* can trigger a state transition in a statechart, just like an event. The body of the triggered operation is executed as a result of the transition being taken. For more information, see [Triggered operations](#).

Constructors

Constructors are called when a class is instantiated, generally to initialize data members with values relevant to that object.

Rational Rhapsody has the following constructor icons:



The Rational Rhapsody browser icon for a constructor is a red triangle with a black arrow.



The icon for a protected constructor is overlaid with a key.



The icon for a private constructor is overlaid with a padlock.

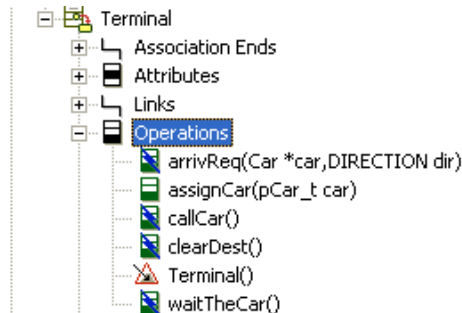
Creating a constructor

To create a constructor:

1. Depending on if you want to use the Features window or the Rational Rhapsody browser:
 - ♦ On the Rational Rhapsody browser, double-click a class to open its Features window and on the **Operations** tab, either click the **<New>** row in the list of operations or click the New button in the upper, right corner of the window and select **Constructor**.
 - ♦ On the Rational Rhapsody browser, right-click either the class or the **Operations** category under the class and select **Add New > Constructor**.
2. The Constructor Arguments window opens.
3. Click **Add**. The Argument window opens.
4. Type in a name for the new constructor and change the default values as necessary.
5. Click **OK** twice.

Note: Alternatively, you can open this window by right-clicking the appropriate element in a diagram and selecting **New Constructor**.

The new constructor is listed under the **Operations** category for the class in the Rational Rhapsody browser.



Defining constructor features

The Features window enables you to change the features of a constructor, including its arguments and initialization code. Double-click the constructor in the Rational Rhapsody browser to open its Features window.

On this **General** tab, you define the general features for a constructor through the various controls on the tab. Notice that the signature for the constructor is displayed at the top of the **General** tab of the Features window.

- ◆ In the **Name** box you specify the name of the constructor. The default name is the name of the class it creates. To enter a detailed description of the constructor, use the **Description** tab.
- ◆ If the **Name** box is inaccessible, click the **L** button to open the Name and Label window to change the name, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ◆ In the **Stereotype** list you specify the stereotype of the attribute, if any. For information on creating stereotypes, see [Stereotypes](#).
 - To select from a list of current stereotypes in the project, click the Select Stereotype button .
 - To sort the order of the selected stereotypes, click the Change Stereotype Order button .

Note: The COM stereotypes are constructive; that is, they affect code generation.

- ◆ In the **Visibility** list you specify the visibility of the reception (**Public**, **Protected**, or **Private**), if available. The default value is **Public**.

- ◆ In the **Initializer** box you enter code if you want to initialize class attributes or super classes in the constructor initializer. To access the text editor, click the Ellipses button .

For example, to initialize a class attribute called a to 5, type the following code:

```
a(5)
```

Note: In C++, this assignment is generated into the following code in the class implementation file to initialize the data member in the constructor initializer rather than in the constructor body:

```
//-----  
// A.cpp  
//-----  
  
A::A() : a(5) {  
    //#[operation A()  
    //#]  
};
```

Note: You must initialize const data members in the constructor initializer rather than in the constructor body.

Adding initialization code

To enter code for any initializations that you want to perform in the constructor body rather than in the constructor initializer, use the **Implementation** tab. You can create and initialize objects participating in relationships within the body of a constructor. You can pass arguments to these objects if they have overloaded constructors using, for example:

```
new relatedClass(3)
```

This code in the body of the class constructor calls the constructor for the related class and passes it a value of 3. The related class must have a conversion constructor that accepts a parameter. The constructor of the related class then performs its initialization using the passed-in value.

Destructors

A *destructor* is called when an object is destroyed, for example, to de-allocate memory that was dynamically allocated for an attribute during construction.

Rational Rhapsody has the following destructor icons:



The Rational Rhapsody browser icon for a destructor is a tombstone (RIP = Rest In Peace).



The icon for a protected destructor is overlaid with a key.



The icon for a private destructor is overlaid with a padlock.

Creating a destructor

To create a destructor, follow the instructions for [Creating a primitive operation](#), but for the type, select **Destructor**.

Modifying the features of a destructor

Use the Features window to change the features of a destructor including its visibility and modifier.

On this **General** tab, you define the general features for a destructor through the various controls on the tab.

- ♦ In the **Name** box you specify the name of the destructor. By definition, destructors have the same name as the class, preceded by a tilde (~) symbol. To enter a detailed description of the reception, use the **Description** tab.
- ♦ If the **Name** box is inaccessible, click the **L** button to open the Name and Label window to change the name, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ In the **Stereotype** list you specify the stereotype of the attribute, if any. For information on creating stereotypes, see [Stereotypes](#).
 - To select from a list of current stereotypes in the project, click the Select Stereotype button .
 - To sort the order of the selected stereotypes, click the Change Stereotype Order button .

Note: The COM stereotypes are constructive; that is, they affect code generation.

- ♦ In the **Visibility** list you specify the visibility of the reception (**Public**, **Protected**, or **Private**). By default, destructors are **Public**.

- ◆ In the **Modifiers** area you specify the modifiers of the destructor. Select **Virtual**, if wanted.

To type code for the body of the destructor, use the **Implementation** tab.

Define class ports

Use the **Ports** tab to create, modify, and delete class ports.

The **Ports** tab contains the following columns:

- ◆ **Name** specifies the name of the port.
- ◆ **Contract** specifies the contract of the port. For more information about contracts, see [The Port Contract tab](#). The possible values are as follows:
 - **Implicit** means that the contract is a “hidden” interface that exists only as the contract of the port.
 - **Explicit** mean that the contract is an explicit interface in the model. An explicit contract can be reused so several ports can have the same contract.
- ◆ **Multiplicity** specifies the multiplicity of the port. The default value is 1.
- ◆ **Behavior** specifies whether the port is a behavioral port, which means that the messages of the provided interface are forwarded to the owner class. If it is *non-behavioral*, the messages are sent to one of the internal parts of the class.
- ◆ **Reversed** specifies whether the interfaces are reversed, so the provided interfaces become the required interfaces and vice versa.

For instructions on how to use the interface on this tab to create, modify, or delete a port, see [Defining the attributes of a class](#).

For detailed information about specifying ports, see [Ports](#).

Define relations

The term “relations” refers to all the relationships you can define between elements in the model (not just classes); for example, associations, dependencies, generalization, flows, and links. (In previous versions of Rational Rhapsody, the term referred to all the different kinds of associations.)

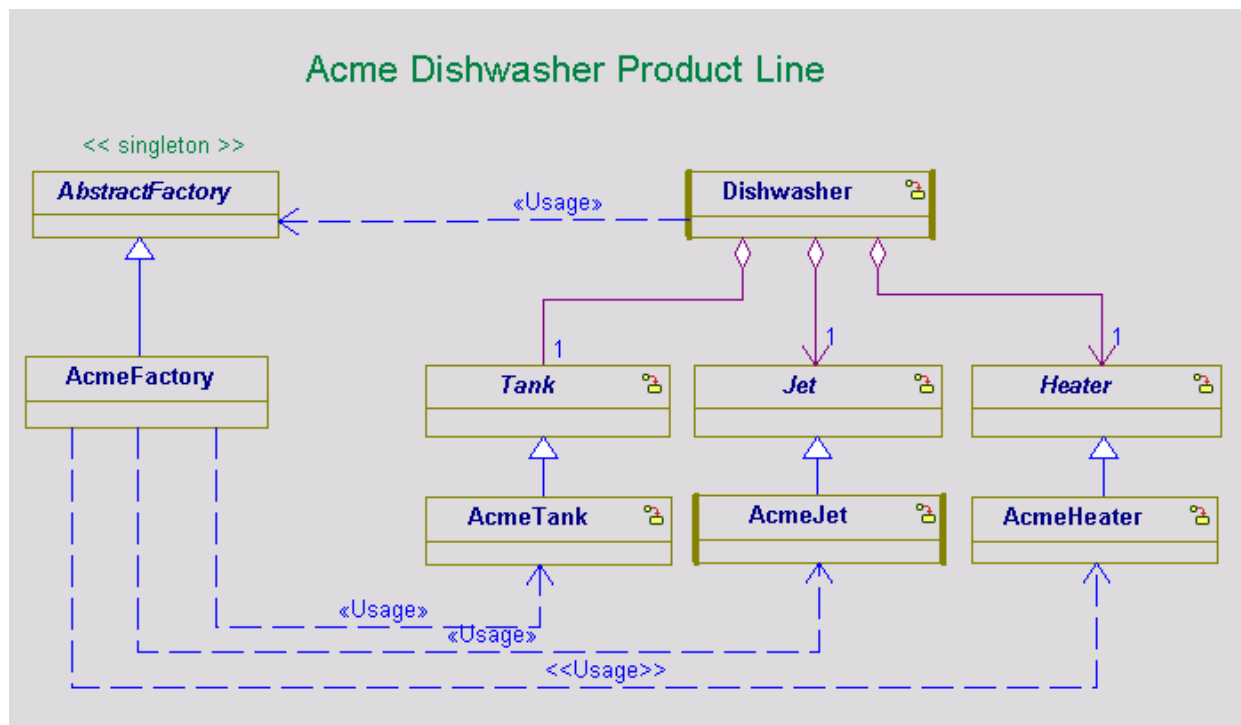
The **Relations** tab lists all the relationships (dependencies, associations, and so on) the class is engaged with.

The **Relations** tab contains the following columns:

- ◆ **Name** specifies the name of the relation.

- ◆ **Type** specifies the relation type (for example, **Association End** and **Dependency**).
- ◆ **Direction** specifies the direction of the relationship (**From** or **To**).
- ◆ **From/To** specifies the target of the relationship. For example the dependency `Dishwasher.AbstractFactory` goes from the `Dishwasher` class to the `AbstractFactory`.
- ◆ **Data** specifies any additional data used by the relationship.

For example, if you have a port that provides the interface `MyInterface`, the **Data** column would list `Provided Interface`.



For more information on relationships, see [Associations](#).

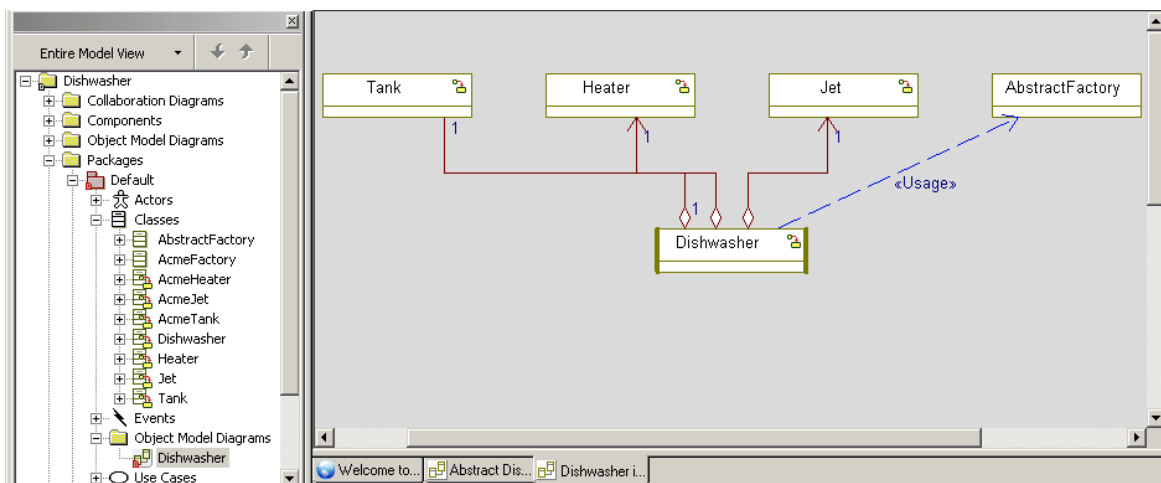
Showing all relations for a class, object, or package in a diagram

To understand the full meaning or purpose of a class, Rational Rhapsody lets you create an object model diagram that shows a class and its relations to all the other elements in a project.

Note that you can show the relations for a class, an object, and a package. You would use the same procedure as noted below, except that you would select that particular element (for example, an object) instead.

To show all the relations of a class:

1. Right-click a class in the Rational Rhapsody browser and select **Show Relations in New Diagram**.
2. Notice the following information:
 - ♦ Rational Rhapsody created an object model diagram that shows all the relations for the selected class.
 - ♦ Rational Rhapsody also named the new object model diagram from the name of the class you selected and it created an **Object Model Diagram** category to hold the new diagram within the package where the class resides.



Note the following information:

- ♦ The **Show Relations in New Diagram** pop-up menu command is available for classes, objects, and packages from the Rational Rhapsody browser as well as on a diagram. For both the browser and on a diagram, you would right-click the element (for example, an object) and select **Show Relations in New Diagram**.
 - Note:** For a diagram you can select multiple elements (for example, two classes). Use **Ctrl+Click** to make multiple selections and then right-click one of the selected elements to open the pop-up menu and click **Show Relations in New Diagram**.
- ♦ The location of the new object model diagram created by **Show Relations in New Diagram** depends on whether you started the command from the browser or a diagram:
 - When started from the browser, **Show Relations in New Diagram** creates the diagram in the same location as the selected element and places it in an **Object Model Diagram** category (which Rational Rhapsody creates if the category is not already available).
 - When started from a diagram, **Show Relations in New Diagram** creates the object model diagram in the same location in which the source diagram resides.
- ♦ The new object model diagram created by **Show Relations in New Diagram** will be named the same name as the class, object, or package from which it was created. If there is already an object model diagram with that name, a number will be appended to the name (for example, Dishwasher_9).
- ♦ See also [Automatically populating a diagram](#).

Defining class tags

The **Tags** tab lists the available tags for this class. For detailed information on tags, see [Profiles](#).

Defining class properties

The **Properties** tab lets you set and edit class properties and displays the definitions for individual properties with their default values. The definition of an individual property displays at the bottom of the Features window when a property is selected in the **Property** tab.

Adding a class derivation

A class *derivation* is modeled as a dependency relationship with a stereotype of <<derive>>. Code is not generated from that relationship.

The <<derive>> stereotype specifies a derivation relationship among model elements that are usually, but not necessarily, of the same type. A derived dependency specifies that:

- ♦ The client might be computed from the supplier.
- ♦ The mapping specifies the computation.

The client might be implemented for design reasons, such as efficiency, even though it is logically redundant.

To create a class derivation:

1. Right-click the class for which you are creating a derivation and select **Add New > Relations > Derivation**.
2. In the window, select from the **Depends on** drop-down menu. The items in the drop-down menu are as follows:
 - ♦ Elements currently in the model.
 - ♦ Profiles for the development language, as defined in [Opening an existing Rational Rhapsody project](#).
 - ♦ <<Select>> displays another window with model browser to allow you to make a selection that is not listed in the drop-down menu.
3. Make your selection and click **OK**. The selected element is then listed under the Class Derivation in the browser.

Making a class an instance

To make a class an instance in an OMD, select the **Instance** tab in the Features window.

If the class represents an instance, select the **This box is also an instance** check box. This is equivalent to giving the class an instance name in the OMD. If this box is checked, the following boxes become active:

- ◆ **Instance Name** specifies the name of the instance.
- ◆ **Multiplicity** specifies the number (or range) of times to instantiate the class.

The multiplicity indicates the number of instances that can participate at either end of a relation. Multiplicity can be shown in terms of a fixed number, a range, or an asterisk (*), meaning any number of instances including zero.

Defining class behavior

To define the behavior of a class, you give it either a statechart or an activity diagram:

1. In the OMD, right-click the class
2. Select either **New Statechart** or **New Activity Diagram**.

For more information on these diagrams, see [Statecharts](#) or [Activity diagrams](#).

Generating, editing, and roundtripping class code

Rational Rhapsody enables you to generate code and open a text editor for editing the generated code directly from within an OMD. The following sections describe these tasks in detail.

Generating class code

To generate code for a single class:

1. Right-click the class and then select **Generate**.
2. If a directory for the configuration that the class belongs to has not yet been created, Rational Rhapsody asks if you want to create the directory. The configuration directory is under the component directory. Click **Yes**.
3. Rational Rhapsody creates the configuration directory and generates the class code to it. An output window opens at the bottom of the Rational Rhapsody window for the display of code generation messages.

Editing class code

To edit code once it has been generated, right-click the class and select **Edit Code**. By default, the code generated for the class opens in the Rational Rhapsody internal code editor. If both a specification and an implementation file were generated for the class, both files open, with the specification file in the foreground.

To set Rational Rhapsody to open the editor associated with the file extension instead of the internal code editor:

1. Select **File > Project Properties**.
2. Set the `General::Model::ClassCodeEditor` property to `Associate`.
3. Click **OK**.

Roundtripping class code

When generating code, Rational Rhapsody places all user code for method bodies and transition code written in statecharts between special annotation symbols. The symbols are as follows:

Language	Body Annotation Symbols
Ada	--+[<ElementType> <ElementName> --+]
C	/*#[<ElementType> <ElementName> */ /*#]*/
C++ and Java	//#[<ElementType> <ElementName> //#]

For example, the following `Initialize()` operation for the `Connection` class in the PBX sample contains user code that was entered in the `Implementation` field of the `Operation` window. The user code is placed between the annotation symbols when code is generated for the class:

```
void Connection::Initialize() {  
    /*#[ operation Initialize()  
    DigitsDialed = 0;  
    Digits[0] = 0;  
    Digits[1] = 0;  
    Busy = FALSE;  
    Extension = 0;  
    /*#]  
}
```

You can edit the code between the *annotation symbols* in a text editor and then roundtrip your changes back into the model. The roundtripped edits will be retained upon the next code generation. This is how Rational Rhapsody keeps the code and the model in sync to provide model-code associativity.

Note

Any text edits made outside the annotation symbols might be lost with the next code generation. For more information, see [Deleting Redundant Code Files](#).

To roundtrip code changes back into the model:

1. Edit the generated class code between the `/// and /// annotation symbols.`
2. In the browser or diagram, right-click the class containing the code that you just edited and select **Roundtrip**.

If you view the Implementation box of the specification window for the operation (or the statechart for the class if you edited transition code), you can see that your text edits were added to the model.

Opening the main diagram for a class

You can specify a main diagram for a class in the Features window. This is usually the diagram that shows the most important information for a class. For example, in the PBX sample, the PBX diagram is specified as the main diagram for the `Connection` class. The main diagram for a class must be either an object model diagram (OMD) or a use case diagram (UCD).

1. In the OMD, right-click the class.
2. Select **Open Main Diagram**.

Display option settings

Rational Rhapsody allows a great deal of flexibility in how elements are displayed. Display options relate to how the element name, stereotype, and compartments are displayed in the diagram. For example, to change the display options for a class, right-click the class in the diagram and select **Display Options** (or select it and choose **Edit > Display Options**). The Display options for the selected class opens.

General tab display options

The first tab of the Display Options window enables you to set general display options, including the class name and stereotype. The **General** tab contains the following controls:

- ♦ **Display name** specifies how the class name is displayed. A class is always inside at least one package, but the package can be nested inside other packages, and the class can also be nested inside one or more classes. The class name displayed in the OMD can show multiple levels of nesting in the class name.

The possible display options are as follows:

- **Full path** means the full path name and includes the entire class nesting scheme in the following format:

`<p1>::<p2>::...::<pn>::<c1>::<c2>::... ::<classname>` In this syntax, `p[n]` are packages and `c[n]` are classes.
- **Relative** means the relative name that shows nesting of a class inside other classes, depending on its context within the diagram. For example, if class A contains class B, then inside of A, the relative name for B is B, but outside of A, the relative name for B is A::B.
- **Name only** means this option displays only the class name without any path information.
- **Label** means this option displays the label for the class.
- ♦ **Show Stereotype Label** indicates whether or not to display the class stereotype as text at the bottom of the class box between guillemet symbols (for example, «Interface»).
- ♦ **Show Compartment Label** indicates whether or not to display the labels of available compartments. If you check the box, you then select the compartment for which you want the labels displayed from the list in the **Available** column of this window. Select your selections and click the **Display** button to move them into the **Displayed** column.
- ♦ **Image View** specifies how the image is displayed. Check the box “Enable Image View” and either of the following options:
 - **Use Associated Image** means this option uses the default Rational Rhapsody provided image for the object selected in Rational Rhapsody.
 - **Select An Image** means if this option is selected, Rational Rhapsody displays the **Image File Path** for you to locate it on your computer and click **OK**.

- ◆ **Advanced** opens the Advanced Image View Options window. See [Advanced Image View Options window](#)
- ◆ **Ports** specifies whether to show new ports and their interfaces. Rational Rhapsody allows you to specify which ports to display in the diagram using the **Show New Ports** functionality. For more information, see [Ports](#).

Advanced Image View Options window

To open the Advanced Image View Options window, click the **Advanced** button on the **General** tab of the Display Options window, when it is available. This window has the following options:

- ◆ Select **Image Only** to display the just the image.
- ◆ Select **Structured** to see the picture in a separate compartment below the name compartment for the object.
- ◆ Select **Compartment** to see the picture in a separate compartment of its own between the object name compartment and bottom compartment.

Displaying attributes and operations

The **Attributes** tab in the Display options window enables you to select which, if any, attributes to display in the diagram.

To specify which elements to displayed in the diagram:

1. Select the element in the **All Elements** list.
2. Click the **Display** button to move the element to the **Shown in Diagram** list.
3. Repeat for each element or simply click the **All** button to select all of the elements in the list and display them.
4. You can move elements up and down in the **Shown In Diagram** list or remove them using the other three buttons in this window.
5. Click **OK**.

Similarly, the **Operations** tab allows you to select which, if any, operations to display in the diagram.

Removing or deleting a class

You can remove a class from the current view (diagram) or delete the class entirely from the model. Removing a class from the view does not delete it from the model.

To remove a class from the diagram, right-click the class and select **Remove from View**.

To delete the class entirely from the model, do one the following actions:

- ◆ Right-click the class, then select **Delete from Model**.
- ◆ Select the class, then click **Delete** in the main toolbar.

Note

When you delete a class, all of its objects are also deleted.

Ports

A *port* is a distinct interaction point between a class and its environment or between (the behavior of) a class and its internal parts. A port allows you to specify classes that are independent of the environment in which they are embedded. The internal parts of the class can be completely isolated from the environment and vice versa.

A port can have the following interfaces:

- ◆ **Required interfaces** characterizes the requests that can be made from the class for a port (via the port) to its environment (external objects). A required interface is denoted by a socket notation.
- ◆ **Provided interfaces** characterizes the requests that could be made from the environment to the class via the port. A provided interface is denoted by a lollipop notation.

These interfaces are specified using a *contract*, which by itself is a provided interface. For more information, see [The Port Contract tab](#).

If a port is *behavioral*, the messages of the provided interface are forwarded to the owner class; if it is *non-behavioral*, the messages are sent to one of the internal parts of the class. Classes can distinguish between events of the same type if they are received from different ports.

Note

See the HomeAlarmwithPorts sample model (under <Rational Rhapsody installation path>\Samples\CppSamples) for an example of a model that uses ports.

Partial specification of ports

If you specify ports without any contract (for example, an implicit contract with no provided and required interfaces), Rational Rhapsody assumes that the port relays events using the code generator. You could link two such ports and the objects would be able to exchange events via these ports. However, Rational Rhapsody will notify you during code generation (with warnings or informational messages) because the specification is still incomplete.

Considerations

Ports are interaction points through which objects can send or receive messages (primitive operations, triggered operations, and events). Ports in UML have a type, which in Rational Rhapsody is called a *contract*. A contract of a port is like a class for an object.

If a port has a contract (for example, interface *I*), the port provides *I* by definition. If you want the port to provide an additional interface (for example, interface *J*), then, according to UML, *I* must inherit *J* (because a port can have only one type). In the case of Rational Rhapsody, this inheritance is created automatically once you add *J* to the list of provided interfaces (again, this is a port with an explicit contract *I*). According to the UML standard, if *I* and *J* are unrelated, you must specify a new interface to be the contract and have this interface inherit both *I* and *J*.

Implicit port contracts

Some found that enforcing a specification of a special interface as the contract for a port to be artificial, so Rational Rhapsody provides the notion for an *implicit contract*. This means that if the contract is implicit, you can specify a *list* of provided and required interfaces that are not related to each other, whereas the contract interface remains implicit (no need to explicitly define a special interface to be the contract for the port in the model).

Working with implicit contracts has pros and cons. If the port is connected to other ports that provide and require only subsets of its provided and required interfaces, it is more natural to work with implicit contracts. However, if the port is connected to another port that is exactly “reversed” (see the check box in the Features window for the port) or if other ports provide and require the same set of interfaces, it makes sense to work with explicit contracts. This is similar to specifying objects separately from the classes, or objects with implicit classes in the case when only a single object of this type or class exists in the system.

Rapid ports

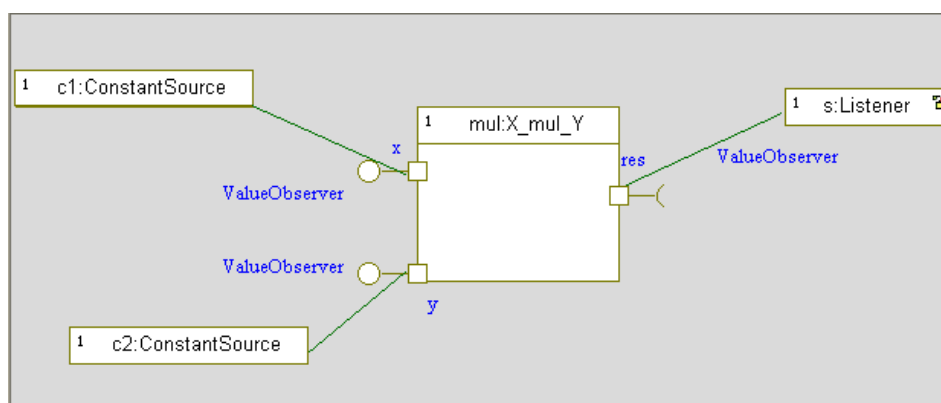
Rapid ports are ports that have no provided and required interfaces (which means that the contract is implicit, because a port with an explicit contract, by definition, provides a contract interface). These ports relay any events that come through them. The notion of rapid ports is Rational Rhapsody-specific, and enables users to do rapid prototyping using ports. This functionality is especially beneficial to users who specify behavior using statecharts, without the need to elaborate the contract at the early stages of the analysis or design.

Creating a port

To create a port in an object model diagram:

1. Click the **Port** button.
2. Click the class boundary to place the port. A text box opens so you can name the new port.
3. Type the name for the port, then press **Enter** to dismiss the box. Note that the port label uses the convention `portName{[multiplicity]}`. For example:
 - a. `p`
 - b. `p[5]`
 - c. `p[*]`

The new port displays as a small square on the boundary of its class.



Alternatively, you can create a port in the following ways:

- ◆ Use the **Ports** tab of the Features window for the class. For more information, see [Define class ports](#).
- ◆ Right-click the class in the browser and then select **Add New > Port**.

Specifying the features of a port

As with all elements, you use the Features window to specify the features that define a port. The Features window for a port includes five tabs: **General**, **Contract**, **Relations**, **Tags**, and **Properties**.

The Port General tab

On the **General** tab, you define the general features for a port through the various controls on the tab.

- ♦ In the **Name** box you specify the name of the port. The default name is port_n, where n is an incremental integer starting with 0. To enter a detailed description of the attribute, use the **Description** tab.
- ♦ You use the **L** button to open the Name and Label window to specify the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ In the **Stereotype** list you specify the stereotype of the port. For information on creating stereotypes, see [Stereotypes](#).
 - To select from a list of current stereotypes in the project, click the Select Stereotype button .
 - To sort the order of the selected stereotypes, click the Change Stereotype Order button .

Note: The COM stereotypes are constructive; that is, they affect code generation.

- ♦ In the **Contract** drop-down list box you specify the port contact. The list contains the following possible values:
 - **<Implicit>** means the contract is implicit.
 - **<New>** enables you to define a new contract. If you select this value, Rational Rhapsody displays a separate class Features window so you can define the new interface.
 - A list of classes with a stereotype that includes the word “interface.”

The arrow button next to the **Contract** box opens the Features window for the contract. However, this button is disabled if the contract is implicit.

- ♦ In the **Multiplicity** drop-down list box you specify the multiplicity of the port. The multiplicity is included in the port label if it is greater than 1.
- ♦ In the **Visibility** drop-down list box you specify the visibility for the port (**Public**, **Protected**, or **Private**). The default value is **Public**.

- ♦ In the **Attributes** area you specify the port attributes:
 - If you select the **Behavior** check box, messages sent to the port are relayed to the owner class. By default, the check box is cleared.
 - If you select the **Reversed** check box, the provided interfaces become the required interfaces, and the required interfaces become the provided interfaces.

The Port Contract tab

The **Contract** tab enables you to specify the port contract. The contract specifies the provided and required interfaces through relations to other interfaces.

There are two types of contract:

- ♦ **Explicit** means the contract is an explicit interface in the model. An explicit contract can be reused so several ports can have the same contract.
- ♦ **Implicit** means the contract is a “hidden” interface that exists only as the contract of the port.

For both provided and required interfaces, three buttons are available:

- ♦ **Add** to add a new interface to the list of available interfaces. For provided interfaces, this means that the contract inherits the selected interface; for required interfaces, this means that the contract has a new dependency stereotyped «Usage» towards the interface.
- ♦ **Edit** to open the Features window for the selected element so you can modify it.
- ♦ **Remove** to remove the relation with the contract for the selected interface.

Note that if you selected the **Reversed** check box on the **General** tab, the bottom of the **Contract** tab displays a message in red stating that the contract is reversed.

Specifying the port contract

To specify the contract information for the port:

1. To specify the provided interfaces, select the **Provided** folder icon, then click the **Add** button in the top group box. The Add new interface window opens.
2. Either type in the new name of the interface, or use the list to specify the interface.
3. Click **OK**.
4. You return to the **Contract** tab, which now lists the provided interface you just specified.
5. To specify the required interface, click the **Required** folder, then select **Add**. The Add New Interface window opens.

6. Specify the required interface and then click **OK**.

Note: If a provided interface (including the contract) has an association to another interface, the other interface is a required interface.

7. Click **OK**.

Note: If an interface provided by a port inherits from another interface, then by definition, the port also provides the base interface. This means that if you want to remove the base interface from the contract, you must remove the generalization between the two interfaces. (Before removing such an interface, Rational Rhapsody will notify you that the generalization will also be removed.)

Display options for ports

The owning class or object specifies whether ports and their interfaces are displayed. By default, new ports and their interfaces are displayed.

To disable these default settings:

1. In the diagram, right-click the class or object that owns the port and select **Display Options**.
2. Clear the **Show Ports Interfaces** or **Show New Ports** check box, as intended.
3. Click **OK**.

You can specify how the port name and stereotype are displayed using the Display Options window for the port itself.

For more information on displaying ports, see [Selecting which ports to display in the diagram](#).

The Tags tab

The **Tags** tab lists the tags available for the port. For detailed information on tags, see [Profiles](#).

The Properties tab

The **Properties** tab lets you set and edit the properties that affect your model. The definition of each property displays at the bottom of the Features window when you select a property in the list.

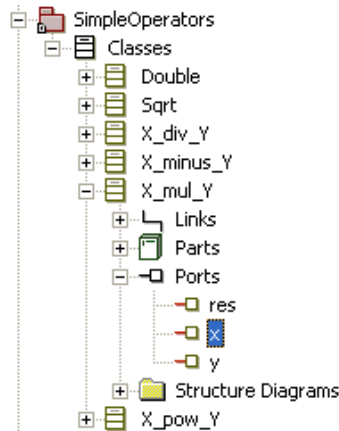
The following table shows the properties (under the `ObjectModelGe` subject) that control how ports are displayed.

Metaclass	Property	Description
Class/Object	ShowPorts	Determines whether ports are displayed in OMDs
	ShowPortsInterfaces	Determines whether ports are displayed in OMDs
Port	color	Specifies the color used to draw the port
	FillColor	Specifies the default fill color for the port
	name_color	Specifies the default color of the port name
	UseFillColor	Specifies whether to use the fill color for the port

Note that you cannot selectively show ports in diagrams, either all the ports are displayed, or none of them are.

Viewing ports in the browser

Ports are displayed in the browser under the appropriate class.



Connecting ports

To exchange messages using ports, you must specify links between their objects. You can use one of these methods:

- ◆ Draw a link to the ports as described in [Links and associations](#).
- ◆ Right-click each object individually and select **Make an Object**. This allows you to link the two new objects via their ports without explicitly creating a link between the ports.

You can specify a link between a part (or a port of a part) to a port belonging to the enclosing class.

However, you cannot specify associations via ports nor specify a link between classes, even if the ends are connected to ports.

Using rapid ports

Rational Rhapsody supports *rapid ports*: you can simply draw ports, connect them via links, create a statechart, and the ports will exchange events without any additional information. In addition, if a port is not connected to any of the internal parts for a class, the code generator assumes it is a behavioral port and messages will be relayed to or from the class. In rapid mode, the classes must be reactive because Rational Rhapsody assumes that events are exchanged.

Rapid ports would be useful in the following situations:

- ◆ In component-based design. For example, when you have a class to be reused in different systems and has a behavior of its own (not that of one of its parts) that provides and requires the interfaces of the contract for the port.
- ◆ The class has a statechart in which the triggers of the transitions are based on the ports through which the events were received. In other words, because the statechart is able to distinguish between the ports through which an event was sent, it could react differently to the same events based on which port the event came from.

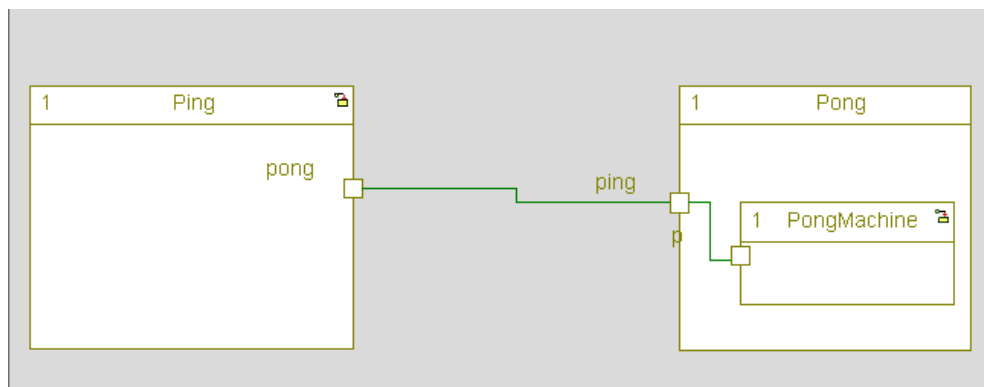
Note

Once you specify the contract on a port, you must specify the contract on all the ports that are connected to it. Otherwise, the code generator will issue a warning that there is a mismatch in the contracts and the links will not be created.

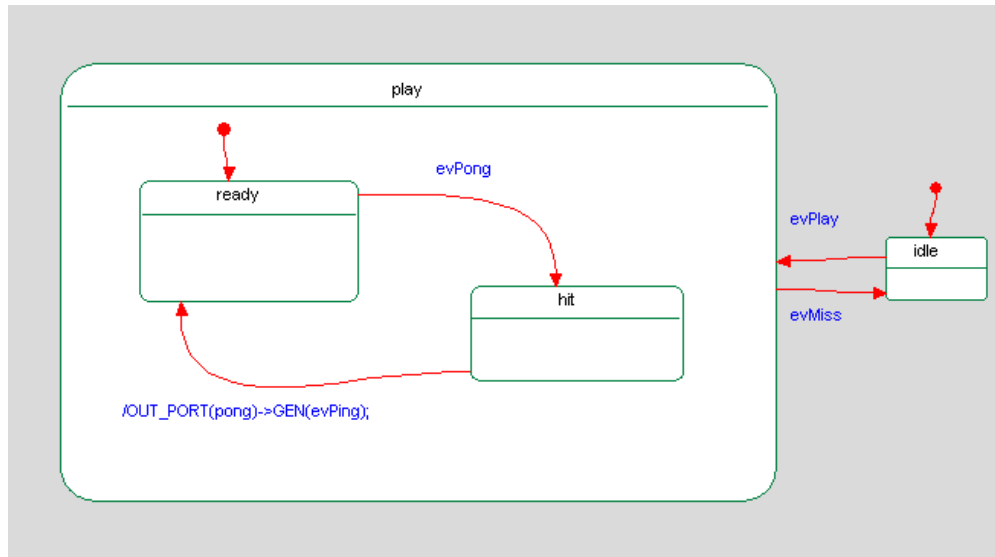
Rational Rhapsody uses the values of the following framework properties to implement the rapid ports:

- ◆ `DefaultProvidedInterfaceName` specifies the interface that must be implemented by the “in” part of a rapid port.
- ◆ `DefaultReactivePortBase` stores the base class for the generic rapid port (or default reactive port). This base class relays all events
- ◆ `DefaultRequiredInterfaceName` specifies the interface that must be implemented by the “out” part of a rapid port
- ◆ `DefaultReactivePortIncludeFile` specifies the include files that are referenced in the generated file that implements the class with the rapid ports

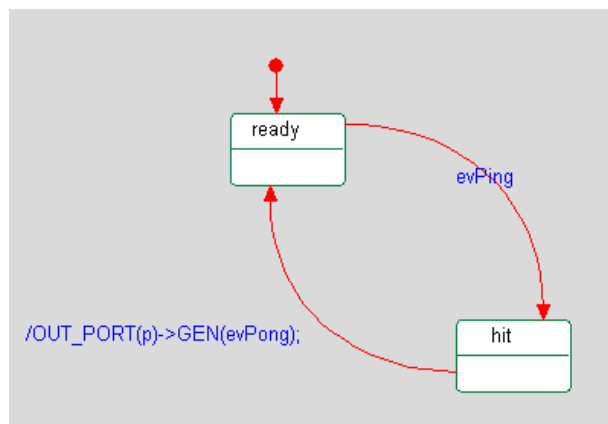
Consider the following figure, which shows an OMD that uses rapid ports.



The following figure shows the statechart for the Ping class.



The following figure shows the statechart for the PongMachine part.



During animation, the two objects exchange evPing and evPong events.

Selecting which ports to display in the diagram

If you right-click a class and select **Ports**, the following options are available:

- ♦ **New Port** creates a new port on the specified class.
- ♦ **Show All Ports** shows all the ports that currently exist in the specified model class.
- ♦ **Hide Ports** hides all the ports that are currently displayed by the specified model class. However, ports that are created later will be displayed.

Using this show/hide functionality in conjunction with the **Show New Ports** display option for the owning class, you can show and hide ports as intended to simplify your model. For more information on display options for classes, see [General tab display options](#).

If you set **Show New Ports** mode to on, each new port that is added to the class is also displayed in the diagram class. Ports created before this graphic class, or while the **Show New Ports** feature is off, are not synthesized in the diagram, unless they were created using the graphic editor **New Port** option.

If **Show New Ports** mode is off, any ports created after disabling this mode will not be displayed.

Creating a new port for a class

To create a new port:

1. Right-click a class and select **Ports**.
2. Select **New Port** creates a new port on the specified class.

Showing all ports

To show all the ports in the diagram, select the **Show New Ports** check box in the Display Options window for the owning class, and select **Ports > Show All Ports** for the class.

Showing new ports only

To show only new ports in the diagram, select the **Show New Ports** check box in the Display Options window for the owning class, and select **Ports > Hide Ports** for the class.

Hiding all ports

To hide all ports in the diagram, clear the **Show New Ports** check box in the Display Options window for the owning class, and select **Ports > Hide Ports** for the class.

Deleting a port

To remove a specific port from a class, use the **Delete from Model** or **Remove from View** option in the menu for the owning class.

Programming with the port APIs in C++

The following sections describe the APIs you use to program using ports. The topics are as follows:

- ◆ [Basic API tasks](#)
- ◆ [Intermediate-level tasks](#)
- ◆ [Advanced-level tasks](#)

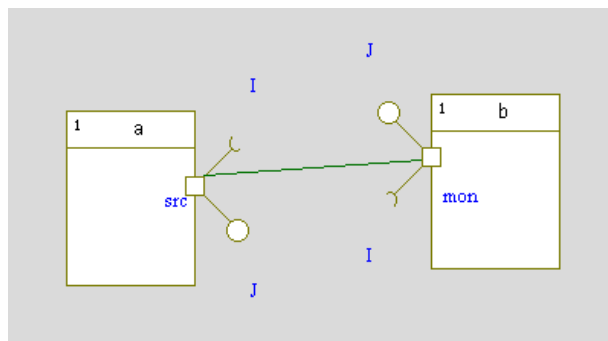
Basic API tasks

This section describes the basic APIs used to exchange messages with and instantiate ports.

Note

The following example is not complete; it is simply a reference for the subsequent table of API calls.

Consider the following example:



The following table shows the calls to use to perform the specified tasks.

Task	Call
Call an operation.	<code>OUT_PORT(src)->f();</code>
Send an event from a to b using the ports.	<code>OUT_PORT(src)->GEN(evt);</code>
Listen for an event from port src to port mon.	<code>evt[IS_PORT(mon)]/ doYourStuff();</code>

You could also use the `OPORT` macro, which is equivalent to `OUT_PORT`.

Communicating with ports with multiplicity

The following table shows the calls to use if the multiplicity of the ports is 10 and you want to communicate with the ports using index 5.

Task	Call
Call an operation.	<code>OUT_PORT_AT(src, 5)->f();</code>
Send an event from a to b using the ports.	<code>OUT_PORT_AT(src, 5)->GEN(evt);</code>
Listen for an event from port src to port mon.	<code>evt[IS_PORT_AT(mon, 5)]/ doYourStuff();</code>

You could also use the `OPORT_AT` macro, which is equivalent to `OUT_PORT_AT`.

Intermediate-level tasks

This section describes the intermediate-level APIs used when programming with ports. You use these APIs whenever the code generator cannot create the links on its own, including the following cases:

- ◆ An external source file is used to initialize the system and you must write the code to create the links between the objects.
- ◆ The port multiplicities are not definite (for example, *).
- ◆ The port multiplicities do not match across the links. This could happen when ranges of multiplicities are used (for example, 1..10).

Connecting objects via ports

If you are using an external application (such as the MFC GUI) where the links are created at run time, you can link objects with ports specified by Rational Rhapsody using calls similar to the following examples:

```
a.getSrc()->setItsJ(b.getMon()->getItsJ());  
b.getMon()->setItsI(a.getSrc()->getItsI());
```

To link the objects:

1. Create a temporary package that creates the links for you.
2. Copy the .cpp file for the new package to the correct class.
3. Modify the code as needed.

Linking objects via ports with multiplicity

Using the example in the previous API illustration if the multiplicity of both ports is 10, you would link the objects as follows:

```
for (int i=0; i<10; ++i) {  
    a.getSrcAt(i)->setItsJ(b.getMonAt(i)->getItsJ());  
    b.getMonAt(i)->setItsI(a.getSrcAt(i)->getItsI());  
}
```

Advanced-level tasks

This section describes the advanced-level APIs used when programming with ports. You use these APIs when the code generator cannot determine how to instantiate the ports. This situation occurs when the port multiplicity is `*`.

Creating ports programmatically

By default, ports are created by value. However, if at design time you do not know how many ports there will be (multiplicity of `*`), you can create the ports programmatically.

For example, to instantiate 10 of the `src` ports, use the following call:

```
for (int i=0; i<10; ++i) {  
    // instantiate and add to the container of the owner  
    newSrc();  
}
```

Linking behavioral ports to their owning instance

Similarly, if you do not know what the multiplicity of the behavioral port at design time, you can specify it programmatically.

Behavioral ports are connected to their owning instance using the method `connect[ClassName]`. For example, to connect behavioral port `p` to its owner object `a` (of type `A`), use the following call:

```
a.getP()->connectA(a);
```

If the ports in previous API illustration are behavioral, you would use the following code:

```
for (int i=0; i<10; ++i) {  
    newSrc();  
    //hooks the class so it takes care of the messages  
    getSrcAt(i)->connectA(this);  
}
```

For more efficiency, use the following code:

```
for (int i=0; i<10; ++i) {  
    newSrc()->connectA(this);  
}
```

Port code generation in C

In C, code can be generated for the following types of ports:

- ♦ Rapid ports (see [Using rapid ports](#))
- ♦ Standard ports where the provided and required interfaces contain only event receptions

Action language for sending events

The following macros are used for working with ports and events:

- ♦ Generating and sending an event via a port:

– `RiCGEN_PORT([pointer to port], [event])`

Examples:

```
RiCGEN_PORT (me->myPort, myEvent())
```

For ports with multiplicity greater than one:

```
RiCGEN_PORT (me->myPort[2], myEvent())
```

- ♦ Detecting the input port through which an event has been sent:

– `RiCIS_PORT([pointer to port])`

Examples:

```
RiCIS_PORT(me->myPort)
```

This returns True if the event currently being handled by the Rational Rhapsody Developer for C Reactive (instantiated as me) was sent via the port myPort.

For ports with multiplicity greater than one:

```
RiCIS_PORT(me->myPort[2])
```


Port code generation in Java

The following operations are used for working with ports and events in Java:

- ♦ Calling an operation:
 - for port called MyPort and operation called myop:
`getMyPort().myop();`
 - for port called MyPort, operation called myop, and multiplicity greater than 1:
`getMyPortAt(port index).myop()`, for example,
`getMyPortAt(2).myop();`
- ♦ Generating and sending an event via a port:
 - for port called MyPort and event called evt:
`getMyPort().gen(new evt());`
 - for port called MyPort, event called e2, and multiplicity greater than 1:
`getMyPortAt(port index).gen(new e2())`, for example,
`getMyPortAt(2).gen(new e2());`
- ♦ Detecting the input port through which an event has been sent:
 - for port called MyPort:
`isPort(getMyPort())`
 - for port called MyPort, and multiplicity greater than 1: `isPort(getMyPort(port index))`, for example,
`isPort(getMyPort(3))`

Composite types

Rational Rhapsody enables you to create composite types that are modeled using structural features instead of verbatim, language-specific text. In addition, Rational Rhapsody includes classes wherever types are used to increase the maintainability of models: if you change the name of a class, the change is propagated automatically throughout all of the references to it.

To create a composite type:

1. Right-click a package or the Types category, then select **Add New > Type**.
2. Edit the default name for the type.
3. Open the Features window for the new type. The Type window opens.
4. If wanted, specify a stereotype for the type.
5. Specify the kind of data type using the **Kind** list. The possible values are as follows:
 - a. **Enumeration** specifies the new type is an enumerated type. Specify the enumerated values on the **Literals** tab. For more information, see [Creating enumerated types](#).
 - b. **Language** specifies the new type is a language-specific construct. This is the default value. For more information, see [Creating language types](#).
 - c. **Structure** specifies the new type is a structure, which is a data record. For more information, see [Creating structures](#).
 - d. **Typedef** specifies the new type is a typedef. For more information, see [Creating Typedefs](#).
 - e. **Union** specifies the new type is a union, which is an overlay definition of a data record. For more information, see [Creating unions](#).

See the appropriate data type to continue the creation process.

The following table shows the mapping of composite types to the different languages.

Type Kind	Ada	C and C++	Java
Language	As in previous versions	As in previous versions	As in previous versions
Structure	Not supported	struct ¹	N/A
Union	Not supported	union	N/A
Enumeration	Enumeration types	enum	N/A
Typedef	Subtypes (in simple cases) or subtype	typedef	N/A

1. The generated struct is a simple C-style struct that contains only public data members.

Code generation analyzes the types to automatically generate:

- ◆ Dependencies in the code (`#include`)
- ◆ Type descriptions
- ◆ Field descriptions

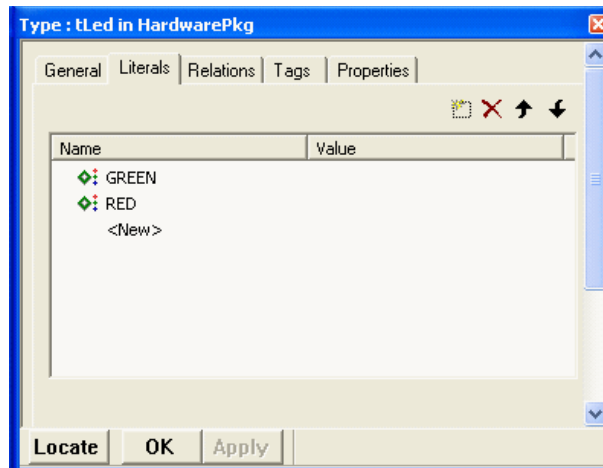
Each field in a structure and union has an attribute annotation.

Creating enumerated types

If you selected **Enumeration** as the **Kind**, continue the creation process as follows:

1. On the **Literals** tab, select the **<New>** line, and then type the name for the enumerated value.
2. Repeat for each value of the enumerated type.

The following figure shows values for an enumerated type.



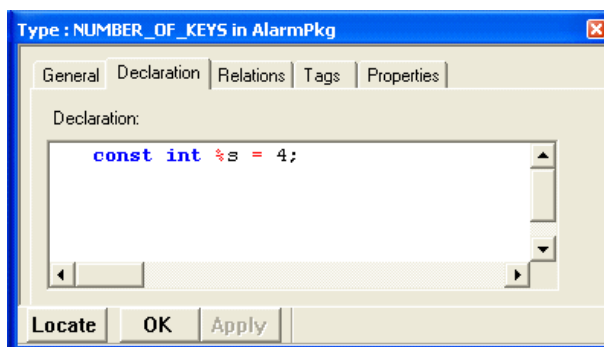
3. Click **OK**.

Creating language types

If you selected **Language** as the **Kind**, continue the creation process as follows:

1. On the **Declaration** tab, type the declaration statement in the **Declaration** text box. Use the expression %s as a placeholder for the type name in the declaration.

The following figure shows an example of a type of kind **Language**.



2. Click **OK**.

Using %s

The Rational Rhapsody code generator substitutes %s in type declarations with the type name. This automates the update of declarations when you rename a type.

To escape the %s characters, type a backslash (\) character before the %s. For example:

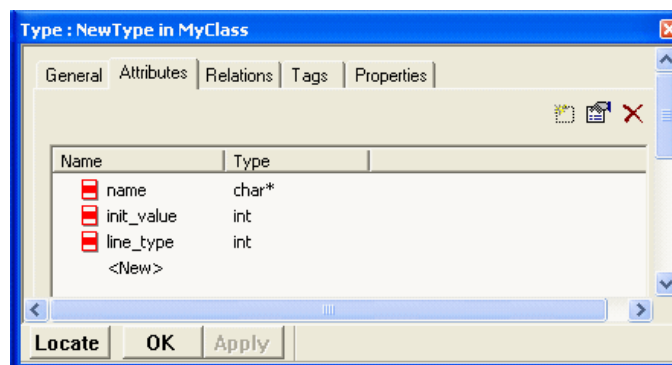
```
#define PRINT printf("\%s\n", myString())
```

Creating structures

If you selected **Structure** as the **Kind**, continue the creation process as follows:

1. On the **Attributes** tab, select the **<New>** line, and then type the name for the member.
2. Use the **Type** list to select the type of the member. Note that the type can be another composite type.
3. Repeat Steps 1 – 2 for each structure member.

The following figure shows an example of a type of kind **Structure**.



4. Click **OK**.

Note

Bit fields (for example, `int a : 1`) are not supported. You can model them using language types. For more information, see [Creating language types](#).

Creating Typedefs

If you selected **Typedef** as the **Kind**, continue the creation process as follows:

1. On the **Details** tab, specify the typedef in the **Basic Type** box, or use the list to select the type. Note that the **Basic Type** cannot be an implicit type.

Note: If you select a type defined within the model, the arrow button next to the **Basic Type** box is available. Click the arrow button to open the Features window for that class.

2. Specify the multiplicity in the **Multiplicity** box. The default value is 1.

Note: If the multiplicity is a value higher than 1, the **Ordered** check box is available. Click this check box if the order of the reference type items is significant.

3. If the typedef is defined as a constant (is read-only, such as the `const` qualifier in C++), enable the **Constant** check box; if the typedef is modifiable, leave the check box disabled (empty).
4. If the typedef is referenced as a reference (such as a pointer (*) or a C++ reference (&), enable the **Reference** check box.

The implementation of the reference is set by the property `<lang>_CG::Type::ReferenceImplementationPattern`. See the definition of this property in the Features window.

5. Click **OK**.

Creating unions

If you selected **Union** as the **Kind**, continue the creation process as follows:

1. On the **Attributes** tab, select the **<New>** line, then type the name for the member.
2. Use the **Type** list to select the type of the member. Note that the type can be another composite type.
3. Click **OK**.

Note

Ada variant record keys and conditions are not supported.

Properties

The following table lists the properties that support composite types.

Subject and Metaclass	Property	Description
CG subject		
Attribute/Type	Implementation	The Implementation property enables you to specify how Rational Rhapsody generates code for a given element (for example, as a simple array, collection, or list). (Default = Default) When this property is set to Default and the multiplicity is bounded (not *) and the type of the attribute is not a class, code is generated without using the container properties (as in previous versions of Rational Rhapsody). Note that Rational Rhapsody generates a single accessor and mutator for an attribute, as opposed to relations, which can have several accessors and mutators. In smart generation mode, a setter is not generated when the attribute is Constant and either: • The attribute is not a Reference. • or The multiplicity of the attribute is 1. • or The CG::Attribute::Implementation property is set to EmbeddedScalar or EmbeddedFixed.
<ContainerType> subject		
<ImplementationType>	Various properties	Contain the keywords \$constant and \$reference to support the Constant and Reference modifiers
<ImplementationType>	FullTypeDefinition	Specifies the typedef implementation template
<lang>_CG subject		
Attribute	MutatorGenerate	Specifies whether mutators are generated for attributes
Attribute/Type	ReferenceImplementationPattern	Specifies how the Reference option is mapped to code
Class/Type	In	Specifies how code is generated when the type is used with an argument that has the modifier In

Subject and Metaclass	Property	Description
	InOut	Specifies how code is generated when the type is used with an argument that has the modifier InOut
	Out	Specifies how code is generated when the type is used with an argument that has the modifier Out
	ReturnType	Specifies how code is generated when the type is used as a return type
	TriggerArgument	Is used for mapping event and triggered operation arguments to code instead of the In, InOut, and Out properties
Type	EnumerationAsTypedef	Specifies whether the generated enum should be wrapped by a typedef. This property is applicable to enumeration types in C and C++.
	StructAsTypedef	Specifies whether the generated enum should be wrapped by a typedef. This property is applicable to structure types in C and C++.
	UnionAsTypedef	Specifies whether the generated union should be wrapped by a typedef. This property is applicable to union types in C and C++.

Language-independent types

Rational Rhapsody enables you to build static models using language-independent, predefined types, with no dependency on the implementation language.

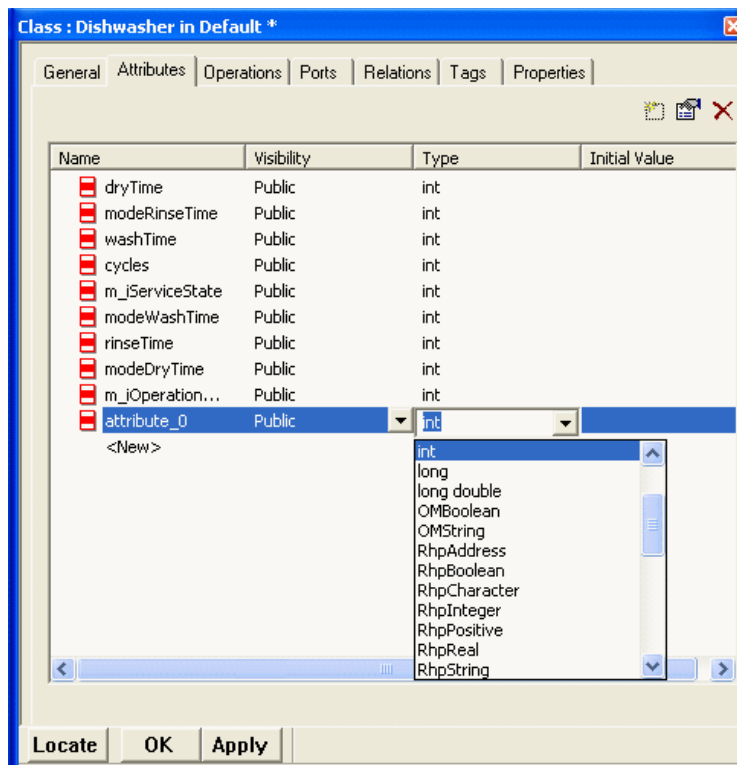
The types are defined in the following files (under <Rational Rhapsody installation path>\Share\<lang>\oxf):

- ◆ Ada, the RiA_Types package
- ◆ C, the RiCTypes.h
- ◆ C++, the rawtypes.h
- ◆ Java, the types are converted during code generation, based on properties defined in the PredefinedTypes package loaded by Rational Rhapsody (under <Rational Rhapsody installation path>\Share\Properties\PredefinedTypes.sbs).

The following table shows the mapping between the predefined types and the language implementation types.

Model Type	Ada	C	C++	Java
RhpInteger	integer	int	int	int
RhpUnlimitedNatural	long_integer	long	long	long
RhpPositive	unsigned	unsigned int	unsigned int	int
RhpPositive	unsigned	unsigned int	unsigned int	int
RhpReal	long_float	double	double	double
RhpCharacter	character	char	char	char
RhpString	string	char*	OMString	String
RhpBoolean	boolean	RiCBoolean	bool	boolean
RhpVoid	Used in procedure declaration only	void	void	void
RhpAddress	address	void*	void*	Object

When you create attributes or operations, these language-independent types are included in the **Types** list.



Changing the type mapping

The file `PredefinedTypes.sbs` contains the set of predefined types included in Rational Rhapsody. This file is opened automatically when you create a new model or open an existing one.

If you previously changed this file, you can merge your changes:

1. Open the model or create a new one.
2. Select **File > Add to Model**. The Add To Model window opens.
3. Navigate to `<Rational Rhapsody installation path>\Share\Properties`.
4. Set the **Files of type** box to **Package (*.sbs)**.
5. Select the **PredefinedTypes.sbs** file.
6. Click **Open**.

7. Because the package already exists in the model, a window opens so you can add the package to the model under a new name, such as `PredefinedTypes_new`.
8. Click **OK**.
9. Change the property `<lang>_CG::Type::LanguageMap` to TBS.
10. Save the modified package.
11. Close the model.
12. Run DiffMerge on your original file and the new one, which is located in the `<project name>_rpy` directory.
13. Add your changes to the `.sbs` file located in `<Rational Rhapsody installation path>\Share\Properties`.

Note

The following behavior and restrictions apply to the language-dependent types.

- ♦ In Rational Rhapsody in J, language-independent types are supported only as modeling constructs. You cannot use them in actions or operation bodies.
- ♦ This feature does not apply to COM and CORBA.

Changing the order of types in the generated code

Types are generated in code in the order in which they appear in the browser. This can be a problem if one type depends on another that is defined later.

For example, you can define a type `FirstType` as:

```
typedef SecondType %s
```

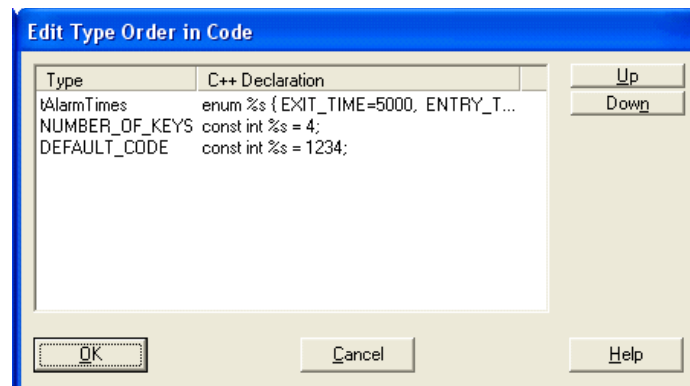
Next, define a type `SecondType` as:

```
typedef int %s
```

These two types defined in this order would result in a compilation error. To avoid this kind of error, you can control the order in which types are generated using the Edit Type Order in Code window, which is accessible via the menu for the Types category for an individual package or a class.

To edit the order of types:

1. Right-click the **Types** category (or a package or a class) and select **Edit Type Order**. The Edit Type Order in Code window opens.



2. Select the type you want to move.
3. Click **Up** to generate the type earlier or **Down** to generate it later.
4. Click **OK**.

Using fixed-point variables

For target systems that do not include floating-point capabilities, Rational Rhapsody in C provides an option to use fixed-point variables.

This is done by scaling integer variables so that they can represent non-integral values. Rational Rhapsody uses the 2 *factorial* approach to achieve this. For example, setting the bit that usually represents 2^0 to represent 2^{-3} (.125).

For each such variable, the user specifies the word-size and the precision of the variable. The specific steps involved are described in [Defining fixed-point variables](#).

Defining fixed-point variables

The elements required for defining fixed-point variables are included in a profile called *FixedPoint*. This profile contains:

- ◆ Predefined types representing 8, 16, and 32-bit fixed-point variables: *FXP_8Bit_T*, *FXP_16Bit_T*, *FXP_32Bit_T*. (These are the only types that can be used with fixed-point operations.)
- ◆ A “new term” stereotype, applicable to attributes, called *FixedPointVar*, with a tag called *FXP_Shift* which is used to define the scale of the fixed-point variable.

The word-size is determined by the type chosen, while the shift to use is determined by the value entered for the tag *FXP_Shift*.

The profile uses a file called *FixedPoint.h*, which contains:

- ◆ Typedefs representing the predefined fixed-point variable types
- ◆ Macros that are used for carrying out operations on fixed-point variables.

The file is “included” into the generated code where fixed-point variables are generated.

To define a fixed-point variable:

1. Add the *FixedPoint* profile to your project as a reference.

Note: The *FixedPoint* profile is added with Rational Rhapsody only if you selected the Automotive add-on during installation.

2. In the browser, right-click the element that will contain the fixed-point variables and select **Add New > General Elements > FixedPointVar**.
3. Name the new variable.

4. Open the Features window for the new variable, and for *Type* select one of the fixed-point variable types (*FXP_8Bit_T*, *FXP_16Bit_T*, or *FXP_32Bit_T*). (If you do not see these types in the list, click **Select** and locate the relevant type in the tree that is displayed.)
5. Set the shift to use by providing a value for the tag *FXP_Shift* (default value is 4). (The variable that was created already has the *Fixed-Point* stereotype applied to it.)

Operations permitted for fixed-point variables

The following operations can be performed on fixed-point variables:

- ♦ Arithmetic: addition, subtraction, multiplication, division
- ♦ Assignment (=)
- ♦ Relational operators (<, >, <=, >=, ==, !=)

To carry out the operations, you use the relevant macros that are contained in *FixedPoint.h*. For example to add fixed-point variables, use the macro *FXP_ASSIGN_SUM*. (Note that some of the macros in this file are macros that are called by the operation macros. These macros should not be called directly.)

Restrictions on use of fixed-point variables

Keep the following points in mind when working with fixed-point variables:

- ♦ The supported operations can only be performed on fixed-point variables, and not on the result of fixed-point calculations. For example, these are not permitted:
`FXP_ASSIGN_SUM(FXP_ASSIGN_SUM(varA,varB),varC).`
- ♦ Operations can be performed on fixed-point variables only. If you try to use one of the operations with a combination of fixed-point and ordinary variables, compilation errors will result.
- ♦ The shift specified can range from 0 to (word size - 1). Rational Rhapsody does not check that the shift you entered for the variable is within this range.
- ♦ When calling a function that takes a fixed-point variable as an argument, make sure that the variable provided to the function has the same fixed-point characteristics (word size and shift) as the defined argument.
- ♦ When calling a function that returns a fixed-point variable, make sure that the return value is being assigned to a variable that has the same fixed-point characteristics (word size and shift) as the defined return type.
- ♦ Programmers must take into account that operations on fixed-point variables can result in an arithmetic overflow.
- ♦ Programmers must take into account that operations on fixed-point variables can result in a loss of precision.

Fixed-point conversion macros

Rational Rhapsody provides the following macros for converting to/from fixed-point variables:

- ◆ `FXP2INT(FPvalue, FPshift)` - Converts a fixed-point variable to an integer
- ◆ `FXP2DOUBLE(FPvalue, FPshift)` - Converts a fixed-point variable to a double
- ◆ `DOUBLE2FXP(Dvalue, FPshift)` - Converts a double to a fixed-point variable

These macros can be used in conjunction with the macros that require fixed-point variables as arguments, for example:

```
FXP_ASSIGN_EXT(myFixedPointVar, FXP_16Bit_T, 4, DOUBLE2FXP(3.5, 1), 1);
```

The arguments provided represent:

- ◆ Fixed-point variable to be initialized
- ◆ FXP type of the variable to be initialized
- ◆ FXP shift of the variable to be initialized
- ◆ Initializing value in integer representation
- ◆ Shift of the integer initializing number

Java enums

Rational Rhapsody allows use to include Java enums (introduced in Java 5.0) in your models.

Adding a Java enum to a model

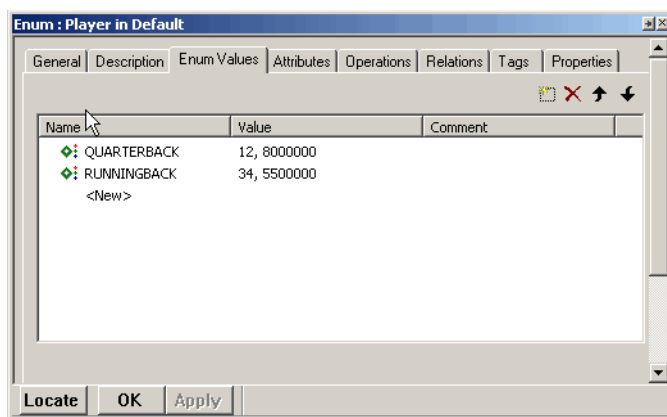
To add a Java enum to your model, right-click the package to which you would like to add the enum and select **Add New > Enum**.

Enums are displayed as their own category in the Rational Rhapsody browser.

Defining constants for a Java enum

To define constants for an enum:

1. Double-click the relevant enum in the browser to open its Features window.
2. Select the **Enum Values** tab.



3. Click **<New>** in the list.
4. Click the Name column to change the default name assigned by Rational Rhapsody.
5. Click the Value column to define the argument values that will be used to instantiate this instance of the enum (if you are providing more than one argument, separate them with commas).
6. Optionally, add comments for the constants you have defined.
7. Click **OK**.

After they have been defined, enum constants appear underneath the relevant enum in the browser.

Note

Rational Rhapsody does not support the definition of anonymous classes that extend enum constants.

Adding Java enums to an object model diagram

To add a Java enum to an object model diagram, drag the enum from the browser to the diagram.

Code generation

Rational Rhapsody generates Java code for the enums you have defined.

The reverse engineering feature does not support Java enums.

Creating Java enums with the Rational Rhapsody API

The following lines of code will add a new enum to the selected package, and define two constants for the new enum:

```
Dim p As RPPackage
Dim c As RPClass
Dim a1 As RPAttribute
Dim a2 As RPAttribute

Set p = getSelectedElement
Set c = p.addNewAggr("Enum", "SampleEnum")
Set a1 = c.addNewAggr("EnumValue", "FIRST_CONSTANT")
Set a2 = c.addNewAggr("EnumValue", "SECOND_CONSTANT")
```

Template classes and generic classes

Rational Rhapsody allows you to include generic design elements in your models. Specifically, you can:

- ◆ Create and use template classes (C++)
- ◆ Create and use generic classes (Java)
- ◆ Create template functions (C++)
- ◆ Create generic methods (Java)

The terminology used for these concepts differs slightly between C++ and Java. In this section, we will use the UML terms *template class* and *template operation* to represent the generic elements in both C++ and Java.

In general, the procedures described in this section apply to both C++ and Java. Where there are language-dependent differences, these differences are noted.

Creating a template class

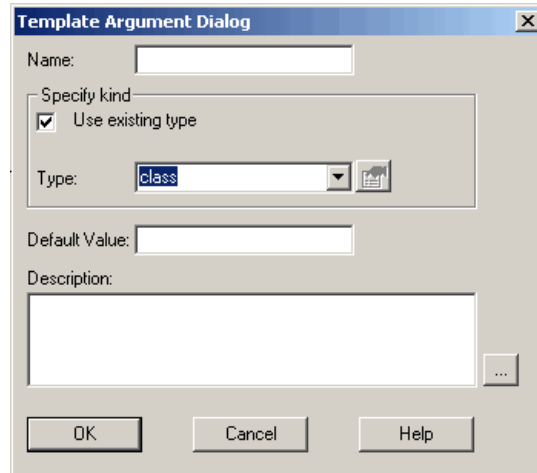
You can use a class to create a template class. In addition, some template parameters can be specified as specific types and a specialized function to create a specialization or new class/function with content that is unrelated to the original template.

Note that you can use the DiffMerge tool to locate and merge template information.

To create a class template:

1. Double-click the class in the Rational Rhapsody browser to open its Features window.
2. On the **General** tab, in the **Class Type** area, select the **Template** radio button. Notice that the **Template Parameters** tab displays.
3. On the **Template Parameters** tab, click <New>.
4. Type a name to replace the default name that Rational Rhapsody creates as <class_n>. For guidelines for these names, see [Template limitations](#).

5. Accept the default type or select another one from the **Kind** list.
6. To add arguments for the template, click the Invoke Features window button  to open the Template Argument window. Note the following for the Template Argument window:
 - a. If you select the **Use existing type** check box, you can change the type and enter a description. In C++, you can also provide a default value for the template argument.



- b. If you clear the **Use existing type** check box, you can enter code that further refines the argument type, for example a pointer to a type or an array of a certain type. When entering code in the **C++[Java] Declaration** box, you can also see other arguments that have been defined.
 - c. Click **OK** to close the Template Argument window and return to the **Template Parameters** tab.
7. Add more templates as needed by clicking **<New>** on the **Template Parameters** tab.
8. To determine the argument order on the **Template Parameters** tab, use the Move Item Up  and Move Item Down  buttons.
9. If there is a primary template that you want to use, select it in the **Primary Template** drop-down list box. This box contains templates for which this class is a specialization. Its parameters to be instantiated appear in the box below the **Primary Template** drop-down list box.

You can define specialization parameters only if you select a template as a primary class.

Note: When you try to delete a template that has specialization, Rational Rhapsody warns you that the template has references. If you do delete the template, such specialization will generate an error when you check a model.

10. Click **OK**.

The template is listed in the browser in the **Classes** category.

Once you have created the template class, you can begin using it directly in your code.

You can create templates in other situations. For example, you can:

- ◆ Re-use any type defined for a template parameter as a type within the template.
- ◆ Use the template class as a generalization, as described in [Using template classes as generalizations](#).
- ◆ Create an operation template, as described in [Creating an operations template](#).
- ◆ Create a function template, as described in [Creating a functions template](#).

See also [Instantiating a template class](#).

Using template classes as generalizations

To use a template class as a generalization:

1. Create a class in an OMD, or in the Rational Rhapsody browser.
2. Create the generalization by adding a super class in the browser or by drawing a generalization connector from the new class to the template class.
3. Open the Features window of the generalization by using the context menu of the super class in the browser or the generalization connector in the OMD.
4. On the **Template Instantiation** tab of the Features window, provide a value for each of the arguments listed by selecting an item from the **Value** list or entering a new value.

Creating an operations template

To create an operation template:

1. Create an operation in a class.
2. Open the Features window for the operation.
3. On the **General** tab, select the **Template** check box. Notice that the **Template Parameters** tab displays.
4. Set your template parameters for your operation on the **Template Parameters** tab. For detailed instructions, see [Creating a template class](#).

Once you have created the template operation you can begin using it in your code.

Creating a functions template

To create a functions template:

1. Create a function and open its Features window.
2. On the **General** tab, select the **Template** check box. Notice that the **Template Parameters** tab displays.
3. Set your template parameters for your function on the **Template Parameters** tab. For detailed instructions, see [Creating a template class](#).

Once you have created the template operation you can begin using it in your code.

Instantiating a template class

To instantiate a template class:

1. Create a class in an OMD, or in the Rational Rhapsody browser.
2. Open the Features window for the class.
3. On the **General** tab, in the **Class Type** area, select the **Instantiation** radio button. Notice that the **Template Instantiation** tab displays.
4. On the **Template Instantiation** tab, select a template from the **Template Name** drop-down list box.
5. To view/modify the parameters for a template, double-click the template name or click the Invoke Features window button to open the Template Instantiation Argument window. Click **OK** to return to the **Template Instantiation** tab.

When code is generated, the template instantiation is represented by a typedef in C++ and by a class in Java.

Code generation and templates

The creation of templates and specializations are supported in code generation. If both the template and its specialization are in the same package, they are generated into the same file. In the file, the template is generated before its specialization to ensure that the code runs as expected. A check is added to warn that the template and template specialization are in different packages.

Note

If a nested class or attribute is marked as a template parameter, it is not generated.

Template limitations

The following limitations apply for templates:

- ♦ If there is a template parameter named “T” in an operation/function, the user cannot assign a class named “T” to the owner of the operation/function.
- ♦ If there are more than one operation/function with a template parameter of the same name under the same owner, renaming the created nested class renames all of the parameters with this name.
- ♦ For templates in a Java project, these are additional limitations:
 - Wildcards are not supported.
 - Bounded wildcards are not supported.
 - Generic methods are not supported.

Eclipse platform integration

This subject describes the **Rational Rhapsody Platform Integration**, which lets software developers work on a Rational Rhapsody project within the Eclipse platform. This integration is currently available only for C, C++, or Java development in a Windows environment only and not on Linux.

If you want to work in the Rational Rhapsody interface and use some Eclipse features, you can use the **Workflow Integration**, which is the other Rational Rhapsody plug-in implementation. In this integration, software developers work in the Rational Rhapsody product and open Rational Rhapsody menu commands to use some Eclipse features. You can also navigate between the two environments. This integration can be used for C, C++, and Java development in either Windows or Linux environments. Both Eclipse and Rational Rhapsody must be open when you are using this integration. For information about this implementation, see [Rational Rhapsody projects](#).

Note

See the Rational Rhapsody installation instructions for Eclipse-specific installation and set-up information.

Platform integration prerequisites

The following software needs to be used to create a fully functioning Eclipse platform integration with Rational Rhapsody:

- ◆ Eclipse Ganymede
- ◆ CDT plug-in for C and C++ application development
- ◆ JDT plug-in for Java development
- ◆ Compilers required for the development language or languages you are using
- ◆ Rational Rhapsody Developer edition (multi-language)

The Platform Integration of Rational Rhapsody for Eclipse requires a multi-language Rational Rhapsody license.

Note

The stand-alone version of Rational Rhapsody and the Rational Rhapsody Platform Integration within Eclipse both use the same repository so that you can switch between the two interfaces if you want.

Confirming your Rational Rhapsody Platform Integration within Eclipse

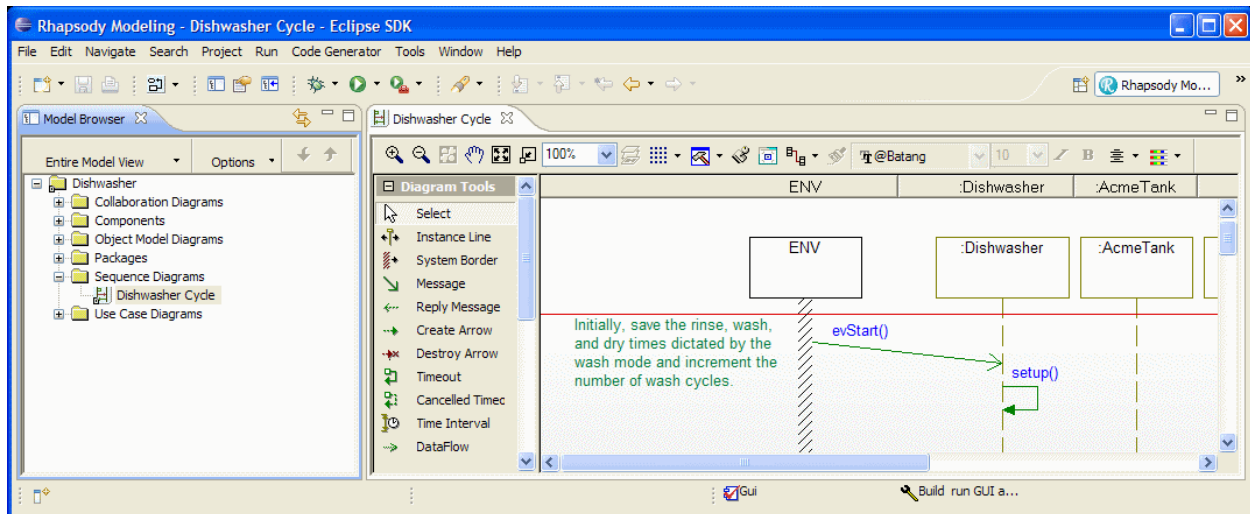
To confirm that you have the Rational Rhapsody Platform Integration within Eclipse:

1. In Eclipse, choose **Help > About Eclipse SDK** to open the About Eclipse SDK box.
2. You should see a Rational Rhapsody icon on the About Eclipse SDK box. If this icon does not appear, you are not set up for the Rational Rhapsody Platform Integration. See the Eclipse set-up instructions in the Rational Rhapsody installation instructions to set up for this integration.

For descriptions of the areas in the Rational Rhapsody interface, see [Rational Rhapsody Platform Integration within Eclipse](#).

Rational Rhapsody Platform Integration within Eclipse

The standard Rational Rhapsody interface elements displayed in Eclipse have the same features as in the stand-alone version except that the icons associated with a specific window are displayed at the top of the window.



The Rational Rhapsody Platform Integration within Eclipse adds two Rational Rhapsody perspectives on tabs in the upper right corner of the Eclipse IDE:

- ◆ [Rational Rhapsody modeling perspective](#)
- ◆ [Rational Rhapsody Debug perspective](#)

Rational Rhapsody perspectives in Eclipse

When you create a Rational Rhapsody project in Eclipse, the *Rhapsody Modeling* perspective is automatically displayed and is set as the open Eclipse perspective.

If you have the tabs for perspectives displayed in the upper-right corner, you can click a tab to go to another perspective.

Opening perspectives

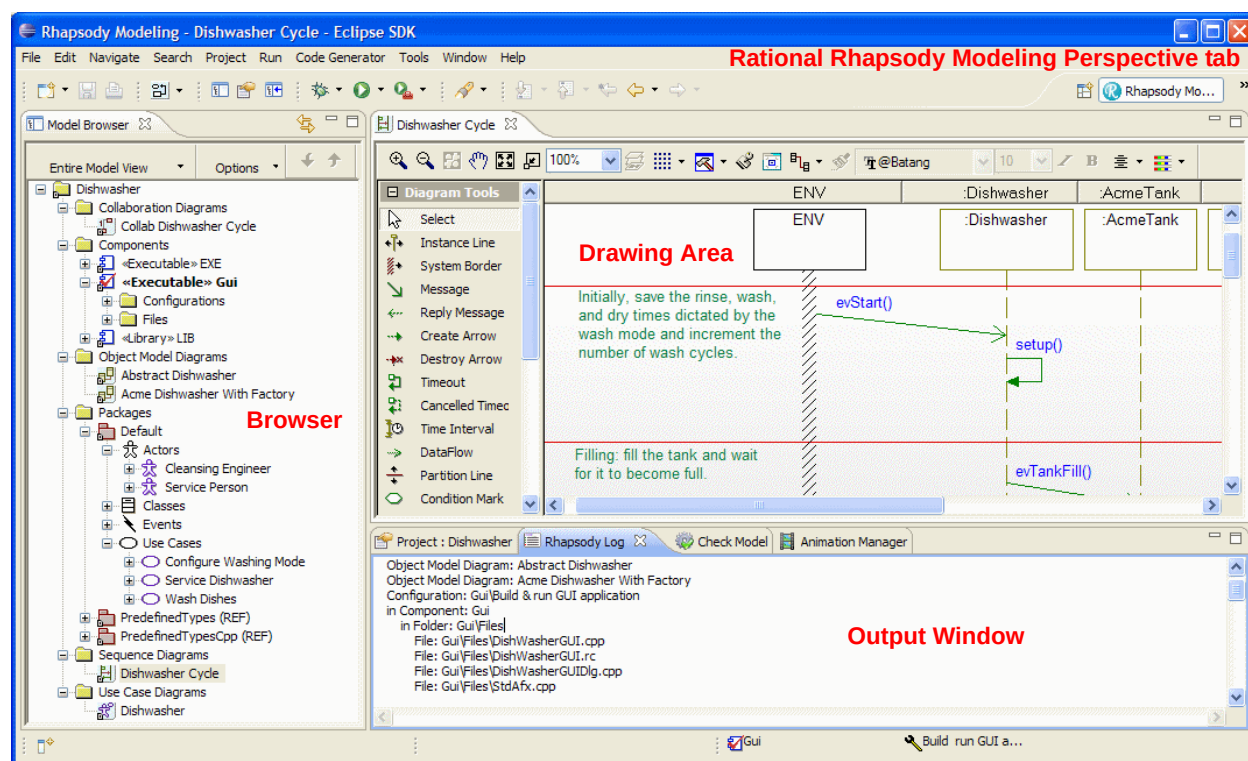
To open a different perspective manually:

1. Choose **Window > Open Perspective > Other**.
2. On the Open Perspective window, select another perspective, such as **Rhapsody Debug**, and click **OK**.

Rational Rhapsody modeling perspective

The Rational Rhapsody Modeling Perspective displays the main Rational Rhapsody interface components, as illustrated in the following figure:

- ◆ Browser (Model Browser tab in Eclipse)
- ◆ Diagram Drawing Area
- ◆ Output window and Features window

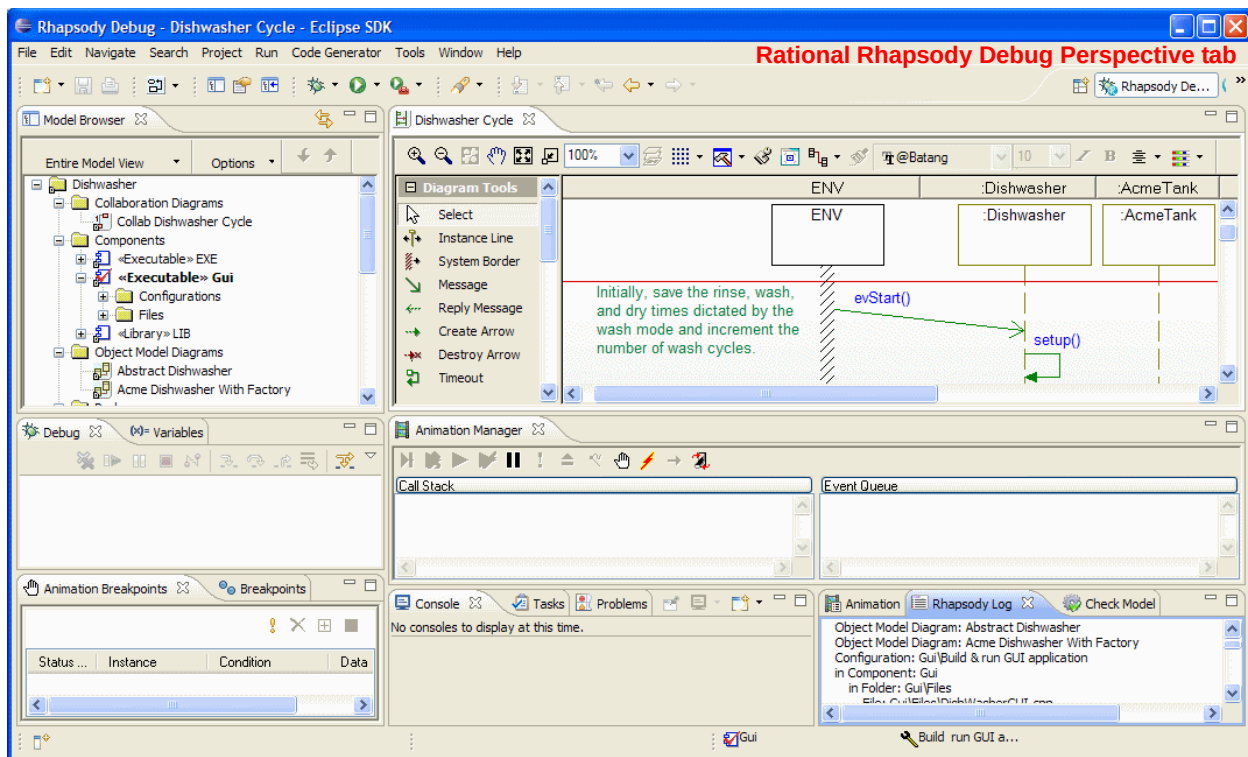


The Rational Rhapsody menu commands and drawing capabilities have been added to the Eclipse code editing, interface customization, and other capabilities.

Rational Rhapsody Debug perspective

The Rational Rhapsody Debug perspective displays a number of windows. When you open the Rational Rhapsody Debug perspective, the following windows might open in the Eclipse IDE:

- ◆ Debug
- ◆ Variables
- ◆ Animation Breakpoints
- ◆ Animation Manager
- ◆ Console
- ◆ Tasks
- ◆ Problems
- ◆ Animation
- ◆ Rhapsody Log
- ◆ Check Model



Developers can then use the Eclipse code level debugger and Rational Rhapsody design level debugging with animation and breakpoints for a thorough and efficient debugging strategy.

Rational Rhapsody Eclipse support for add-on tools

The Eclipse Platform Integration of Rational Rhapsody supports for the add-on configuration management tools and XMI file import and export capabilities.

Configuration management tools

The Rational Rhapsody Eclipse Platform Integration supports the common open source configuration management (CM) tools, Concurrent Versions System (CVS) and Subversion (SVN), in addition to these CM tools:

- ◆ IBM Rational ClearCase
- ◆ IBM Rational Synergy

XMI import and export

You can use the Rational Rhapsody XMI import and export facility with Eclipse projects. For more information, see [Using XMI in Rational Rhapsody development](#).

Eclipse projects

The Eclipse plug-in platform integration provides Rational Rhapsody and Eclipse features for software developers.

Creating a new Rational Rhapsody project within Eclipse

To create a new Rational Rhapsody project within Eclipse:

1. In Eclipse, choose **File > New > Project**.
2. In the New Project window, select **Rhapsody Project** and click **Next**.
3. Type a name for your Rational Rhapsody project and select the language (C, C++, or Java) from the list. In addition, you can type the name for your first object model diagram.
4. If you want to change the location of where you want to store your project, clear the **Use default location** check box and then use the **Browse** button to navigate to another location.
5. Click **Finish** on the New Rhapsody Project window when you are done defining your new Rational Rhapsody project.
6. If the directory for your new project has not yet been created, click **Yes** when you are asked if you want to create it.

Rational Rhapsody creates a new project in the work area you specified and opens the new project.

Opening a Rational Rhapsody project in Eclipse

To open an existing Rational Rhapsody project in Eclipse:

1. In Eclipse, choose **File > Import** to open the Import window.
2. Expand the **Rhapsody** folder, select **Rhapsody Project**, and then click **Next**.
3. On the Import window, click the **Browse** button to open the Browse for Folder window.
4. In the Browse For Folder window, select the folder that contains the Rational Rhapsody project that you want to import, and then click **OK**.
5. On the Import window, select the project that you want to open and click **Finish** to open the project in Eclipse.

Adding new elements

To add new elements to your project including diagrams, packages, and files:

1. With Eclipse Model Browser displayed, right-click an element for which you want to add an element.
2. Select **Add New** > (element type) from the menu depending on what type of element you selected. (See also [Filtering out file types](#).)

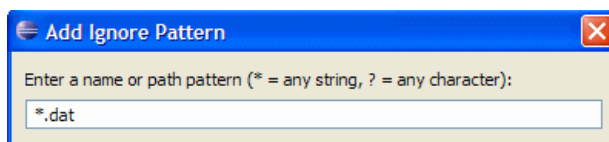
The following figures show how you can add an element to your project. While this example shows adding a package. The same method is true for other elements, such as diagrams, files, actors, operations, and so on.

If you want to create a new package, right-click the project for a new main package or an existing package to create a sub package and select **Add New** > **Package**. You can add other elements (such an object, dependence, and a class) from the menu.

Filtering out file types

To prevent Eclipse from presenting unwanted Rational Rhapsody project file types when adding files:

1. In Eclipse, choose **Window** > **Preferences**.
2. In the Preferences window, expand the **Team** section of the tree structure.
3. Select **Ignored Resources** to display the list of file extensions that can be set to be ignored.
4. If the list does not contain the file types you want to filter out, click the **Add Pattern** button.
5. On the Add Ignore Pattern window, type the file extension using the format shown in the following figure and click **OK** to add the extension to **Ignored Resources**.



6. Using this method, you can enter the Rational Rhapsody *.dat, *.ehl, *.rpx, and *.vba file extensions to **Ignored Resources**.
7. Click **OK**.

Exporting Eclipse source code to a Rational Rhapsody project

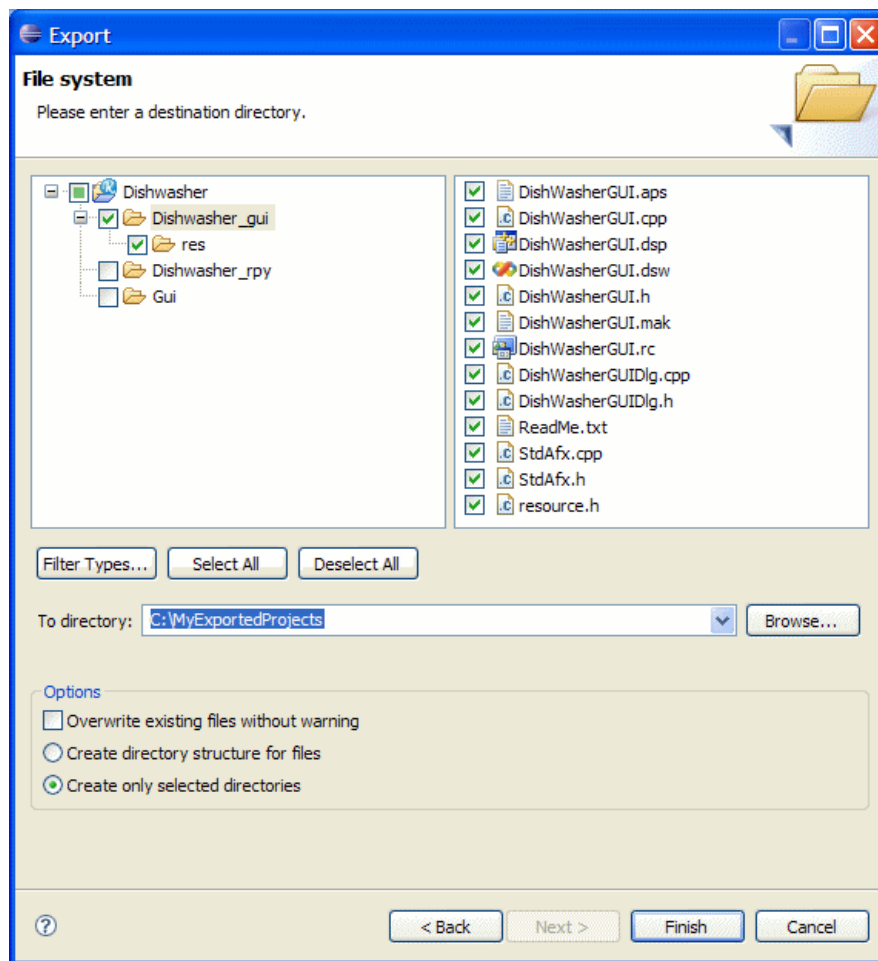
To export an Eclipse source code project from Eclipse to Rational Rhapsody:

1. With the model open in Eclipse, choose **File > Export** to open the Export window.
2. Expand the **General** folder, select **File System**, and then click **Next**.
3. In the File System view of the Export window, in the list in the left box:

- ♦ Select the top-level project to select everything in the project, or
- ♦ Select a subfolder folder to select only those items in that subfolder.

Note: You can select specific files to export from the list on the right. You can click the **Filter Types** button to select specific file extensions to be exported.

4. Use the **Browse** button to navigate to a location directory for the exported files.



5. Select any of the **Options** that apply to this operation.
6. Click **Finish** to complete the export operation.

Note: If any problems were encountered during the operation, an Export Problems message box displays. Click the **Details** button for more information.

7. To see your exported files, go to the location directory for your exported files.

Importing Rational Rhapsody units

To import Rational Rhapsody units into Eclipse:

1. In Eclipse, set the project for which you want to import units as the *active configuration*. Right-click the project and select **Set as Active Project**.
2. Choose **File > Import** to open the Import window.
3. Expand the Rhapsody folder, select **Rhapsody Unit**, and then click **Next**.
4. In the Add to Model window, you can select a single file type (unit type) that you want to import by using the **Files of Type** list or you can select the Rational Rhapsody project (.rpy) to select all of the units in the project:
5. Click **Open**.
6. In the Add To Model From Another Project window, select the units you want and make your applicable selections to the **Add Subunits** and/or **Add Dependents** check boxes, and **As unit** or **As reference** radio buttons.
7. Click **OK**.
8. After the import is completed, a window displays the status of the import. Click **Finish**.

Importing source code (reverse engineering)

To import source code:

1. In Eclipse, set the project and the component as the active configuration.
 - a. Right-click the project and select **Set as Active Project**.
 - b. Right-click the component and select **Set as Active Component**.
2. In your Eclipse project, choose **File > Import** to open the Import window.
3. Expand the Rhapsody folder, select **Source Code**, and then click **Next**.

4. When the message window opens, confirm that you want to launch the Rational Rhapsody Reverse Engineering interface, click **Finish**.
5. When asked to confirm that you want the reverse engineered code to be saved to the active component and configuration, click **Continue**.
6. On the Reverse Engineering window, click **Add Files** or **Add Folder** to add items to be reverse engineered.
7. After selecting the items to reverse engineer, click **Start**.

Note: You have a choice of a flat view or a tree view for the selected items. To toggle between the views, click the Flat View button  or the Tree View button .

8. Confirm that you want to continue with the reverse engineering process, click **Yes**.
9. Click **Finish**. The reverse engineering messages display in the **Rhapsody Log** window.

For more information about reverse engineering, see [Reverse engineering](#).

Search and replace in models

The Rational Rhapsody search facility is available to use in Eclipse for these operations:

- ◆ Perform standard search-and-replace operations
- ◆ Search for the following types:
 - unresolved elements in a model
 - unloaded elements in a model
 - units in the model
 - both unresolved elements and unresolved units
- ◆ Work with the search results

For more detailed instructions for the Rational Rhapsody Search and Replace facility, see [Search and replace facility](#).

Accessing the Rational Rhapsody search facility in Eclipse

To access Rational Rhapsody search in Eclipse:

1. With your Rational Rhapsody model open in Eclipse, choose **Search > Search** to open the Search window. (You can also right-click an element in the model browser and select **Search**.)
2. For the Rational Rhapsody search facility, click the **Rhapsody** tab. (You can also right-click an element on the model browser and select **Search inside**.)

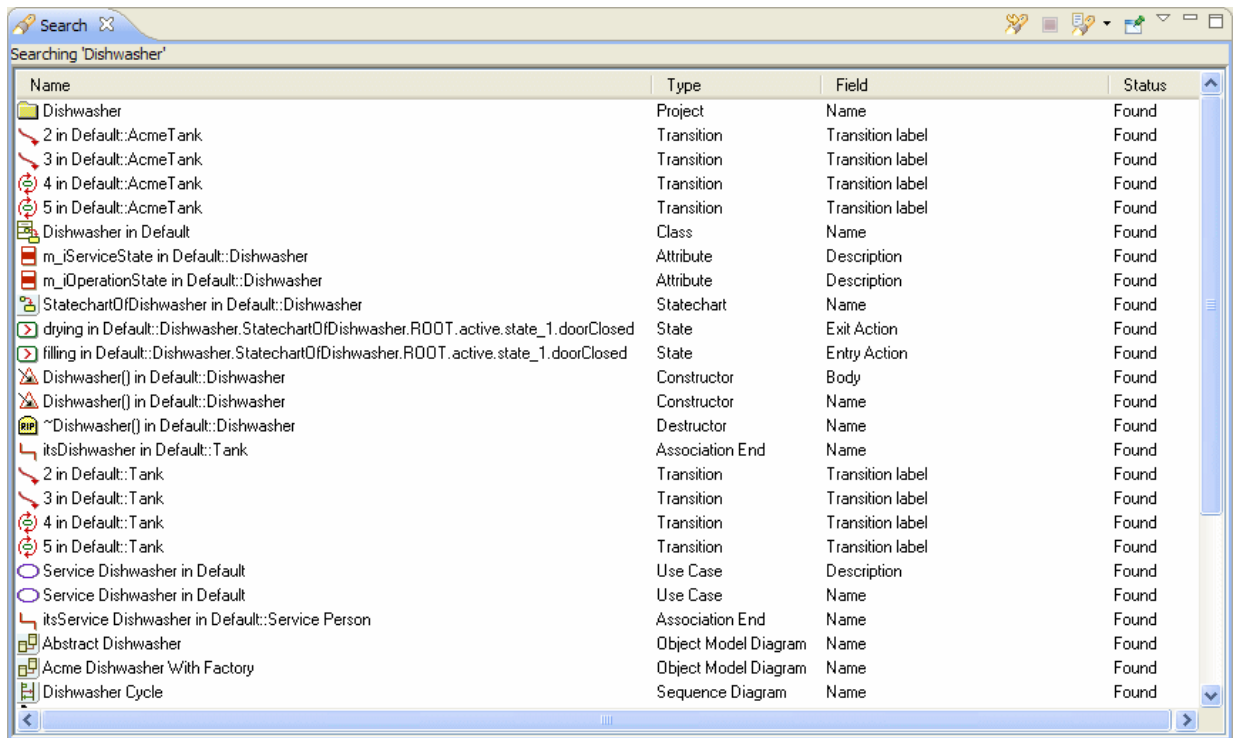
Customize the search criteria

In the Rational Rhapsody search facility you can use any of the standard search facilities. You can also customize your search using the following buttons.

- ♦ **Exact string** allows a non-regular expression search. When selected, the search looks for the string entered into the search field (such as `char*`).
- ♦ **Wildcard** allows wildcard characters in the search field such as `*` and produces results during the search operation that include additional characters. For example, the search `*dishwasher` matches class `dishwasher` and attribute `itsdishwasher`.
- ♦ **Regular Expression** allows the use of Unix-style regular expressions. For example, `itsdishwasher` can be located using the search term `[s]dishwasher`.

Search results display

The search results display in the Search tab of the Eclipse output window.



The screenshot shows the Eclipse Search window with the search term 'Dishwasher'. The results are displayed in a table with four columns: Name, Type, Field, and Status. The results list various elements found in the project, including classes, attributes, statecharts, states, transitions, constructors, destructors, association ends, use cases, and diagrams.

Name	Type	Field	Status
Dishwasher	Project	Name	Found
2 in Default::AcmeTank	Transition	Transition label	Found
3 in Default::AcmeTank	Transition	Transition label	Found
4 in Default::AcmeTank	Transition	Transition label	Found
5 in Default::AcmeTank	Transition	Transition label	Found
Dishwasher in Default	Class	Name	Found
m_iServiceState in Default::Dishwasher	Attribute	Description	Found
m_iOperationState in Default::Dishwasher	Attribute	Description	Found
StatechartOfDishwasher in Default::Dishwasher	Statechart	Name	Found
drying in Default::Dishwasher.StatechartOfDishwasher.ROOT.active.state_1.doorClosed	State	Exit Action	Found
filling in Default::Dishwasher.StatechartOfDishwasher.ROOT.active.state_1.doorClosed	State	Entry Action	Found
Dishwasher() in Default::Dishwasher	Constructor	Body	Found
Dishwasher() in Default::Dishwasher	Constructor	Name	Found
~Dishwasher() in Default::Dishwasher	Destructor	Name	Found
itsDishwasher in Default::Tank	Association End	Name	Found
2 in Default::Tank	Transition	Transition label	Found
3 in Default::Tank	Transition	Transition label	Found
4 in Default::Tank	Transition	Transition label	Found
5 in Default::Tank	Transition	Transition label	Found
Service Dishwasher in Default	Use Case	Description	Found
Service Dishwasher in Default	Use Case	Name	Found
itsService Dishwasher in Default::Service Person	Association End	Name	Found
Abstract Dishwasher	Object Model Diagram	Name	Found
Acme Dishwasher With Factory	Object Model Diagram	Name	Found
Dishwasher Cycle	Sequence Diagram	Name	Found

Working with search results

After locating elements using the Search facility, you can perform the following operations in the Search window:

- ◆ Sort items
- ◆ Check the references for each item
- ◆ Delete
- ◆ Load

To sort the items in the list, click the heading for the column to sort according to information in that column.

To work with an item located in the search:

1. Double-click an item in the list to open the its Features window and highlight its location on the model browser.
2. Make any required changes from these entry points.

Generate and edit code

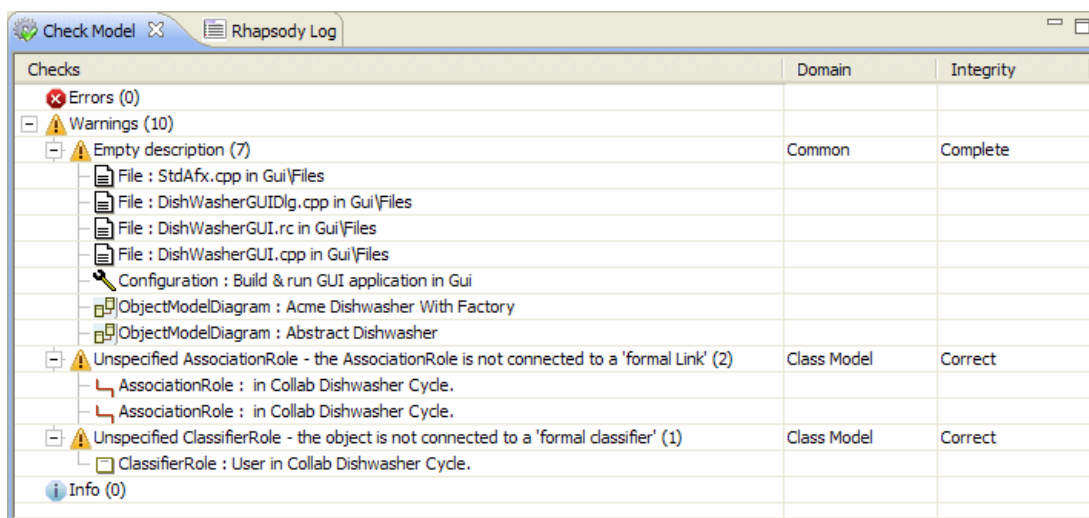
You can check your model, generate code, and edit the resulting code using Eclipse facilities. For detailed instructions describing Rational Rhapsody code generation, see [Basic code generation concepts](#).

Checking the model

To launch the check model process, use one of these methods:

- ♦ Choose **Tools > Check Model** and select one of these options:
 - **<name of active configuration>**
 - **Selected Elements**
 - **Configure**
- ♦ Right-click the active configuration and select **Check**.

The results display in the Check Model tab of the Output window.



As with search results, you can double-click an item on the Check Model tab to open the Features window for the item and highlight it in the model browser.

Generate code

Rational Rhapsody uses an Eclipse IDE project for code generation. Before you can generate code, you must perform the following tasks:

- ♦ Create an Eclipse IDE project
- ♦ Associate the Rational Rhapsody Eclipse configuration with the Eclipse IDE project

Creating an Eclipse IDE project

To create an IDE project to use for Rational Rhapsody code generation:

1. In the model browser, right-click an Eclipse configuration and select **Create IDE Project**.
2. If you have an existing Eclipse project in your work area, that project is listed in the **Existing Project** list. However, you need to create a special Eclipse IDE project, so you should select the **New Project** radio button.
3. Click **Finish**.
4. In the New Project window, select the project type based on your environment.
5. Click **Next**.
6. Enter a project name on the Project window and click **Next**.
7. If necessary, make a configuration selection on the Select Configurations window, and then click **Finish**.
8. If your project type selection is different from your current perspective, an Open Associated Perspective window opens to give you an opportunity to switch perspectives; click **Yes**. If you want to keep using your currently active perspective, click **No**.

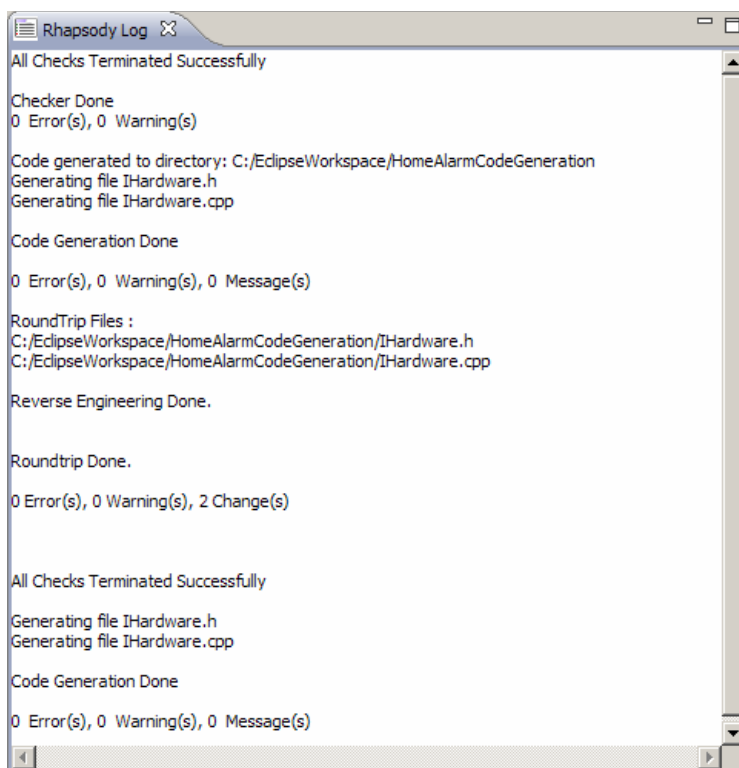
Generating code

After setting up the IDE project to receive the generated code, choose **Code Generator > Generate** and one of these options:

- ♦ **<Active Configuration>** (In the following figure, the active configuration is called GUI.)
- ♦ **Selected classes**
- ♦ **<Active Configuration> with Dependencies**
- ♦ **Entire Project**

The system generates the requested code and displays messages in a log file.

Note: To show a complete log report, the following figure shows a very short report. Typically, especially for the first code generation, there might be more messages for each group (for example, the listing of code generated might be longer).



Selecting Dynamic Model-Code Associativity

Rational Rhapsody lets you work in the model or code and maintain synchronization between each so that changes in one are reflected in the other automatically. This is called Dynamic Model-Code Associativity (DMCA).

To select the DMCA option you want to use:

1. Choose **Code Generator > Dynamic Model Code Associativity**.
2. From the submenu, select one of these commands:
 - ♦ **Bidirectional** changes made to the code or model are synchronized with the other.
 - ♦ **Roundtrip** changes made in the code are automatically synchronized with the model.
 - ♦ **Code Generation** changes made in the model are updated in the code automatically.
 - ♦ **None** turns off DMCA.

Edit code

You can edit your Rational Rhapsody code using the Eclipse editor.

Launching the Eclipse code editor from the browser or diagram

To display generated code for a specific element in the code editor:

1. In the model browser, right-click an item and select **Edit Code** to launch the corresponding source code in the Eclipse code editor.
2. On the Eclipse code editor, highlight any items of interest and right-click to display the editing menu.

Editing code from a diagram element

To launch the Eclipse code editor from a diagram element:

1. Open the diagram.
2. Right-click an item in the diagram and select **Edit Code**.
3. The Eclipse code editor opens.

Locating an element in the browser from the editor

If you are editing code and need to see an item in the project, usually in the model browser:

1. Highlight the item in the code.
2. Right-click and select **Show in** and then select one of the options. The option listed on this submenu are controlled by the type of project displayed. For example, this option list is for a C++ project:
 - ♦ Model Browser
 - ♦ C/C++Projects
 - ♦ Outline
 - ♦ Navigator

Viewing code associated with a model element

To see the code automatically for a selected model element:

1. Click the Link with Editor button  located at the top of the Eclipse model browser.
2. Click an item in the model browser and notice that the code for the item is immediately displayed and highlighted in the Eclipse code editor.

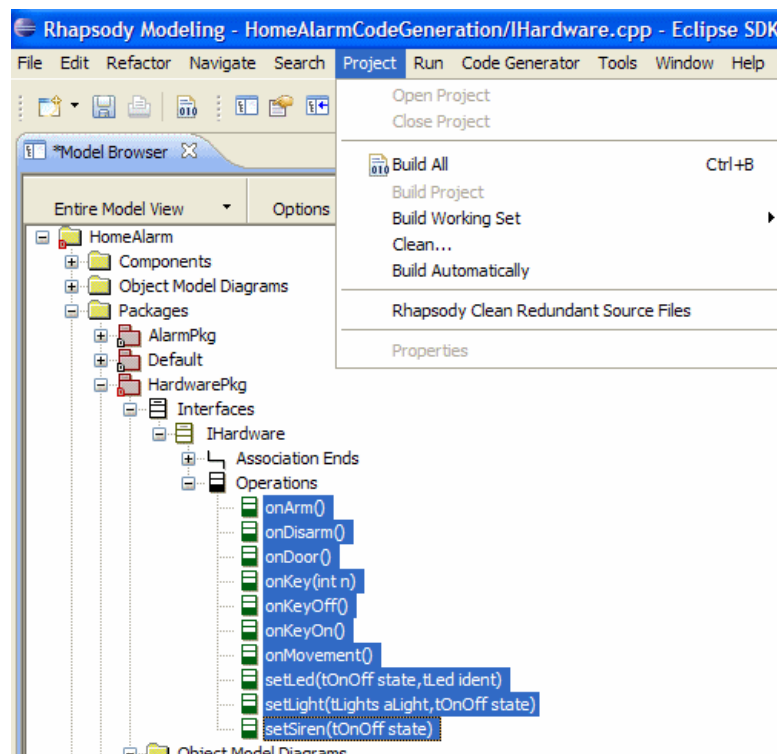
Build, debug, and animate

After checking your model and generating code, you can build your project.

Building your Eclipse project

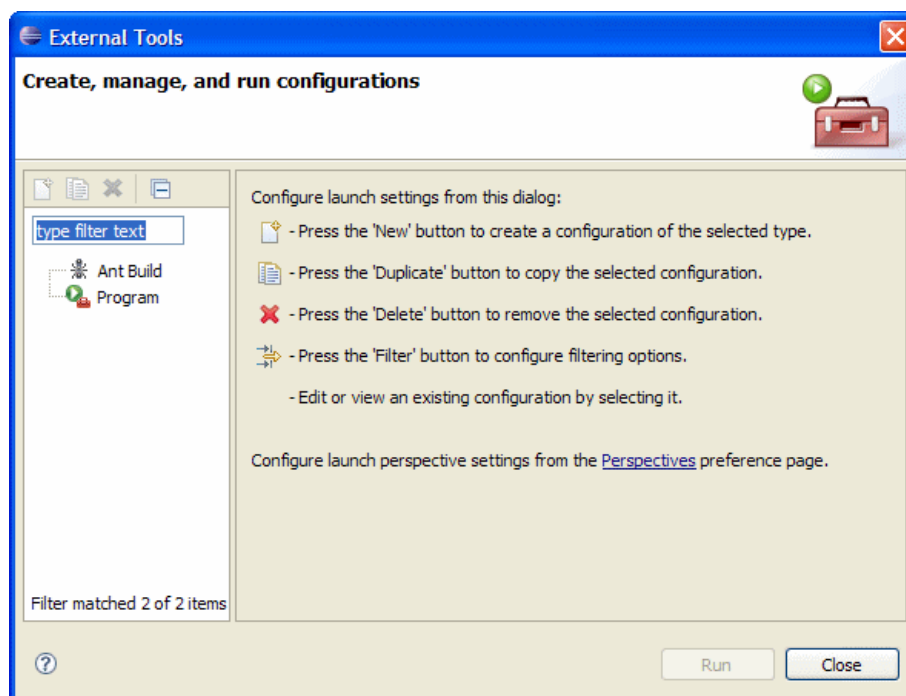
You can use any of these methods to build your Eclipse project:

- ◆ In the model browser, right-click the active configuration for your Eclipse project and select **Build Configuration**.
- ◆ Choose **Project > Build Automatically** to build from Eclipse using Java or C/C++ build tools.
- ◆ Select an element or group of elements in the model browser and choose the **Project** menu to display this menu and select a build option.



Debugging your Eclipse project

After you have built your project, you can use the Eclipse debugging facilities with the Rational Rhapsody Debug perspective. Choose the **Run** menu and then any of the debugging tools you usually use in Eclipse. The External Tools window provides access to programs used to create, manage, and run configuration you built project.



Rational Rhapsody animation in Eclipse

When running animated Rational Rhapsody applications, Eclipse switches to the Rhapsody Debug perspective. You can debug an animated application using both Rational Rhapsody Animation functionality and Eclipse Debugging tools.

For detailed instructions, see [Animation](#).

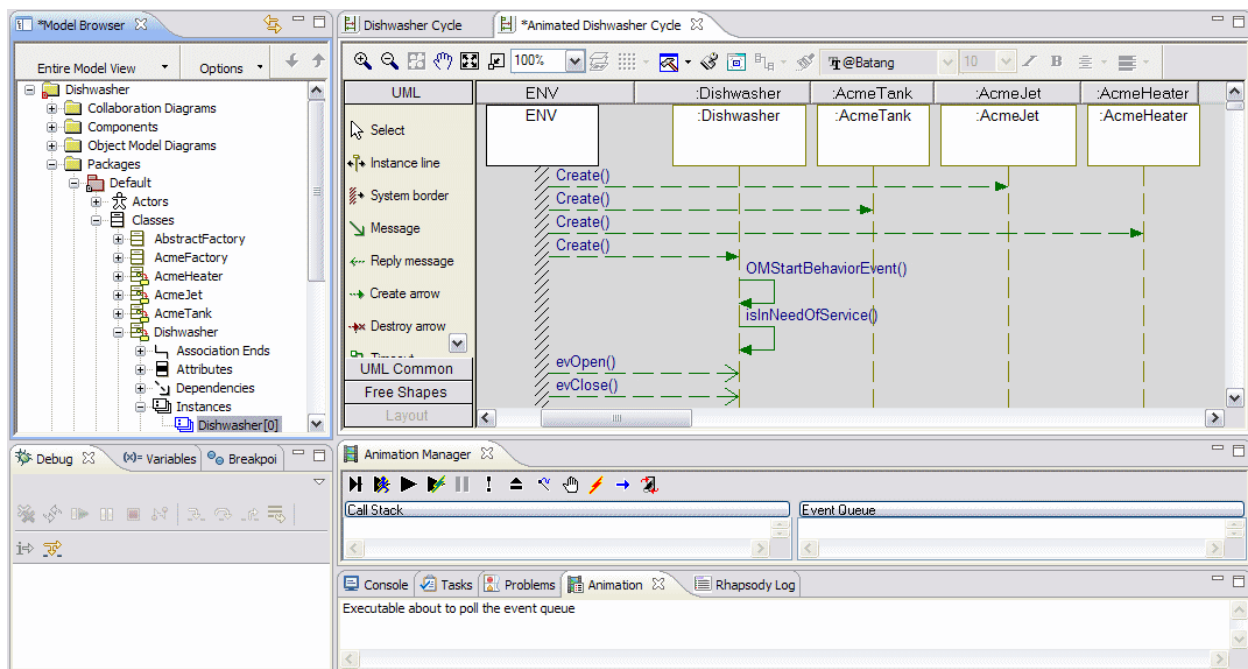
Preparing for animation

Before you can begin animation, you must prepare for it:

1. In the model browser for the project, expand the Components folder.
2. Right-click the configuration you want to animate in the configuration folder and open the **Features** window.
3. On the **Settings** tab, in the **Instrumentation Mode** box, select **Animation**.
4. Click **Apply**.
5. Right-click the configuration you want to animate in the configuration folder and select **Generate Configuration**.

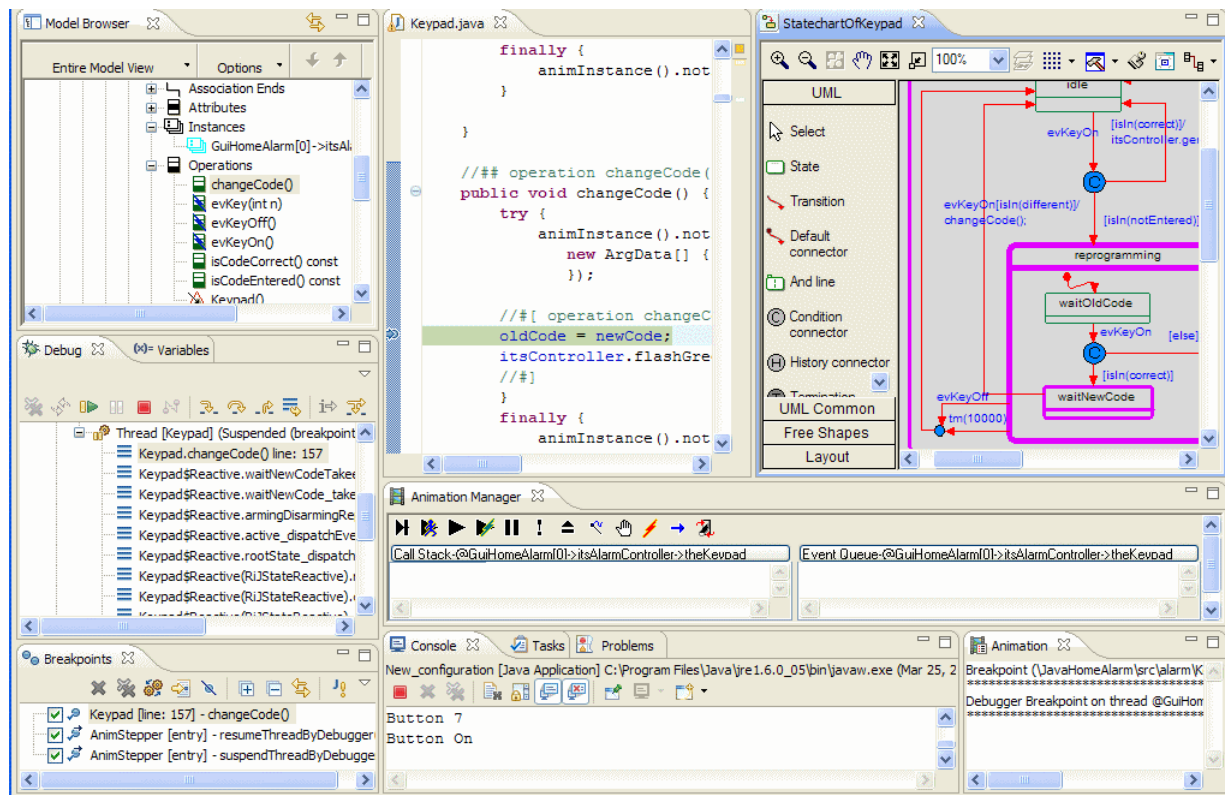
Run animation

Eclipse displays the Animation Manager window with Call Stack and Event Queue and an Animation toolbar (at the top of the Animation Manager window, as shown in the following figure) to perform all of the Rational Rhapsody standard animation tasks, watch the animation, and note the messages on the Animation Log tab.



Debug animated applications

You might want to use the animated applications for debugging. The Java example shows the moment that a breakpoint is hit. The breakpoints are listed in the lower left window with the application output to the right.



Eclipse configuration management

The Rational Rhapsody Eclipse plug-in integrates a Rational Rhapsody model into the Eclipse environment, enabling software developers to streamline their workflow with the benefit of working within the same development environment. You can work in the code or model in a single development environment. This enables you to use the Rational Rhapsody modeling capabilities or to modify the code using the Eclipse editor, while maintaining synchronization between both and easily navigating from one to the other.

Parallel development

When many developers are working in distributed teams, they often need to work in parallel. These teams use a configuration management (CM) tool, such as Rational ClearCase, to archive project units. However, not all files might be checked into CM during development.

Developers in the team need to see the differences between an archived version of a unit and another version of the same unit that might need to be merged. To accomplish these tasks, they need to see the graphical differences between the two versions, as well as the differences in the code.

A Rational Rhapsody unit is any project or portion of a project that can be saved as a separate file. These are examples of Rational Rhapsody units with the file extensions for the unit types:

- ◆ Class (.cls)
- ◆ Package (.sbs)
- ◆ Component (.cmp)
- ◆ Project (.rpy)
- ◆ Any Rational Rhapsody diagram

Note

The illustrations in this section show the use of the Rational ClearCase Eclipse Team plug-in. Different Team plug-ins (for example, Rational Synergy) might have different graphical user interfaces and menus.

Configuration management and Rational Rhapsody unit view

For the Eclipse plug-in, your individual Team plug-ins handle configuration management operations.

The Rational Rhapsody Unit View provides a hierarchical view of Rational Rhapsody model resources as per the model structure. Use this view to perform configuration management operations.

Navigating to the unit view

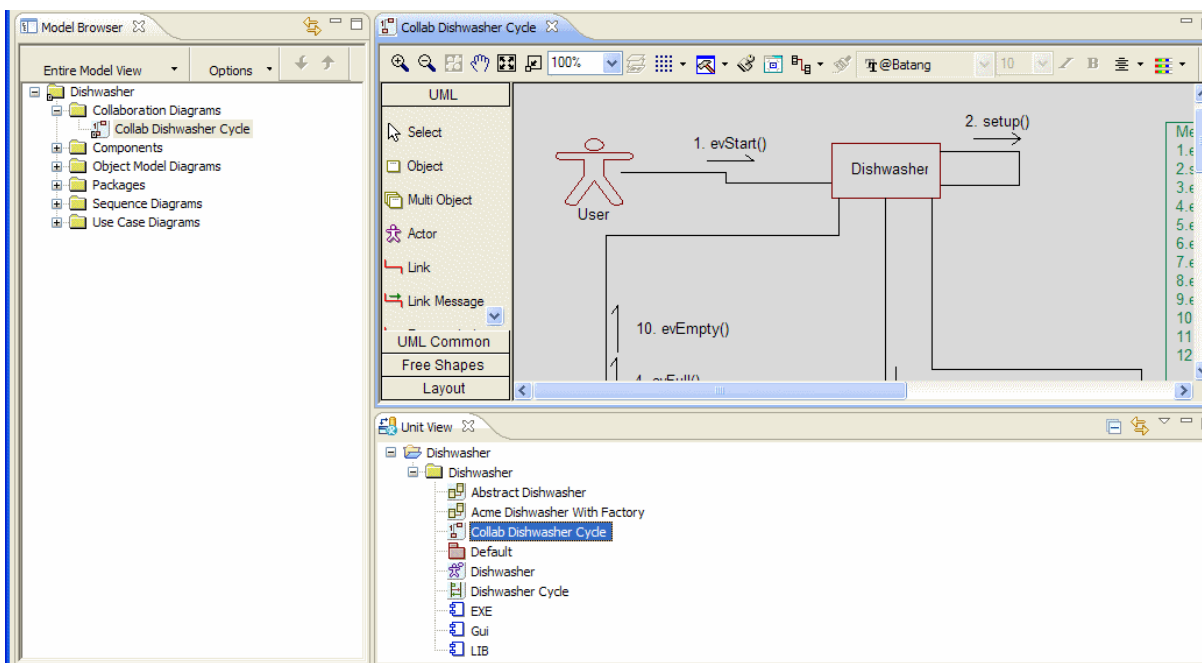
To navigate to the Unit View, on the model browser, right-click an element and select **Configuration Management > Select in Unit View**.

From the model browser, you can choose **Select with Descendants in Unit View**. This menu command selects all the descendants for Unit View.

Navigating to the model browser

To navigate to the model browser from the Unit View, right-click an element and select **Show in Model Browser**.

The following figure shows the result of selecting **Show in Model Browser**.

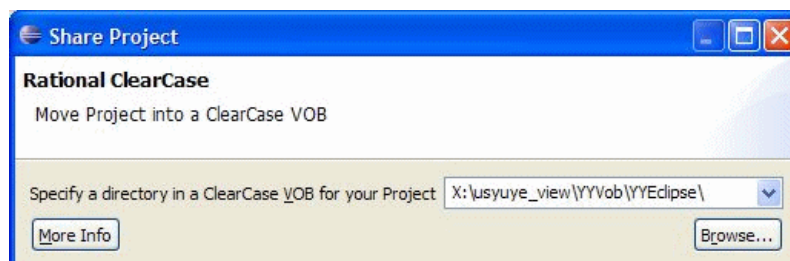


Sharing a Rational Rhapsody model

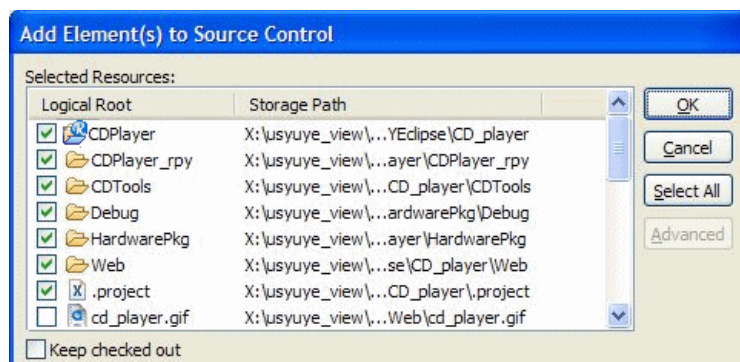
Team members can share a Rational Rhapsody model using configuration management.

To share a Rational Rhapsody model using configuration management:

1. If you have multiple Rational Rhapsody models loaded, make whichever project you want to share be the active one.
2. For the active Rational Rhapsody project, at the top-level project node in the Unit View, choose **Team > Share Project**.
3. On the Share Project window, select the repository plug-in that you want to use to share the selected project and click **Next**.
4. On the next window (which shows the name of the configuration management tool you selected in the previous step), enter the required information. For example, if you are using Rational ClearCase, specify a Rational ClearCase VOB for your project.



5. Click **Finish**.
6. Address any other windows that might open. For example, for Rational ClearCase, decide which elements you want to add to source control by clearing or selecting the applicable check boxes.



7. Click **OK**.

8. Restart Eclipse.

Performing team operations

To perform team operations in configuration management:

1. Before performing Team operations (such as Check In), save your Rational Rhapsody model in the model browser.
2. For a Rational Rhapsody project in Unit View, right-click the element you want, choose **Team** > (configuration management operation); for example, **Team** > **Check Out**.
3. On the window that opens, confirm the elements that you want to perform the configuration management operation by clearing or selecting the applicable check boxes, and then click **OK**.

Notice the white check mark within a green background  CDPlayer that denotes an element that is checked out.

Note

You can set Team plug-in related preferences through the Eclipse Preferences window (choose **Windows** > **Preferences**).

Rational Rhapsody DiffMerge facility in Eclipse

The DiffMerge tool can be operated inside and/or outside your CM software to access the units in the repository. It can compare two units or two units with a base (original) unit. The units being compared only need to be stored as separate files in directories and accessible from the PC running the DiffMerge tool.

In addition to the comparison and merge functions, this tool provides these capabilities:

- ♦ Graphical comparison of any type of Rational Rhapsody diagram
- ♦ Consecutive walk-through of all of the differences in the units
- ♦ Generate a Difference Report for a selected element including graphical elements
- ♦ Print diagrams, a Difference Report, Merge Activity Log, and a Merge Report

Generate Rational Rhapsody reports

To generate reports from the model, use the Rational Rhapsody ReporterPLUS documentation tool to create Microsoft Word, Microsoft PowerPoint, HTML, RTF, and text documents. The ReporterPLUS pre-defined templates extract text and diagrams from a model, arrange the text and diagrams in a document, and format the document.

Generating a report

To generate a report from the Rational Rhapsody model:

1. Choose **Tools > ReporterPLUS > Report on all model elements** or **Report on selected package**.
2. On the ReporterPLUS Wizard: Select Task window, select the output format you want and click **Next**.
3. Use the Browse  button on the ReporterPLUS Wizard: Select Template window to select a template from the template directory (for example, <Rational Rhapsody installation path>\reporterplus\Templates) and click **Next**.
4. On the Confirmation window, review the report criteria, and then click **Finish** to produce the report.
5. On the Generate Document window:
 - ♦ Enter a document name.
 - ♦ Browse to where you want to locate the files that will be produced.
 - ♦ Click the **Generate** button to generate your document.

6. Wait while your document is generated. ReporterPLUS spends some time loading the template and the model. Then it analyzes the model and the model element relationships.
7. When available, click **Yes** to open your report.

Properties

As an open tool, Rational Rhapsody provides a high degree of flexibility in how you set up the visual programming environment (VPE). Project *properties* are name-value pairs that enable you to customize many aspects of environment interaction and code generation. You can set existing properties, or create whole sets of new ones, to tailor Rational Rhapsody to your particular needs.

Note

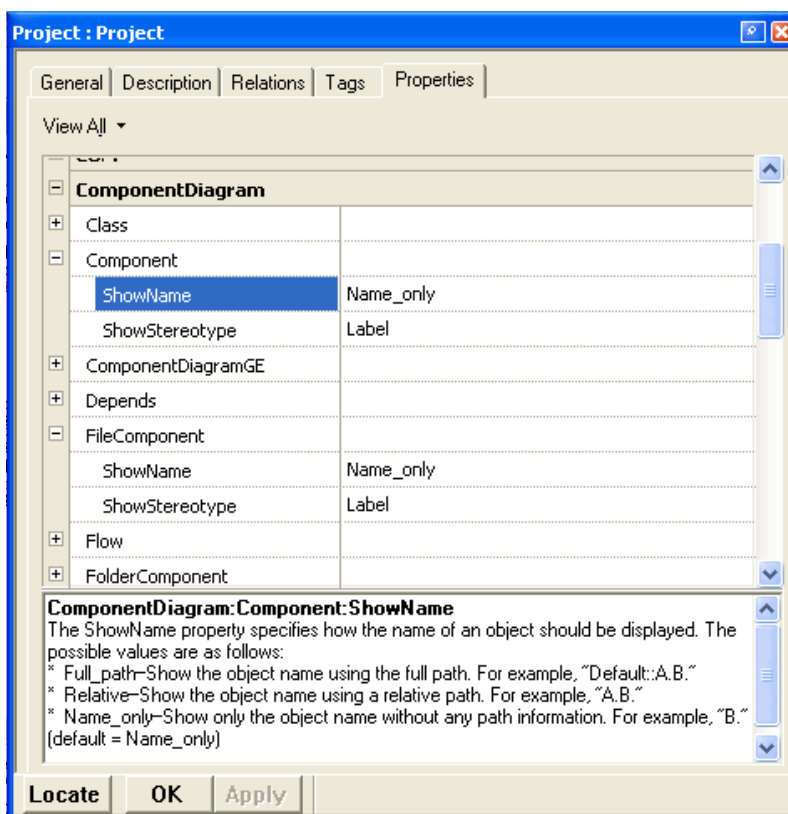
This section contains generic information about the Rational Rhapsody properties and how to work with them.

Rational Rhapsody properties overview

Properties are user-defined, tagged values that can be attached to any modeling element. You can think of each element as having its own set of properties. Rational Rhapsody tools, such as the code generator, reference many properties. You can modify properties to customize the tool to work in a certain way, such as setting the default color of a state box in a statechart. You can change properties at the site, diagram, package, configuration, or class level (or even at the individual operation or attribute level). Only properties that are relevant for a particular element are accessible from that element. The element on which you set a property determines its effectiveness. In other words, setting a property for a configuration provides a default for elements in the configuration. Precedence goes to the element with the lowest level of granularity. Meaning, properties explicitly defined for an individual operation would override those set at the project level.

Property groups and definitions

The Rational Rhapsody properties are classified according to *subject* and *metaclass* with the individual property names listed under each metaclass. Selecting a subject, metaclass, or property name listed in the Features window displays the definition of the selected item:



The right column indicates the type of information in the property value. Metaclasses are listed in alphabetical order under each subject. For information on changing property values, see [Rational Rhapsody properties](#).

Subjects

The following table lists the Rational Rhapsody subjects.

Subject	Description
Activity_diagram	Controls the appearance of activity diagrams.
Animation	Controls the behavior of black box animation.
ATL	Controls ATL classes. This subject applies only to Rational Rhapsody in C++.
Browser	Controls the information displayed in the Rational Rhapsody browser.
CG	Controls how code is generated. These properties are language-independent.
Collaboration_Diagram	Controls the appearance of collaboration diagrams
COM	Controls how mixed, distributed applications and objects find and interact with each other over a network. This subject applies only to Rational Rhapsody in C++.
ComponentDiagram	Controls the appearance of component diagrams.
ConfigurationManagement	Defines the command strings needed by various configuration management (CM) tools to interface with Rational Rhapsody.
<ContainerTypes>	Controls how items stored in containers are accessed and manipulated. The subject names are Java(1.1)Containers, Java(1.2)Containers, OMContainers, OMCorba2CorbaContainers, OMCpp2CorbaContainers, OMCppOfCorbaContainers, OMUContainers, RiCContainers, or STLContainers, depending on your programming language and environment.
CORBA	Controls how CORBA interacts with Rational Rhapsody. This subject applies only to Rational Rhapsody in C++.
DeploymentDiagram	Controls the appearance of deployment diagrams.
DiagramPrintSettings	Controls how diagrams are printed.
Dialog	Controls which properties are displayed on the Properties tab
General	Controls the general aspects of the Rational Rhapsody display.
IntelliVisor	Controls the IntelliVisor feature.
<lang>_CG	Controls the language-specific aspects of code generation. The subject name is C_CG, CPP_CG, or JAVA_CG.
<lang>_ReverseEngineering	Controls how Rational Rhapsody imports legacy code. The subject name is C_ReverseEngineering, CPP_ReverseEngineering, or JAVA_ReverseEngineering.

Subject	Description
<lang>_Roundtrip	Controls how changes made to the model are roundtripped to the code, and vice versa. The subject name is C_Roundtrip, CPP_Roundtrip, or JAVA_Roundtrip.
Model	Controls prefixes added to attributes, variables, and arguments to reflect their type.
ObjectModelGe	Controls the appearance of object model diagrams (OMDs).
QoS	Provides performance and timing information.
ReverseEngineering	Controls how Rational Rhapsody deals with legacy code.
RoseInterface	Controls how Rational Rhapsody imports models from Rational Rose® 98 or 2000.
RTInterface	Controls how Rational Rhapsody interacts with requirements traceability tools.
SequenceDiagram	Controls the appearance of sequence diagrams.
SPARK	Enables you to control the generation of SPARK annotations from Rational Rhapsody in Ada models so they can be analyzed by the SPARK Examiner.
Statechart	Controls the appearance of statecharts.
TestConductor	Controls contains properties that affect the TestConductor™ tool.
UseCaseExtensions	Controls extended UCDs.
UseCaseGe	Controls the appearance of use case diagrams (UCDs).
WebComponents	Controls whether Rational Rhapsody components can be managed from the Web, and specifies the necessary framework for code generation.

Metaclasses

Under each subject, Rational Rhapsody lists *metaclasses*. Metaclasses define properties for groups of things, such as attributes, classes, and configurations.

For example, under the Statechart subject and the State metaclass, the `color`, `line_width`, and `name_color` properties determine the default color and line width of state boxes and the text color of state names. The notation is `Subject::Metaclass::Property`; for example, `Statechart::State::Color`. Note that you can always change the properties of an element in a statechart or diagram on-the-fly (project properties specify the default appearance).

Regular expressions

Many properties use regular expressions to define valid command strings. For example, the `ParseErrorMessage` property uses the following regular expression for Microsoft® environments:

```
([^(]+)([()([0-9]+)]*) [:] (error|warning|fatal error)
```

This expression defines the rules used to parse error messages on Microsoft systems. If you redefine properties that require regular expressions, you must use the correct expression syntax.

Regular expression syntax

Regular expression syntax is defined¹ as a “regular expression is zero or more branches, separated by `|`. It matches anything that matches one of the branches. A branch is zero or more pieces, concatenated. It matches a match for the first, followed by a match for the second, and so on.”

- ♦ A piece is an atom possibly followed by `*`, `+`, or `?`.
- ♦ An atom followed by `*` matches a sequence of 0 or more matches of the atom.

For example, the atom `.` matches zero or more instances of any character (a period matches any character).

- ♦ An atom followed by `+` matches a sequence of 1 or more matches of the atom.

For example, the atom `.` matches one or more instances of any character.

- ♦ An atom followed by `?` matches a match of the atom, or the null string.

For example, the atom `.` matches a single character or the null string, such as at the end of an input string.

- ♦ An atom is a regular expression in parentheses (matching a match for the regular expression), a range, or:
 - `.` (matching any single character)
 - `^` (matching the beginning of the input string)
 - `$` (matching the end of the input string)
 - `A \` followed by a single character (matching that character)
 - A single character with no other significance (matching that character)

Consider the following regular expression:

```
([a-zA-Z_][a-zA-Z0-9_]*)
```

1. Copyright (c) 1986 by U. of Toronto. Written by Henry Spencer.

This regular expression, enclosed in parentheses, matches a sequence of two ranges, any single uppercase or lowercase letter, or underscore character; followed by zero or more uppercase or lowercase letters, digits 0-9, or the underscore character.

Parsing regular expressions

Parts of the expression contained within parentheses are called *tokens*.

The regular expression for the `ParseErrorMessage` property is as follows:

```
([^()]+)(([0-9]+))[:] (error|warning|fatal error)
```

It consists of the following parts:

- ◆ `([^()]+)` This is the first token. The caret at the beginning of the set is a NOT operator that matches any character except those in the set. This token tells the parser to ignore all characters until the first occurrence of an open parenthesis.
- ◆ `[ ( ]` The parser should search for exactly one opening parenthesis.
- ◆ `([0-9]+)` This is the second token. It tells the parser to search for a sequence of one or more digits in the range of 0 to 9.
- ◆ `[ ) ]` The parser should search for exactly one closing parenthesis.
- ◆ `[ : ]` The parser should search for exactly one colon.
- ◆ `(error|warning|fatal error)` This is the third token. It tells the parser to search for one of the strings “error,” “warning,” or “fatal error.”

The `ErrorMessageTokensFormat` property works with `ParseErrorMessage` to determine how many tokens can be contained in an error message, and the relative positions in the message string of tokens that represent the file name and line number of the error, respectively. The second token in the sample regular expression would most likely represent a line number, depending on how `ErrorMessageTokensFormat` was defined.

Based on this regular expression, the parser would interpret the string “(3457):warning” as a valid error message indicating a warning condition at line 3457 in the program.

Property file format

All the property (*.prp) files use an LL1 syntax for a simple, recursive descent parser. The parser currently has no error recovery and effectively stops at the first error. Tokens enclosed within curly braces {} are optional. Those enclosed within angle brackets <> are further decomposed according to their own BNF (Backus Naur Form) descriptions.

The BNF for the *.prp files is as follows:

```
<file> ::= {"Subject" <subject>} "end"
```

For example, the factory.prp file begins with an optional list of subjects, each beginning with the keyword “Subject,” and ends with the required keyword “end”:

```
Subject General
Subject Statechart
Subject ObjectModelGe
.
.
.
end
```

```
<subject> ::= <name> {"Metaclass" <metaclass>} "end"
```

As another example, the subject General begins with a name, followed by a list of metaclasses, followed by the keyword “end”:

```
Subject General
Metaclass Graphics
Metaclass Model
end
```

```
<metaclass> ::= <name> {"Property" <property>} "end"
```

The file contains the following type declarations:

- ◆ “Bool”

A string that indicates a type with two possible values, TRUE or FALSE.

- ◆ <enum values> ::= <quoted string>

The enum values string is a comma-separated list of legal, enumerated values. A second quoted string indicates the default. For example, the quoted string “on,off” contains enumerated values.

- ◆ <value> ::= <quoted string>

A value. For example, a property value could be the quoted string “Arial 10 NoBold NoItalic”.

- ◆ <quoted string> ::= <quote> <escaped chars> <quote>

A quoted string is a string that starts and ends with double-quotes and can contain newlines. A backslash must precede any literal double-quote or backslash characters within the string. For example, “FALSE” is a quoted string.

Rational Rhapsody keywords

Many properties reference other properties, using the \$ symbol. For example, the command string for the `ConfigurationManagement::ClearCase::AddMember` property begins as follows:

```
"$OMROOT/etc/Executer.exe"
```

This substring references the predefined variable `OMROOT`, set in your `rhapsody.ini` file to the location of the `Share` directory in the Rational Rhapsody installation. Expanded, this string becomes:

```
"<install_dir>\Share/etc/Executer.exe"
```

For a description, see [The Executer](#).

Keywords are used in the following areas:

- ◆ Makefile generation
- ◆ Standard operations
- ◆ Relation implementation properties
- ◆ Names of generated operations
- ◆ Headers and footers
- ◆ Configuration management

Predefined variables

The following table lists the predefined variables used in Rational Rhapsody.

Keywords	Where Used	Description
\$archive	ConfigurationManagement	The file name (including the full path) of the archive that you selected in the Connect to Archive window. This can be either a file or a directory.
\$archiveddirectory	ConfigurationManagement	The directory part of \$archive. If \$archive is a directory, \$archive and \$archiveddirectory are the same.
\$arguments	ATL	The arguments of the operation.
\$Arguments	<lang>_CG	The event or operation argument's description, used by the DescriptionTemplate property.

Properties

Keywords	Where Used	Description
\$attribute	CG	The object of an operation on attributes. The qualifier : c capitalizes the name of the attribute.
\$base	<lang>_CG	The name of the reactive object.
\$CheckOut	ConfigurationManagement	The command executed to check configuration items out of the archive using the main Configuration Items window.
\$class	ATL	The name of the ATL class.
\$ClassClean	Makefiles	The list of class files used in a build.
\$cname	CG, <Container Types>, <lang>_CG	The name of the container used to hold relations. Typical containers are arrays, lists, stacks, heaps, and maps.
\$coclass	ATL	The name of the coclass that exposes the COM interface.
\$CodeGeneratedDate	CG, <lang>_CG	The date of code generation. This information is printed in the headers and footers of generated files.
\$component	ATL	The name of the component,
\$ComponentName	CG	The name of the component that caused the code to be generated. This information is printed in the headers and footers of generated files.
\$ConfigurationName	CG, <lang>_CG	The name of the configuration that caused the generation of the model element found in a file. This information is printed in headers and footers of generated files.
\$datamem	ATL	The data member.
\$DeclarationModifier	ATL	The declaration modifier.
\$Description	<lang>_CG	The element description, used by the DescriptionTemplate property.
\$Direction	<lang>_CG	The argument direction (in, out, and so on), used by the DescriptionTemplate property.
\$dupinterface	ATL	The name of the duplicate interface.
\$executable	<lang>_CG	The path to the executable binary file generated by the Rational Rhapsody code generator.

Keywords	Where Used	Description
\$FILENAME	CPP_CG	The name of the file used: - To generate source code for individual classes to user-specified directories - To specify that a statement should not be imported during reverse engineering (the #ifndef that protects h files from multiple includes)
\$Fork	Framework: start method	Used to specify whether the OMMainThread singleton event loop should run on the application main thread or in a separate thread.
\$FullCodeGeneratedFileName	CG, <lang>_CG	The full path name of the file. This information is printed in headers and footers of generated files.
\$FULLFILENAME	CG	The full name of the file used: - To generate source code for individual classes to user-specified directories - To specify that a statement should not be imported during reverse engineering (the #ifndef that protects h files from multiple includes)
\$FullModelElementName	CG, <lang>_CG	The full name of a model element in <package>::<class> format. is printed in headers and footers of generated files. For example, Radar::Engine, for a class named Engine found in a package named Radar.
\$FullName	<lang>_CG	The full path of the element (P1::P2::C.a) used by the DescriptionTemplate property.
\$id	ATL	The identifier.
\$IDInterface	ATL	The interface ID of a COM interface.
\$index	<Container Types>	An index used to randomly access items in a container.
\$instance	Property: CORBA::TAO::InitialInstance	Refers to the default initial instance of the TAO ORB.
\$interface	ATL	The name of the interface.
\$interfaceSeq	Property: CORBA::Class::IDLSequence	Represents the name of the CORBA interface with the string Seq added to the end of the term.

Properties

Keywords	Where Used	Description
\$item	CG, <Container Types>	A class or instance whose behaviors are implemented by a container. Rational Rhapsody generates various add, remove, find, and get operations to manipulate items in containers.
\$iterator	<Container Types>	The name of the iterator used to traverse a container.
\$keyname	<Container Types>	The name of a key used to access items in maps. A key is usually a string that maps to a dictionary that is used to locate items.
\$label	ConfigurationManagement	An optional revision label of a configuration item, provided in the Check In/Check Out window.
\$log	ConfigurationManagement	An optional comment provided in the Check In window.
\$LogPart	ConfigurationManagement	The user-specified comment for the CM operation.
\$Login	CG, <lang>_CG	The login name of the user who generated the file. This information is printed in headers and footers of generated files.
\$makefile	<lang>_CG	The name of the makefile generated by the Rational Rhapsody code generator.
\$maketarget	<lang>_CG	Depending on the option selected in the Code menu, this expands to one of the following operations: • Build • Clean • Rebuild
\$member	<lang>_CG	The name of the reactive member (equivalent to the base class) of the object.
\$mePtr	<lang>_CG	The name of the user object (the value of the Me property). The member and mePtr objects are not equivalent if the user object is active.
\$mode	ConfigurationManagement	A flag indicating the locking mode provided in the Check In/Check Out window. If the item is locked, \$mode is replaced with the contents of the CM property ReadWrite. If unlocked, \$mode is replaced with the contents of the ReadOnly property.

Keywords	Where Used	Description
\$ModePart	ConfigurationManagement	The locking mode of the CM operation. For example, you can check out a file from an archive as either locked or unlocked.
\$Name	<lang>_CG	The element name, used by the DescriptionTemplate property.
\$noOfArgs	ATL	The number of arguments for the operation.
\$OMAdditionalObjs	Makefiles	The list of files to be included in the executable.
\$OMAllDependencyRule	Makefiles	The dependency rule of a specific source file (A.cpp: A.h B.h C.idl).
\$OMBldSet	Makefiles	The compiler switches for Debug versus Release mode, as specified in the Settings window for the active configuration.
\$OMCleanOBJS	Makefiles	The list of delete commands for each object file in the makefile. Each entry in the list is created from the value of the ObjCleanCommand property.
\$OMCOM	Makefiles	Specifies that the COM application to be linked is a windows application rather than a console application. This keyword is resolved based on the value of the <lang>_CG::<Environment>::COM property.
\$OMConfigurationCPPCompileSwitches	Makefiles	The compiler switches specified by the CompileSwitches property for a configuration.
\$OMConfigurationLinkSwitches	Makefiles	The link switches of the configuration, set in the Settings tab for the configuration.
\$OMContextDependencies	Makefiles	The list of dependencies and the compilation command for each model file that should be built as part of the component. Each entry is made up of the value of the DependencyRule property followed by the value of the CPPCompileCommand property.

Keywords	Where Used	Description
\$OMContextMacros	Makefiles	The set of generated macros, including: • OMRROOT • CPP_EXT/C_EXT • H_EXT • OBJ_EXT • LIB_EXT • INSTRUMENTATION • TIME_MODEL • TARGET_TYPE • TARGET_NAME • The “all” rule • TARGET_MAIN • LIBS • INCLUDE_PATH • ADDITIONAL_OBJS • OBJS For more information, see MakeFileContent.
\$OMCPPCompileCommandSet	Makefiles	The compilation switches related to the CPPCompileDebug/ CPPCompileRelease properties. The property to be used is based on the value of the BuildCommandSet property. Set the value of BuildCommandSet using the configuration Settings tab in the browser.
\$OMCPPCompileDebug	Makefiles	The compile switches needed to create a Debug version of a component in a given environment, as specified by the CPPCompileDebug property.
\$OMCPPCompileRelease	Makefiles	The compile switches needed to create a Release version of a component in a given environment, as specified by the CPPCompileRelease property.

Keywords	Where Used	Description
\$OMFileCPPCompileSwitches	Makefiles	This is used in the <code>CPPCompileCommand</code> property to bring in additional GUI-defined settings. The content is generated by Rational Rhapsody (either based on content of fields or based on internal rules). It is one of the predefined keywords including, but not limited to: • <code>\$OMCPPCompileDebug</code> • <code>\$OMCPPCompileRelease</code> • <code>\$OMLinkDebug</code> • <code>\$OMLinkRelease</code> • <code>\$OMBuildSet</code> • <code>\$OMContextMacros</code>
\$OMDefaultSpecificationDirectory	Makefiles	Supports the default specification/implementation source directory feature. To set a default directory for a configuration, set the <code><lang>_CG::Configuration::DefaultSpecificationDirectory</code> and <code><lang>_CG::Configuration::DefaultImplementationDirectory</code> properties
\$OMDEFExtension	Makefiles	The extension of the definition file (<code>.def</code>). This keyword applies to the MicrosoftDLL/COM environments.
\$OMDllExtension	Makefiles	The extension of the dynamic linked library file (<code>.dll</code>). This keyword applies to the MicrosoftDLL/COM environments.
\$OMExeExt	Makefiles	The extension of the compiled executable.
\$OMFileDependencies	Makefiles	Used as part of a source file dependency line. It is a calculated list of files on which the source file depends.
\$OMFileImpPath	Makefiles	The relative name and path of the implementation file. It is used in a source file dependency and compilation commands.

Properties

Keywords	Where Used	Description
\$OMFileObjPath	Makefiles	The relative path and name of an object file that is related to a given implementation and specification files. It is used as part of a file compilation command.
\$OMFileSpecPath	Makefiles	The relative path and name of a specification file. It is used in a source file dependency line.
\$OMFlagsFile	Makefiles	Maintained for backwards compatibility.
\$OMImpIncludeInElements	Makefiles	The list of all <code>#includes</code> done in the related implementation file. It is used as part of a source file dependency line.
\$OMImplExt	Makefiles	The extension of an implementation file generated for a model element.
\$OMIncludePath	Makefiles	The include path. The path is calculated from dependencies between components and from the Include Path setting in the active component/configuration feature window.
\$OMInstrumentation	Makefiles	The active configuration instrumentation mode (None, Tracing, or Animation).
\$OMInstrumentationFlags	Makefiles	Represents the preprocessor directives required for the selected type of instrumentation: animation, tracing, or none.
\$OMInstrumentationLibs	Makefiles	Represents the libraries required for the selected type of instrumentation: animation, tracing, or none.
\$OMLibExt	Makefiles	The extension of library files.
\$OMLibs	Makefiles	The names of additional libraries (besides the framework library) to link when building a component. It is calculated from dependencies between components and the Libraries list in the active component/configuration feature windows.
\$OMLibSuffix	Code Generation	Represents the suffix to use for library names. The keyword is replaced by the value of the <code>DebugLibSuffix</code> property or the <code>ReleaseLibSuffix</code> property depending upon the build.

Keywords	Where Used	Description
\$OMLinkCommandSet	Makefiles	The link switches related to the LinkDebug/LinkRelease properties. The property to be used is based on the value of the BuildCommandSet property. Set the value of BuildCommandSet using the configuration Settings tab in the browser.
\$OMLinkDebug	Makefiles	The environment-specific link switches used to build a Debug version of a component. This is the value of the LinkDebug property.
\$OMLinkRelease	Makefiles	The value of the LinkRelease property.
\$OMMainImplementationFile	Makefiles	The main file name and path: [<imp dir>/]\$TARGET_MAIN)\$(CPP_EXT)
\$OMMakefileName	Makefiles	The name of the makefile.
\$OMModelLibs	Makefiles	The library component the model depends on. For example, if executable component A depends on the library component L, this keyword is replaced with the string <filepath>\L.lib.
\$OMObjExt	Makefiles	The extension of object files (temporary compiler files) for a given environment. This is the value of the ObjExtension property.
\$OMObjs	Makefiles	The list of object files to link into the build by the makefile.
\$OMObjectsDir	Makefiles	A calculated keyword based on the property <lang>_CG::<Environment>::ObjectsDirectory).
\$OMROOT	ConfigurationManagement, General, <lang>_CG, <lang>_Roundtrip, makefiles	The location of the \Share subdirectory in the Rational Rhapsody installation. This is set in your rhapsody.ini file.
\$OMRPFrameworKDll	Makefiles	Links the COM application with the DLL version of the framework instead of the default static libraries. This keyword is resolved based on the value of the <lang>_CG::<Environment>::RPFrameworKDll property.

Properties

Keywords	Where Used	Description
\$OMRulesFile		Maintained for backwards compatibility.
\$OMSourceFileList	Makefiles	(Rational Rhapsody in J) Lists the source (* . java) files used in a build.
\$OMSpecExt	Makefiles	The extension of the specification file generated for a model element.
\$OMSpecIncludeInElements	Makefiles	Lists all the #includes done in the related specification file.
\$OMSubSystem	Makefiles	The type of program for the Microsoft linker (for example, windows).
\$OMTargetMain	Makefiles	The name of the file that contains the main () function for an executable component.
\$OMTargetName	Makefiles	The name of the compiled version of a component.
\$OMTargetType	Makefiles	The type of component to be built (library or executable),
\$OMTimeModel	Makefiles	The time model setting for a configuration (simulated or real time).
\$OMUserIncludePath	INTEGRITY build files (.gpj)	Represents the content of the Include Path field found on the Settings tab of the Features window for configurations. This content is included in generated .gpj files for environments that use such files, for example, INTEGRITY5.
\$operations	ATL	The list of operations.
\$opname	ATL	The name of the operation.
\$opRetType	ATL	The return type of the operation.
\$package	ATL	The name of the package.
\$PackageLib	ATL	The package library.
\$ProgID	ATL	The value of the ProgID property (Default = \$component.\$class.1).
\$projectname	ConfigurationManagement	The project name.
\$<Property>	<lang>_CG	The value of the element property with the specified name (under C or CPP_CG::CG::<metatype>). This keyword is used by the DescriptionTemplate property.

Keywords	Where Used	Description
\$RegTlb	ATL	Specifies whether the COM server needs to register its type library. Automatically expands to TRUE/FALSE depending upon COM ATL server includes type library.
\$RhapsodyVersion	CG, <lang>_CG	The current version of Rational Rhapsody, not including the build number. This information is printed in headers and footers of generated files.
\$rhpdirectory	ConfigurationManagement	The path to the _rpy directory, which consists of the project repository. The repository contains all the configuration items for a project.
\$Signature	<lang>_CG	The operation signature, used by the DescriptionTemplate property.
\$state	Properties CPP_CG::Framework::IsInCall CPP_CG::Framework::IsCompletedCall	In the code generated by Rational Rhapsody for checking whether an application is in a given state, this keyword is replaced by the state name.
\$target	<Container Types>, <lang>_CG	The target of an operation on relations. This is generally the role name. For example, in a class with a relation called myServer, the role name myServer would replace the variable \$target when expanding properties that involve that relation. The value add\$target:c would become: addMyServer() The qualifier :c capitalizes the role name.
\$Target	<lang>_CG	The other end of the association, used by the DescriptionTemplate property.
\$targetDir	ConfigurationManagement	The target directory.
\$ThreadModel	ATL	The value of the ThreadingModel property (Default = Apartment).
\$tlbPath	ATL	The full path of the COM type library file.

Keywords	Where Used	Description
\$type	CG, <lang>_CG	The name of the type. For example, if you create a type named MyType and set its in property to "const \$type&", the generation of an in argument will be as follows: "const MyType& <argname>"
\$Type	<lang>_CG	The argument type, used by the DescriptionTemplate property.
\$TypeName	ATL	The value of the TypeName property, which specifies the declaration of the class type being registered (Default = \$class).
\$unit	ConfigurationManagement	Unit of collaboration. This is the name of the file that corresponds to the configuration item (package, configuration, or diagram) on which a CM command operates. If more than one unit is provided, the command is performed repeatedly in a for each loop.
\$VersionIndepProgID	ATL	Replaced with the value of the VersionIndepProgID property (Default = \$component.\$class).
\$VtblName	<lang>_CG	The name of the object's virtual function table, specified by the ReactiveVtblName property.

The following table lists the predefined Rational Rhapsody macros used in the framework files and makefiles.

Macro	Description
AR	The command to build a library.
ARFLAGS	The flags used to build a library.
CP	Environment-specific copy command.
CPP_EXT	Environment-specific extension for C++ implementation files (for example, .cpp).
DLL_CMD	Expands to the DLL link command that initiates the DLL link phase of a build
DLL_FLAGS	Expands to the switches applied to the DLL link command
H_EXT	Environment-specific extension for C++ implementation files (for example, .h).

Macro	Description
INCLUDE_QUALIFIER	The qualifier used in a given environment to designate an include file in the compiler or link switches.
LIB_CMD	The command to build a library.
LIB_EXT	Environment-specific extension for library files (for example, .lib).
LIB_FLAGS	The flags used to build a library.
LIB_NAME	The name of a library.
LIB_POSTFIX	The postfix added between the main file name and the extension. The possible values are as follows: • sim for simulated time (for example, oxfsim.lib) • inst for instrumentation (for example, oxfinst.lib) • siminst for simulated time and instrumentation (for example, oxfsiminst.lib) This macro is not used for DLLs.
LIB_PREFIX	The prefix added to the beginning of a file name. For example, the prefix "Vx" is added to VxWorks libraries. This macro is not used for DLLs.
LINK_CMD	Expands to the link command that initiates the link phase of a build
LINK_FLAGS	Expands to the link switches applied to the link command
OBJ_EXT	The environment-specific extension for object files (for example, .o or .obj).
OBJS	The intermediate object files to be built (for example, aombrk.obj).
PDB_EXT	The environment-specific extension for PDB debug files (for example, .pdb).
RM	The environment-specific remove command for deleting files.
RMDIR	The environment-specific remove command for deleting directories. This is used in the clean rules when you set the <lang>_CG::<Environment>::ObjectsDirectory property.

Map custom properties to keywords

You can define custom keywords in makefile template properties and standard operations. The property name for the custom keyword should be the same as the keyword string. For example, for the keyword \$AAA, the property name should be AAA.

Define the property in a specific Subject and Metaclass, as follows:

Property Type	Subject	Metaclass
Makefile	CG/<lang>_CG	Component/ Configuration/ <Environment>
Standard operations	CG/<lang>_CG	The keyword context (class, relation, attribute, and so on)

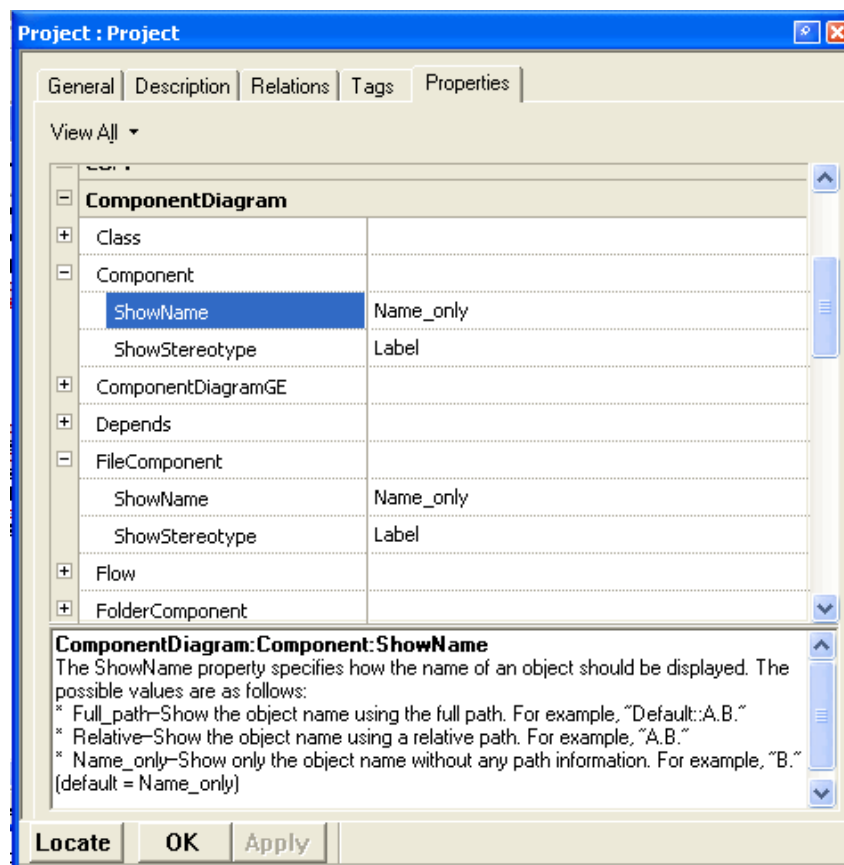
Rational Rhapsody properties

Properties affect aspects of your model, such as the appearance of graphics in various graphic editors, how code is generated, or configuration management settings.

Using the Properties tab in the Features window

Rational Rhapsody provides easy access to the properties through the interface. Use either of the following methods to display properties:

- ◆ With a project open, choose **File > Project Properties** to access all the properties for a model. The Features window displays with the **Properties** tab already selected.
- ◆ Right-click an item in the browser or on a diagram and select **Features** to open the Features window, and then select the **Properties** tab to list the properties for the selected item.



Properties definitions display

The **Properties** tab uses a tree structure to display the subjects, metaclasses, and properties.

In the left column, the subjects are listed in boldface font; expand the plus sign to view the metaclasses for a particular subject. When you expand a metaclass, the corresponding properties are listed in the left column, with their current values (if any) listed in the right column.

For example, in the figure, the `ObjectModelGe::Class::ShowName` property can have the values `Full_path`, `Relative` (the default value), and `Name_only`.

Note that items are usually in alphabetical order; however, metaclasses that are of the same type as the context are “pushed up” to be first. For example, if you are viewing the properties of a selected class, the first metaclass displayed is `CG::Class`.

Selecting a property displays the definition for the property in the bottom pane of the window. Each time a new property is selected, its definition displays below. Definitions are displayed for all three of the property levels: subject, metaclass, and the individual property.

Searching for properties

The **Properties** tab contains a menu to filter the properties displayed in the window.

The **All** option displays all of the available properties for your selection. You can display only the **Overridden** properties for the model or only the properties you overrode locally with the **Locally Overridden** option. The **Common** properties are those most often changed for the selected type of item.

The **Filter** menu option lets you search for specific properties by entering a filter string. Rational Rhapsody displays only the properties that contain the text you entered. When you are filtering the properties and select the **Match property description** check box, Rational Rhapsody searches the property definitions for the string you entered, as well as the property name.

Note

The Filter mechanism is not case-sensitive and does not allow the use of wildcards or expressions.

If you enter more than one word as the Filter, Rational Rhapsody performs an “or” search and displays all of the properties that contain any one of the words entered. To limit the search to only the definitions containing the entire phrase, enclose the words in the search string within quotation marks.

As long as the Features window (or standalone properties window) remains open, the selection you made from the menu (filter text and check box setting) is retained. When the Features window is closed, these are reset.

Resizing the Features window

Some of the properties might have long tables of information. You might find it necessary to resize the Features window in order to see these tables without awkward text wrapping outside the table columns. You can use the typical Windows methods to resize the Features window.

In addition, you can select the vertical line separator to resize the columns for the property names and their values. When you click the vertical separator, a solid-looking line displays so you can control the column widths and display. You can click the horizontal separator to resize the bottom pane of the **Properties** tab where the property definitions appear.

Note

Any continuation of a long “Default Value” is indented under that heading until a new table entry begins.

Filtering views

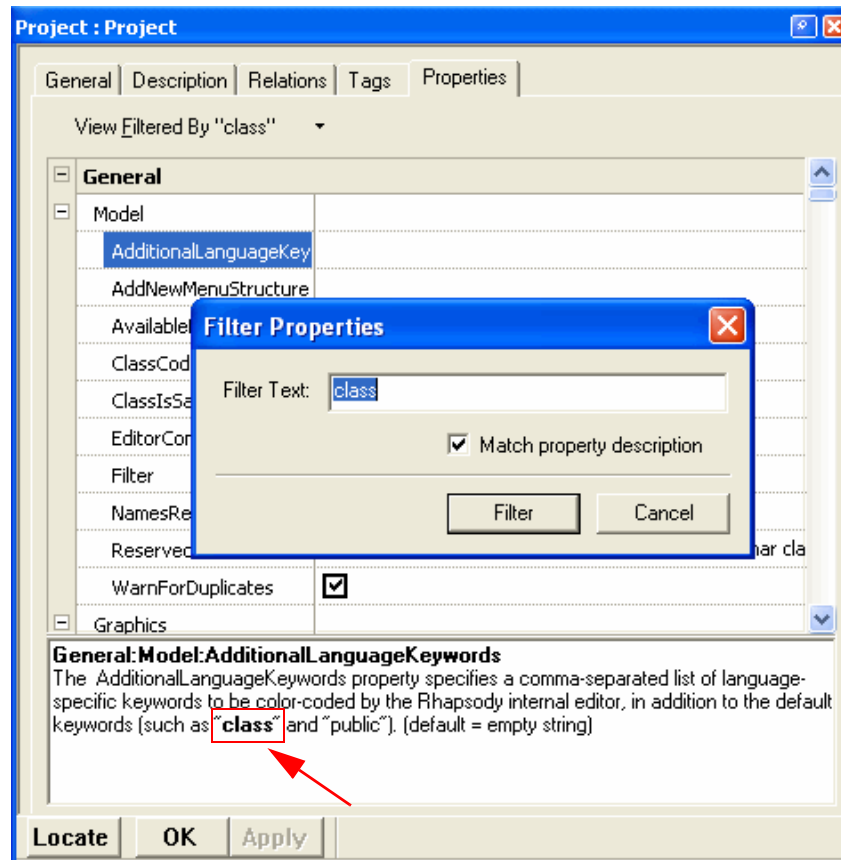
To select the types of properties that are displayed in the Features window, you can specify the view for the View menu.

The possible views are as follows:

- ◆ **All** displays all the available properties, according to context.
- ◆ **Overridden** displays only those properties whose default values have been overridden, up to the project level. When you select this view, the GUI displays all the overridden properties from the selected element up to the scope of the project; overridden properties at a scope higher than the selected element are displayed as regular, non-overridden properties. From the menu, you can also select the **Un-override** option to reverse this action.
- ◆ **Locally Overridden** displays only the locally overridden properties for the selected element. A selected element is a project, component, configuration, package, diagram, view element, and any other model element displayed in the browser.
 - Note:** To specify the default filter used, set the `Dialog::General::PropertiesDialogDefaultFilter` property.
- ◆ **Common** displays the properties contained in the `Dialog::<Metaclass>::CommonProperties` property. This is the default view.
- ◆ **Filter** displays the Filter Properties window to search for specific properties and definitions.

Filtering properties

To search for specific property names or text in descriptions, select Filter from the Properties tab View menu. This feature allows you to input and search for any text in the Property names and the descriptions. Selecting the “Match property description” check box searches property descriptions in addition to the names. Text located through a Filter Properties search is displayed in bold type in the property definition area of the Features window.



Adding and removing the common view

The common view enables you to see only a subset of the hundreds of Rational Rhapsody properties that are available. This makes the properties GUI much easier to use. You can easily add properties that you use frequently to the common view, or remove properties that you do not use.

To add a property to the common view:

1. In the properties GUI, select the All filter so you can find the property to add to the common view.
2. Right-click the property you want to add to the common view and select **Add To common list**.

To remove a property to the common view:

1. In the properties GUI, select the Common filter so you can find the property to add to the common view.
2. Right-click the property you want to remove and select **Remove from common list**.

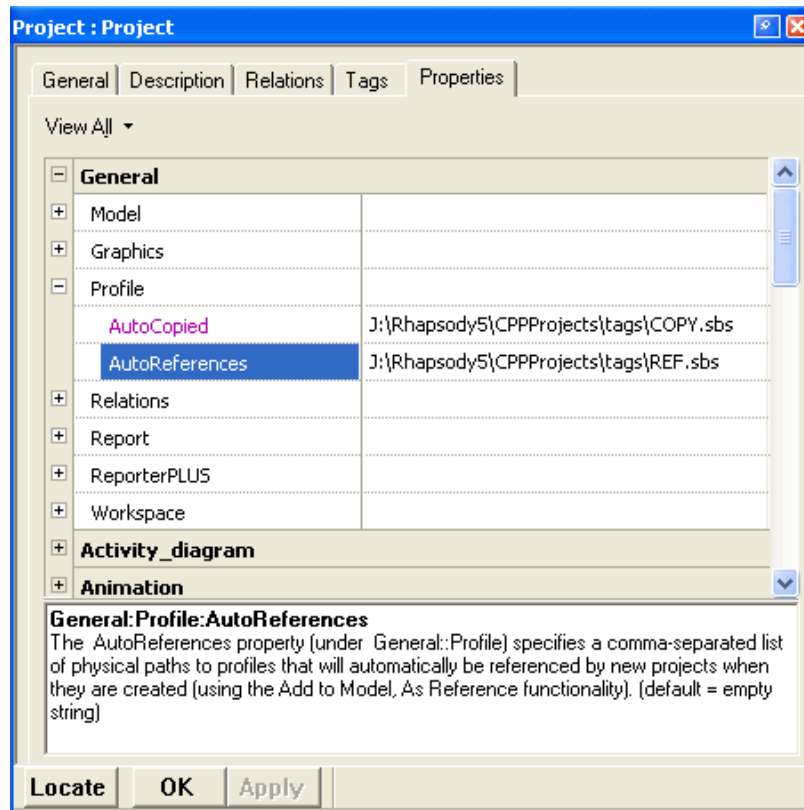
Property controls

The **Property** tab uses different controls depending on the value type of the property (enum, Boolean, and so on). The following table lists the property types and the corresponding controls.

Type	Control
Boolean	Check box (a check mark = checked)
Color	Color selection box, with samples and their RGB equivalents
Enum	Drop-down list
MultiLine	Multiline edit control
Numeric value	Edit box
Text string	Text editing box

Overridden properties

When you override the default value of a property, the property name is displayed in purple. The following figure shows an overridden property.

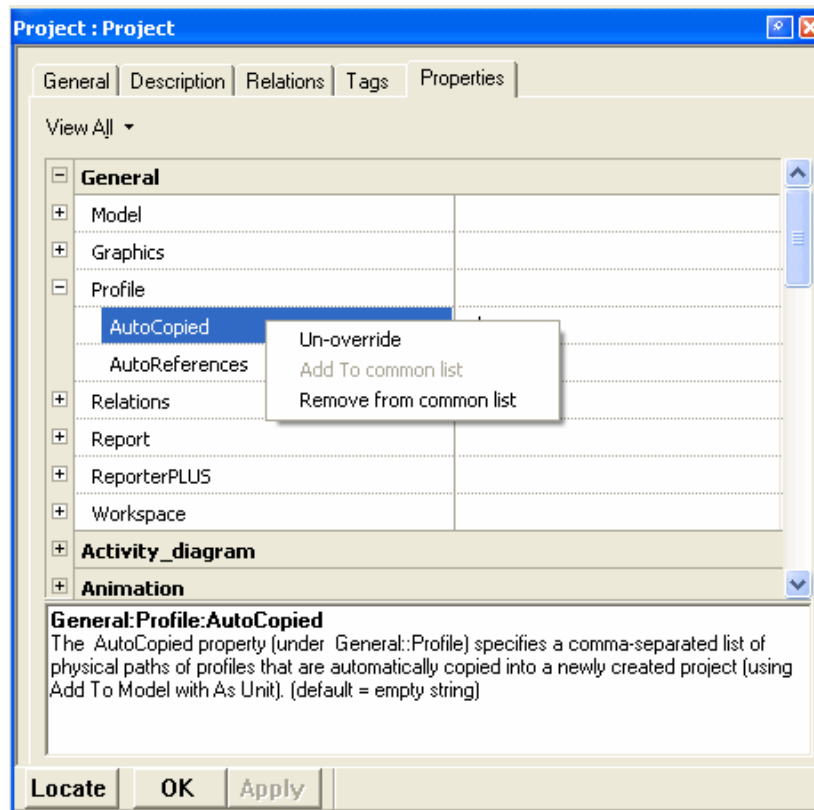


Removing an override

To remove an override:

1. Right-click the property value to display the Un-override command.
2. Click **Un-override**. The property is reset to the default value.
3. Click **OK**.

The following figure shows the **Un-override** command.



Changing a property value

To change the value of a property using the Features window:

1. Choose **File > Project Properties** to set properties at the project level.

or

To set properties at the component level, right-click the component whose property you want to change, choose **Features > Properties**, and then select the **Properties** tab.

2. If wanted, select a different group of properties using the **View** drop-down list.
3. Locate the appropriate property under the subject and metaclass.

For example, to change the class code editor for your model, expand the `General` node in the list of subjects, then expand the `Model` metaclass to locate the `ClassCodeEditor` property.

4. Select the new value for the property in the right-hand column (for example, to change the value of the `ClassCodeEditor` property from `Internal` to `CommandLine`).

The overridden property is displayed in purple.

5. Click **OK**.

Visibility of properties

In general, a subject is displayed for an element if it contains a metaclass that matches the metaclass of the element. The following table lists the exceptions to this rule.

Subjects Visible Only Under the Project	Subjects Visible Under Diagrams and the Project	Subjects Visible Under Only the Configuration/Component and the Project
• General • RTInterface • RoseInterface • Browse • Report • IntelliVisor	• Diagrams • Statechart • ObjectModelGe • SequenceDiagram • UseCaseGe • ComponentDiagram • DeploymentDiagram • Collaboration_Diagram • Activity_diagram	• ConfigurationManagement • ReverseEngineering • CPP_ReverseEngineering

PRP files

Default properties are assigned in the factory and site default files, `factory.prp` and `site.prp`, respectively. These files are located in the `$OMROOT\Share\Properties` directory and provide a way to tune project properties on an individual or site-wide basis without recompiling Rational Rhapsody.

Do not change the `factory.prp` file to make individual site requirements. Instead, change the `site.prp` file for an individual site. Settings in the `site.prp` file will override the settings in the `factory.prp` file. In this way, you can always return to factory default settings in case of mistakes.

Customizing existing properties

You can customize the existing Rational Rhapsody subjects, metaclasses, and/or properties or create new ones. There are many reasons for creating or modifying subjects, metaclasses and/or properties. For example, you might be using an unsupported OS, compiler (configuration), and/or configuration management tool.

When creating a new subject, you can keep existing metaclasses and properties intact. For example, the subjects `OMUContainers`, `OMContainers`, and `STLContainers` are all different subjects which contain the same metaclasses and properties.

Likewise, when creating a new metaclass, you can keep existing subjects and properties intact. You can also create new properties under existing subjects and metaclasses. For example, if you were using a testing tool that Rational Rhapsody did not support, you might create new properties under an existing metaclass.

You can create new metaclasses and properties using existing Rational Rhapsody properties. For example, to add a new configuration management tool to Rational Rhapsody:

1. In the `factory.prp` file, locate the CM tool property.
2. To the comma-separated enum values string, add the name of the new CM tool.
3. If you want this tool to be the default CM tool, change the second quoted string from `None` to the name of the new tool.

When you restart Rational Rhapsody, you will see the name of the new CM tool listed in the drop-down list of the Modify window for the CM tool property.

4. Block and copy the section of code for an existing metaclass. Be sure to include the closing “end” for the metaclass block.
5. Rename the new metaclass to the name you specified in Step 2.

6. Edit the value of every property in the new metaclass, depending on the requirements for CM commands within the individual CM tool.

To do the final step, see the documentation for the CM tool to determine the syntax for commands in that tool. Once you know what information the CM tool requires and the syntax of commands in that tool, you can use regular expression syntax and Rational Rhapsody-internal variables to create the appropriate command strings for the tool.

Note

Do not change the original settings in the `factory.prp` file because you would not be able to roll back to the default settings.

Adding customized properties

You can add your own properties to existing metaclasses. You can add properties for special annotations, specification numbers, part numbers, traceability information, and any kind of comment. For example, you might require that each class be assigned a safety property and a serial number.

To add a custom property:

1. Open the `site.prp` file in the `Properties` directory.
2. Under the appropriate subject and metaclass, add the new property. Make sure to put in the correct number of end statements.

Adding comments to the properties files

Although Rational Rhapsody does not have a formal way to add comments to the property files, you can add comments by creating your own properties.

To add comments to a properties file:

1. Create new subjects, for example:

Subject	.PRP File
Subject SiteComment	site.prp
Subject SiteCPPComment	siteC++.prp
Subject SiteCComment	siteC.prp

2. Create a new metaclass named `Comments` under each subject.
3. To each metaclass, add a new property of type `String` or `MultiLine` that contains the comment text.

If you place this information on top of your `site<Lang>.prp`, you benefit in the following ways:

- ◆ You can add comments in the file header to document why you made changes.
- ◆ Access from inside Rational Rhapsody via the `Property` tab to get an overview of the version and changes inside your site properties files. However, you must keep the comments and content in sync manually.
- ◆ Gain the ability to bring site settings into the `Reporter` documentation.

Note

Do not use the `String` comment ("") inside the assigned strings of the comment properties.

Example

The following example shows a portion of the SiteC++.prp file with comment properties.

```
Subject SiteCPPComment
  Metaclass Comments
    Property RhpVersion String "v7.5 SiteC++ for Rhapsody
      Build 1368921"
    Property ChangeAuthor String "John Smith, Acme Co."
    Property LastChange String "01.30.2009"
    Property ChangeHistory MultiLine "Version 1.0
      02.08.2009"
    Property ChangeList MultiLine "
      List of Changed Properties
      Optimization Properties:
      * CPP_CG->Attribute->AccessorGenerate to False
      * CPP_CG->Attribute->MutatorGenerate to False
      * CPP_CG->Relation->RemoveKeyGenerate to False
      * CPP_CG->Relation->RemoveKeyHelpersGenerate to
        False
      Other properties:
      * None
    "
    Property GeneralComment MultiLine "
      Purpose of the changes in siteC++.prp:
      I like challenges!
      Any questions?
    "
  end
end
end
```

Including PRP files

To include one .prp file in another, use the Include directive. Rational Rhapsody will replace the directive with the contents of the specified file.

The syntax of the directive is as follows:

```
Include "path"
```

The specified path can be relative to the file that does the include, and should include the .prp extension. In addition, the path can include an environment variable. For example:

```
Include "$MY_PATH\some_dir\my_file.prp"
```

To include more than one .prp file, simply use multiple directives. For example:

```
Include "$MY_DIR\my_file1.prp"
```

```
Include "$MY_DIR\my_file2.prp"
```

Note the following information:

- ◆ Include statements must be outside of a Subject block, either before or after. Therefore, Rational Rhapsody expects every included .prp file starts with a Subject line. If not, Rational Rhapsody generates an error.
- ◆ Rational Rhapsody does not check for loops. Therefore, a loop in the include files might cause an infinite loop when the .prp file is read.
- ◆ You can nest include statements. For example:

```
Include "C:\Rhapsody\Share\Properties\IndividualSite.prp"
```

```
Subject General
  Metaclass Model
    Property BackUps Enum "None,One,Two" "Two"
  end
end
```

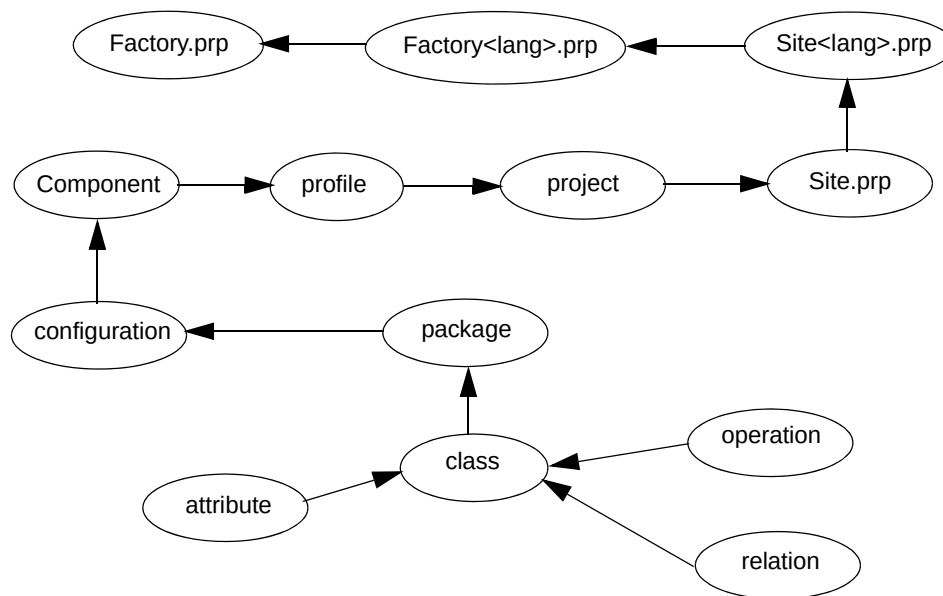
```
Include "..\Properties\IndividualSite.prp"
```

```
Subject General
  Metaclass Model
    Property AutoSaveInterval Int "11"
  end
end
Include "IndividualSite.prp"
```

Property inheritance

The level at which you set a property can affect other elements. For example, if you set a property for a dependency at the class level, and not on an individual dependency, it applies to all the dependencies in that class.

The following illustration shows how property values are inherited.



Note

Note that if a stereotype is applied to an element, a property assigned to that stereotype takes precedence over the element's inherited property values (locally overridden properties take precedence over both inherited properties and those applied via a stereotype).

Concepts used in properties

The following sections provide a brief overview of the concepts used in the Rational Rhapsody properties.

Static architectures

Several properties in Rational Rhapsody provide support for static architectures, found in hard real-time and safety-critical systems that do not use dynamic memory management during runtime. When these properties are used, all events (including timeouts and triggered events) are dynamically allocated during the initialization phase. Once allocated, the memory pool (or event queue) remains static in size during the life of the application. It is important to note that dynamic memory management capabilities are still required in order to initialize these systems. In its current implementation, Rational Rhapsody does not generate applications that can be run in environments that are completely without dynamic memory management capabilities.

Properties that provide support for static architectures include the following properties:

- ◆ `BaseNumberOfInstances`
- ◆ `AdditionalNumberOfInstances`
- ◆ `ProtectStaticMemoryPool`
- ◆ `EmptyMemoryPoolCallback`
- ◆ `EmptyMemoryPoolMessage`
- ◆ `TimerMaxTimeouts`

IncludeFiles

The `IncludeFiles` property (under the `<ContainerTypes>` metaclasses) enables the selective framework includes of templates based on a particular relation implementation.

If this property is defined, includes of the files listed in the property are added to the specification files for classes participating in a relation.

Include files can also be added to class implementation files if the container is added by reference. If the `Containment` property is set to `Reference`, a forward declaration of the container is added to the class specification file, and the `#include` is added to the class implementation file. A new set of properties that describe the forward declaration of the container is added to each container implementation metaclass, and the necessary modifications are made to the code generation.

Selective framework includes

Some compilers (for example, VxWorks) tend to instantiate redundant copies of templates that are defined in the C++ framework. These redundant instantiations cause the resulting code (executable) to be much larger.

To enable the use of relations without templates, a set of typeless (`void*`) containers is supplied as an alternative implementation. The generated code for relations that use the typeless containers is responsible for supplying a type-safe interface.

However, supplying typeless containers does not entirely solve the problem because templates are still included via the framework `.h` files. To resolve this issue, selective includes of framework objects must be used to avoid getting the template definitions “in the back door.”

To support selective framework includes, the `oxf.h` file has been minimized to include only the most basic files. The following properties have also been added:

- ◆ `IncludeFiles`
- ◆ `ActiveIncludeFiles`
- ◆ `ReactiveIncludeFiles`
- ◆ `ProtectedIncludeFiles`
- ◆ `StaticMemoryIncludeFiles`

Reactive classes

A class is considered reactive if it:

- ◆ Has a statechart
- ◆ Consumes events
- ◆ Is a composite

Units of collaboration

In the property descriptions, the term “unit” refers to a unit of collaboration, which can be one of the following types:

- ◆ Object type or class
- ◆ Package (`*.sbs` file)
- ◆ Configuration (`*.cfg` file)
- ◆ Object model diagram (`*.omd` file)
- ◆ Sequence diagram (`*.msc` file)

- ◆ Use case diagram (*.ucd file)

Instances (objects), statecharts, and events are not exchanged in isolation, but together with packages. Therefore, they are not considered units of collaboration.

The Executer

Several Rational Rhapsody properties include calls to the Executer to execute batch files. The location of both the Executer and the target-specific batch makefile (\$makefile) are given relative to the \$OMROOT environment variable.

The commands that reference the Executer do so for two reasons:

- ◆ To allow definition of a single property to represent a series of commands. The Executer executes each one by calling `system()`.
- ◆ To permit execution of commands by means closely resembling those of the shell's command-line (important for wildcards and escape characters).

The Executer accepts two string arguments:

- ◆ An executable command, or list of commands separated by semicolons.
- ◆ The directory from which to run the commands. If not specified, the commands are run from the current directory. (For CM tools, the “current directory” is the `_rpy` directory).

Rational Rhapsody environment variables

In addition to the properties, numerous environment variables help define the Rational Rhapsody environment. These environment variables are stored in the `rhapsody.ini` file, normally located under `C:\winnt` on Windows systems.

The following table lists the environment variables used by Rational Rhapsody. For ease of use, the environment variables are listed by section in the order in which they occur in the file.

Environment Variable	Description
General section	
<code>OMROOT = path</code>	Specifies the location of the <code>Share</code> subdirectory of the Rational Rhapsody installation. For example, if during the installation you specify <code>D:\Rhapsody</code> for the destination folder, the value of <code>OMROOT</code> is as follows: <code>\$OMROOT = D:\Rhapsody\Share</code>
<code>OMDOCR00T = path</code>	Specifies the root directory for some Rational Rhapsody documentation as PDF files.
<code>OMHELPROOT = path</code>	Specifies the root directory for the Rational Rhapsody online help.
<code>RY_LICENSE_FILE</code>	Specifies licensing information needed by <code>FLEXlm</code> . This variable is set to one of the following values: - The path to the <code>license.dat</code> file <code>1717@hostname</code>, where <code>1717</code> is the port number (any number between 1024 and 65534) and <code>hostname</code> is the name of the Rational Rhapsody license server machine
<code>AnimationPortNumber=6423</code>	Specifies the port number used for communicating with the animation server.
<code>UseVBA = Boolean</code>	Specifies whether VBA macros can be used. For example: <code>UseVBA = CHECKED</code>
<code>EnableWebDownload = Boolean</code>	Enables or disables the Download from Web feature. For example: <code>EnableWebDownload=CHECKED</code>
<code>DefaultEdition = edition</code>	Specifies the default edition of Rational Rhapsody to use. For example: <code>DefaultEdition = Developer</code>
<code>DefaultLanguage = language</code>	Specifies the default programming language for Rational Rhapsody. For example: <code>DefaultLanguage = C++</code>

Environment Variable	Description
ImplementBaseClasses=CHECKED	Controls whether the Implement Base Classes window is displayed in implicit requests. By default, this window is displayed only when you explicitly open it. If you select the Automatically show this window check box on the window, Rational Rhapsody writes this line to the <code>rhapsody.ini</code> file. If wanted, you can add this line directly to the <code>rhapsody.ini</code> file to automatically display the window.
RHAPSODY_AFFINITY = <i>number</i>	Sets the affinity of the Rational Rhapsody process. This variable is designed to address cases where Rational Rhapsody has problems with more than one processor. For example, to run Rational Rhapsody on a single processor, add the following line to the <code>rhapsody.ini</code> file: RHAPSODY_AFFINITY=1 A zero value or lack of this variable disables the mechanism.
NO_OUTPUT_WINDOW=CHECKED	Disables the output window for reverse engineering (RE) messages to increase performance. RE messages are logged in the file <code>ReverseEngineering.log</code> .
Helpers section	
name<#>= <i>string</i>	Specifies the name of the helper. For example: name1=Reverse Engineer Ada Source Files
command<#> = <i>path to .exe</i>	Specifies the invocation command for the helper. For example: command1=J:\Rhapsody5\ AdaRevEng\bin\AdaRevEng.exe
initialDir<#> = <i>path</i>	Specifies the initial directory for the helper. For example: initialDir1=J:\Rhapsody5\ AdaRevEng
isVisible<#> = 0 or 1	Specifies whether the helper is visible in the Tools menu. For example: isVisible1=1
isMacro<#> = 0 or 1	Specifies whether the helper is a VBA macro. For example: isMacro1=0
arguments<#> = <i>string</i>	Specifies the command-line arguments for the helper. For example: arguments1=
numberOfElements = <i>number</i>	Specifies the number of helpers. For example: numberOfElements=1

Environment Variable	Description
CodeGen section	
ExternalGenerator = <i>path</i>	Specifies the path to the external generator (if used). For example: ExternalGenerator= J:\Rhapsody5\Sodius\ Launch_Sodius.bat Note that this variable applies only to Rational Rhapsody in Ada.
ModelCodeAssociativityMode = <i>enum</i>	Specifies the dynamic model-code associativity (DMCA) status. A value of Dynamic means that changes to the model are dynamically updated in the code, and vice versa. For example: ModelCodeAssociativityMode= Dynamic
Tip section	
TimeStamp =	Specifies the date and time you ran the Rational Rhapsody installation. For example: TimeStamp=Mon Apr 21 09:34:31 2003
FilePos = <i>position</i>	Specifies the default position at which to display the Tip of the Day. For example: FilePos=3200
StartUp = <i>Boolean</i>	Specifies whether to display the Tip of the Day when you start Rational Rhapsody. For example: StartUp = 1
Animation section	
ViewCallStack = <i>0 or 1</i>	Specifies whether the call stack should be visible in the next animation session. For example: ViewCallStack=0
ViewEventQueue = <i>0 or 1</i>	Specifies whether the event queue should be visible in the next animation session. For example: ViewEventQueue=0
Settings section	
WindowPos = <i>position</i>	Specifies the position of the Rational Rhapsody window on your screen. For example: WindowPos=0, 2, -32000, -32000, -1, -1, 25, 38, 926, 669

Environment Variable	Description
BarsLayout section	
BrowserVisible = <i>Boolean</i>	Specifies whether the browser should be visible, according to the settings from the last session. For example: BrowserVisible=TRUE
FeaturesVisible = <i>Boolean</i>	Specifies whether the Features window should be visible, according to the settings from the last session. For example: FeaturesVisible=FALSE
FeaturesFloating = <i>Boolean</i>	Specifies whether the Features window should be floating or docked, according to the settings from the last session. For example: FeaturesFloating=TRUE
BrowserFloating = <i>Boolean</i>	Specifies whether the browser should be floating or docked, according to the settings from the last session. For example: BrowserFloating=FALSE
Bar<#>	Groups the settings corresponding to each toolbar. For example: [BarsLayout-Bar29]
BarsLayout-Summary section	
Bars = <i>number</i>	Specifies the number of toolbars. For example: Bars=30
ScreenCX = <i>resolution</i>	Specifies the user screen resolution on the X scale. For example: ScreenCX=1024
ScreenCY = <i>resolution</i>	Specifies the user screen resolution on the Y scale. For example: ScreenCY=768
Plugin section	
MTT<Version number> = <i>path</i>	Specifies the path to the TestConductor DLL. For example: MTT4.1=L:\Rhapsody\v41\ TestConductor\ TestConductor.dll
Tornado section	
DefaultTargetServerName = <i>string</i>	Specifies the default target-server name used with Tornado.

Environment Variable	Description
RecentFilesList section	
File<#> = <i>path</i>	Lists the .rpy files that have been loaded recently. The maximum number of files listed is four. For example: <pre>File1=J:\Rhapsody5\ProjectAda\ NewFunc\NewFunc.rpy File2=J:\Rhapsody5\CPPProjects\ NewFunc\NewStuff\NewStuff.rpy File3=J:\Rhapsody41MR2\AdaProject\ Dishwasher\Dishwasher\Dishwasher.rpy</pre>

Format properties

Rational Rhapsody uses properties under the Subject *Format* to determine the format used for displaying various graphical elements.

These properties do not appear on the **Properties** tab of the Features window, but you can control these formatting features using the Format window that is displayed when you select **Format** that displays on the context menu for graphical elements. This window lets you set formatting options up to the project level.

In some cases, you might want to set formatting options across multiple projects. This can be done by overriding the value of formatting properties using the `site.prp` file.

These are the formatting properties that can be used.

- ◆ `DefaultSize` - specifies the default size to use for graphical elements of this type. You can change the default size for elements of a given type by selecting the *New Element Size* check box in the Make Default window. In the value that is used for this property, the third coordinate represents the width of the graphical element, and the fourth coordinate represents the height of the element. For more information, see [Defining default characteristics](#).
- ◆ `Fill.FillColor` - specifies the color to use to fill the background of the graphical element. Corresponds to the *Fill Color* selector on the **Fill** tab of the Format window. For more information, see [Defining line characteristics](#).
- ◆ `Fill.Transparent_Fill` - used to specify whether or not the fill should be transparent. Corresponds to the *Transparent Pattern* check box on the **Fill** tab of the Format window.
- ◆ `Fill.BackgroundColor` - used as the color of the superimposed pattern if you have chosen a pattern to use for the fill. Corresponds to the *Pattern Color* selector on the **Fill** tab of the Format window.
- ◆ `Fill.FillStyle` and `Fill.FillHatch` - represent the fill pattern to use. Correspond to the *Pattern* list on the **Fill** tab of the Format window.
- ◆ `Font.Font` - specifies the font to use for the text on the graphical element. Corresponds to the font list on the **Font** tab of the Format window.
- ◆ `Font.FontColor` - specifies the color of the font to use for the text on the graphical element. Corresponds to the *Text Color* selector on the **Font** tab of the Format window.
- ◆ `Font.Size` - specifies the size of the font to use for the text on the graphical element. Corresponds to the font size list on the **Font** tab of the Format window.
- ◆ `Font.Underline` - specifies whether or not the text on the graphical element should be underlined. Corresponds to the *Underline* check box on the **Font** tab of the Format window.

- ◆ `Font.Strikeout` - specifies whether strikethrough text should be used for the text on the graphical element. Corresponds to the *Strike-Out* check box on the **Font** tab of the Format window.
- ◆ `Font.Weight` - used for bolding of text on the graphical element. Corresponds to the bold/italic control on the **Font** tab of the Format window.
- ◆ `Font.Italic` - used for italicizing text on the graphical element. Corresponds to the bold/italic control on the **Font** tab of the Format window.
- ◆ `Line.LineColor` - specifies the color to use for the outline of the graphical element. Corresponds to the *Color* selector on the **Line** tab of the Format window.
- ◆ `Line.LineStyle` - specifies the style to use for the outline of the graphical element, for example, solid or dotted. Corresponds to the *Style* list on the **Line** tab of the Format window.
- ◆ `Line.LineWidth` - specifies the width of the line to use for the outline of the graphical element. Corresponds to the *Width* list on the **Line** tab of the Format window.

Defining default characteristics

To set default characteristics for an element type based on an existing element:

1. Right-click the element in a diagram that you want to use as the element pattern and select **Make Default**.
2. In the Make Default window, select the **Item** characteristics to be copied from the highlighted element to other elements of the same type:
 - ◆ Display Options
 - ◆ Format
 - ◆ New Element Size (for graphical elements only)
3. Select the **Level** in your project for all of the elements of that type to have the same characteristics:
 - ◆ Diagram
 - ◆ Package
 - ◆ Project
4. Click **OK**.

Defining line characteristics

To define the graphical appearance of a line in a diagram:

1. Right-click the line and select **Format**.
2. In the Format window on the Line tab, select the **Line** color, style, and width.
3. On the Font tab, select the font name and other text characteristics.

Rational Rhapsody projects

A Rational Rhapsody project includes the UML diagrams, packages, and code generation configurations that specify the model and the code generated from it. The term *project* is equivalent to *model* in Rational Rhapsody.

This section provides an overview of a Rational Rhapsody project, the components of a project, and the procedures to create, edit, and store projects.

Project elements

A project consists of *elements* that define your model, such as packages, classes and diagrams. The browser displays these elements in a hierarchical structure that matches the organization of your model. Rational Rhapsody uses these elements to generate code for your final application.

A Rational Rhapsody project has the following top-level elements:

- ♦ **Components** contain configurations and files and also hold the variants for different software product lines.
- ♦ **Packages and profiles** packages contain other packages, components, actors, use cases, classes (C++/J), object types, events, types (C/C++), functions (C,C++), objects, dependencies, constraints, variables, sequence diagrams, OMDs, collaboration diagrams, UCDs, and deployment diagrams.

A profile “hosts” domain-specific tags and stereotypes.

- ♦ **UML diagrams** For example, use case, object model, sequence, collaboration, component, and deployment diagrams.

For more information about project elements, see [Model elements](#).

Creating and managing projects

When working in Rational Rhapsody, you need to know the basic procedures for using a project. You can create a new project, or work with an existing project. You can edit your project and save the changes. You can create an archive of your project to easily exchange files with another engineer or with customer support. You can have Rational Rhapsody create autosave files to periodically store your unsaved changes, and automatically create backup projects of previously saved versions. This section contains instructions for these and other procedures necessary for using Rational Rhapsody.

Creating a project

When you create a new project, Rational Rhapsody creates a folder containing the project files in the location you specify. The name you choose for your new project is used to name project files and folders, and displays at the top level of the project hierarchy in the browser. Rational Rhapsody provides several default elements in the new project to get you started, such as a default component and configuration. Before you begin, create a project folder in your file system to hold all of your Rational Rhapsody projects.

1. With Rational Rhapsody running, create the new project by either selecting **File > New**, or clicking the **New project** button on the main toolbar.
2. Replace the default project name (Project) with <your project name> in the **Project name** field. Enter a new directory name in the **In folder** field or Browse to find an existing directory.
3. The Default **Project Type** provides all of the basic UML structures for most Rational Rhapsody projects. However, you can select one of the specialized [Profiles](#), that provide a predefined, coherent set of tags, stereotypes, and constraints for specific project types.
4. You might want to select one of the **Project Settings**. For example, you might want to work in a code centric setting instead of the default model centered setting.
5. Click **OK**. If the directory does not exist, Rational Rhapsody asks if you want to create it. Click **Yes** to create the new project directory.

Rational Rhapsody creates a new project in the <your project name> subdirectory and opens the new project. The project name in that directory is <your project name>.rpy.

Profiles

A predefined Rational Rhapsody profile becomes part of your project in one of these ways:

- ♦ You select an available profile from the Type pull-down menu, as described in the [Creating a project](#) section.
- ♦ You manually add a specialized profile to your project from the Share\Profiles directory, as described in the [Adding a Rational Rhapsody profile manually](#) section.
- ♦ Rational Rhapsody assigns a starting-point profile based on your project settings and development environment.

The predefined profiles available to you depend on the system language and add-on products licensed for Rational Rhapsody.

- ♦ **AdaCodeGeneration** is the default Ada code generation profile.
- ♦ **AutomotiveC** includes the capabilities provided in the following automotive industry environments:
 - FixedPoint arithmetic
 - MainLoop (no-OS)
 - OSEK21 and Basic/Extended OSEK task stereotypes
 - MicroC

[The AutomotiveC profile](#) also loads the StateMateBlock and SimulinkInC profiles. For more information, see [The Rational Rhapsody automotive industry tools](#).

- ♦ **AUTOSAR_21** and **AUTOSAR_31** create automotive components in accordance with the AUTOSAR development process using the ECU, Internal Behavior, SW Component, System, and Topology diagrams. For more information, see [AUTOSAR modeling](#). The separate AUTOSAR profiles support the related AUTOSAR versions (2.1 and 3.1).

Note: AUTOSAR can only be used in C language projects.

- ♦ **CodeCentricCpp** provides software developers an environment to work on C++ application code rather than development using models.
- ♦ **Default** provides all of the basic UML structures for most Rational Rhapsody projects.
- ♦ **DoDAF** is the Rational Rhapsody profile for DoDAF v1.0. For more information, see [Rational Rhapsody for DoDAF Add On and profile](#).
- ♦ **FixedPoint** profile contains predefined types representing 8, 16, and 32-bit fixed-point variables: FXP_8Bit_T, FXP_16Bit_T, FXP_32Bit_T. For more information, see [Defining fixed-point variables](#).
- ♦ **FunctionalC** profile tailors *Rhapsody in C* for the C coder, allowing the user to functionally model an application using familiar constructs such as files, functions, call graphs and flow charts.

- ◆ **Harmony** creates a project based on the Harmony (SE) Systems Engineering Process. For more information, see [Harmony process and toolkit](#).
- ◆ **IDFProfile** uses the code generation settings for the Rational Rhapsody Developer for C IDF. For more information, see [Using IDF for a Rational Rhapsody in C project](#).
- ◆ **MARTE** supports **M**odel and **A**nalysis for **R**eal-Time **E**mbded systems that are not covered by UML and annotates application models to support analysis by tools.
- ◆ **MODAF** is the Rational Rhapsody profile for MODAF v1.1. For more information, see [IBM Rational Rhapsody MODAF Add On](#).
- ◆ **MicroC** provides the facilities to run automotive C applications on systems with very limited resources or with no operating system. For more information, see [The MicroC profile](#).
- ◆ **MISRA98** controls the code generation settings to comply with the MISRA-C 1998 standard.
- ◆ **NetCentric** imports Web-services Definition Language (WSDL) files to design and generate a services model. For more information, see [Domain-specific projects and the NetCentric profile](#). (This profile requires a separate license.)
- ◆ **RespectProfile** can be used for C and C++ project to preserve the structure of the code and preserves this structure when code is regenerated from the Rational Rhapsody model. Meaning that code generated in Rational Rhapsody resembles the original. For more information about handling regenerated code, see [Code respect and reverse engineering for Rational Rhapsody Developer for C and C++](#).
- ◆ **RoseSkin** is used by Rational Rose Import to set format and other settings to resemble Rational Rose look-and-feel.
- ◆ **SDL** facilitates importing SDL Suite models into Rational Rhapsody SDL Blocks.
- ◆ **Simulink** and **SimulinkInC** allow integration of MATLAB Simulink models into Rational Rhapsody as Simulink Blocks (*Simulink* profile is for C++).
- ◆ **SPARK** is the Rational Rhapsody Ada SPARK profile.
- ◆ **SPT** (Scheduling, Performance, and Time) is an implementation of the SPT standard (OMG standard) that specifies the method to add timing analysis data to model elements. For more information, see [Schedulability, Performance, and Time \(SPT\) profile](#).
- ◆ **StatemateBlock** creates a new block/class allowing a Statemate model to become part of a Rational Rhapsody architecture. This profile is only available for Rational Rhapsody in C and requires a licensed version of Statemate 4.2 MR2 or greater with a license for the Statemate MicroC code generator. For more information, see [StatemateBlock in Rational Rhapsody](#).
- ◆ **SysML** supports both UML and SysML model diagrams for systems engineering. This profile is the Rational Rhapsody implementation of the OMG SysML profile. For more information, see [Systems engineering with Rational Rhapsody](#).

- ♦ **TestingProfile** is an implementation of the OMG Testing Profile. The TestingProfile is for use with Rational Rhapsody TestConductor. For more information about this profile, see the third-party documentation provided for Rational Rhapsody TestConductor.
- ♦ **UPDM_L0** and **UPDM_L1** are the Rational Rhapsody implementation of the OMG UPDM L0 and L1 profiles that combine the MODAF and DoDAF profiles.

Opening an existing Rational Rhapsody project

To open an existing Rational Rhapsody project with all units loaded:

1. Choose **File > Open**. The Open window displays.
Note: To open one of the last-opened projects, select it from the list of projects that appear on the **File** menu just above the **Exit** command.
2. In the **Look in** field, browse to the location of the project.
3. Select the .rpy file, or type the name of the project file in the **File name** field.
4. Select the **With All Subunits** check box. This causes Rational Rhapsody to load all units in the project, ignoring workspace information. For information on workspaces, see [Using workspaces](#).
5. Click **Open**. The entire Rational Rhapsody project opens.

Search and replace facility

Engineers and developers can use the Rational Rhapsody Search and Replace facility for simple search operations and to manage large projects and expedite collaboration work. The search results display in the [Output window](#) with the other tabbed information.

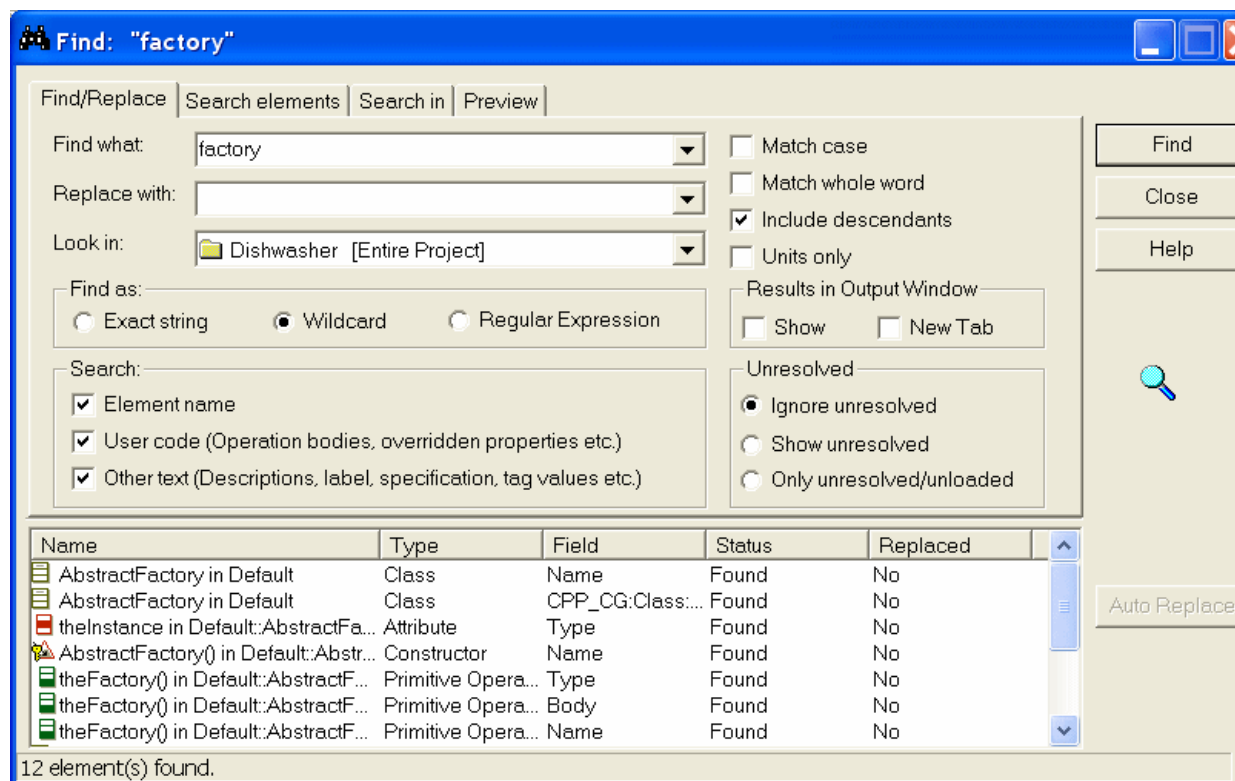
This facility provides the following capabilities:

- ♦ Perform quick searches
- ♦ Locate unresolved elements in a model
- ♦ Locate unloaded elements in a model
- ♦ Identify only the units in the model
- ♦ Search for both unresolved elements and unresolved units
- ♦ Perform simple operations on the search results
- ♦ Create a new tab in the Output window to display another set of search results
- ♦ For more detailed instructions for the Search and Replace facility, see [Searching in the model](#).

Searching models

To search models:

1. With the model displayed in Rational Rhapsody, there are three methods to launch the Search facility: select **Edit > Search**, click the binoculars button , or press **Ctrl+F**.
2. The Search window and perform a quick search. Type the search criteria into the **Find what** field and click **Find**. The results display in the Output window. The search criteria displays on the **Search** tab of the Output window.
3. To display the more detailed search window, select **Edit > Advanced Search Replace** or click the **Advanced** button in the Search window (above). Both methods display this window. The advanced search window provides the Unresolved and Units only search features.
4. Select the **Search elements** tab to narrow the scope of the search to specific project element types such as requirements, classes, packages, components, and diagram types.
5. Select the **Search in** tab to identify parts of the project for the search. Of course, you can use combinations of selections on different tabs and the **Find** window to narrow your search.



6. The advanced search also includes these capabilities:
 - ♦ **Exact string** permits a non-regular expression search. When selected the search looks for the string entered into the search field (such as char*)
 - ♦ **Wildcard** permits wildcard characters in the search field such as “*” produces results during the search operation that include additional characters. For example, the search *dishwasher matches class dishwasher and attribute itsdishwasher.
 - ♦ **Regular Expression** allows the use of Unix style regular expressions. For example, itsdishwasher can be located using the search term *dishwasher.
7. If after performing one search you want another **Search** tab with additional search results displayed in the Output window, check the **New Tab** box in the **Results in Output Window** area. Perform the next search.

Search results

After locating elements using the Search facility, you can perform these operations in the Search window or in the Output window:

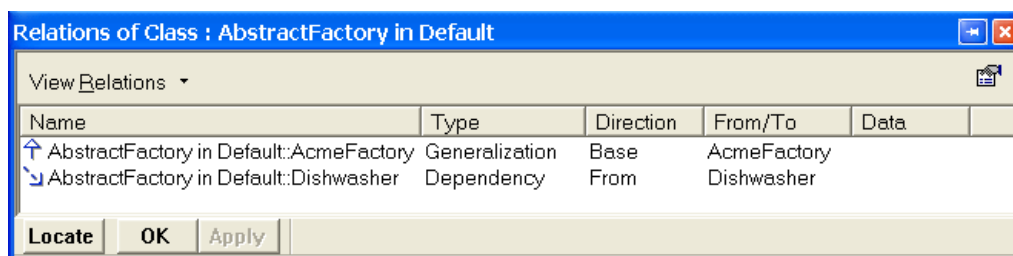
- ♦ Sort items
- ♦ Check the references for each item
- ♦ Delete
- ♦ Load

To sort items in the list, click the heading above the column to sort according to display that feature of each item in the list.

Examining references in search results

To examine the *references* for an item in the search results:

1. Right-click an item in the search results list to display the menu.
2. Select **References** and examine the information displayed in the window, as shown in this example:



Deleting located items

To delete an item located in search process:

1. Right-click an item in the search results list to display and then select **Delete from Model**. The system displays a message for you to confirm the deletion.
2. Click **Yes** to complete the deletion process.

If you have located an unloaded item in the search results and want to load it into the model, right-click the item. Load the item in the same manner as it is loaded from within the browser.

Replacing

If you want to replace item names or other terminology throughout the model:

1. Display the **Advanced Search**.
2. Enter the current terminology in the **Find what** field.
3. Enter the new terminology into the **Replace with** field.
4. Make any additional selections to limit the search and replace process.
5. Click **Find** and approve or skip the possible replacements.

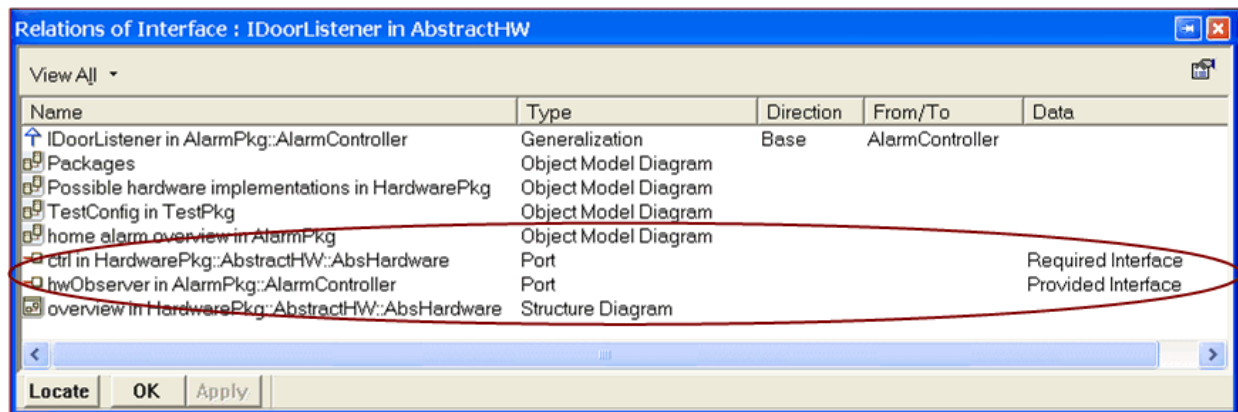
Locating and listing specific items in a model

During development you might need to examine a specific feature or create a list of items in the model. This is one of the situations when the Rational Rhapsody [Application accelerators](#) keys are useful.

In this example, the developer wants a list of the ports that are used by a specific interface.

To display a list of specific items in a Rational Rhapsody model:

1. Display the model in Rational Rhapsody.
2. If you need to locate the section you want to examine, you can use [Search and replace facility](#) to find it. Then in the browser, select the item for which you need more information.
3. In this example, press the **Ctrl-R** (for relationships) accelerator keys to list the relations with the IDoorListener interface.
4. You can sort the displayed list by the **Name** or **Type** by clicking the heading of the column. In this example, the items are sorted by Type to show the ports grouped together.



Note

You can also use ReporterPLUS to generate a report for a selected section of your model.

File menu commands

The File menu provides file management capabilities for the entire project. You can use the File menu to create a New project, Open an existing Rational Rhapsody project, Save, Close files, Print, and Exit the project as these features are typically used in Windows applications. The Rational Rhapsody File menu also contains special project management features to insert other files and projects and check files in and out of your configuration management (CM) system. These features are also accessible from the [Standard tools](#) icons.

File > New

This menu command creates a new Rational Rhapsody project. See [Creating a project](#).

File > Add to Model

Use this File menu command to insert an element into the active project.

File > Add Java API Library

Use this File menu command to import a Java API as a library that you want to use within the project.

File > Configuration Items

Use this File menu command to launch Rational Rhapsody access to your configuration management (CM) system to check files in and out and perform other CM operations.

File > Compare

If you have launched the Rational Rhapsody DiffMerge tool, the Compare menu command is available from the DiffMerge File menu. This menu command allows you to browse and select files to compare. You can compare files, list all of the differences, and merge the files if wanted.

Editing and changing a project

You build projects in Rational Rhapsody by creating and defining elements using either the browser or the graphic editors. Elements include components, packages, classes, operations, events, diagrams, and so on.

Adding elements

To add elements from the browser:

1. Right-click an element, then select **Add New**. A submenu that lists all the elements that can be added at the current location in the project hierarchy is displayed.
2. Select the element you want to add.

For detailed information about adding elements from the browser, see [Model elements](#). To add elements from a graphic editor, use the drawing tools to draw the element in the diagram. For information on using the graphic editors, see [Graphic editors](#).

Adding a Rational Rhapsody profile manually

To add another predefined Rational Rhapsody profile manually to your project:

1. Open the existing project.
2. Choose **File > Add Profile to Model**.
3. The Add to Profile to Model window lists all of your available profiles, either as separate .sbs files or in folders.
4. Select a .sbs file for the profile and click **Open**. The system checks to be certain that the selected profile is compatible with the language being used in the existing project.

Rational Rhapsody lists the newly added profile in the Profiles section of the browser. If you are not familiar with the profile, open the profile in the browser to examine its characteristics.

Editing in the Features window

Each element in the project has features that can be edited using the Features window. The features for an element include things like its name, description, type, and implementation code.

- ◆ Double-click the element.
- ◆ Right-click the element and then select **Features**.
- ◆ Select an element in the browser, then type **Alt + Enter**.

For more information on using the Features window, see [The Features window](#).

Undo and redo

Rational Rhapsody allows you to undo the last 20 operations, and to redo the operation that was most recently undone.

To undo the last operation, do one of the following actions:

- ◆ Choose **Edit > Undo**.
- ◆ Click the **Undo** tool.

To redo an operation, do one of the following actions:

- ◆ Choose **Edit > Redo**.
- ◆ Click the **Redo** tool.

The **Undo** menu command does not become active until you perform at least one operation that can be undone. Similarly, the **Redo** menu command is not active until you have used the **Undo** command at least once.

By default, Rational Rhapsody allows you to undo the last 20 operations, but you can set this value in the `General::Model::UndoBufferSize` property. Setting this property to a value of zero disables the **Undo/Redo** feature.

You cannot use the **Undo** command after large operations that affect the file system. The undo operation buffer is cleared and the **Undo** and **Redo** tools are deactivated. The following operations cannot be undone:

- ◆ Saving, opening, or closing a project
- ◆ Automatic diagram layout
- ◆ Roundtripping code with the “generated code in browser” option
- ◆ Loading a unit into a workspace
- ◆ Configuration management operations
- ◆ Importing Rational Rose models
- ◆ Reverse engineering

- ◆ Adding a file unit to the model (via the **File > Add to Model** command)
- ◆ Code generation with the “generated code in browser” option

Using IDF for a Rational Rhapsody in C project

For some systems developed using Rational Rhapsody in C, the OXF provided with Rational Rhapsody is not appropriate because the systems require a solution with a smaller footprint. To provide a solution for these environments, a limited framework called IDF (Interrupt-Driven Framework) is also provided with Rational Rhapsody.

Rational Rhapsody provides a base IDF model that can be adapted for different target systems. Also included is a sample adaptation for *Microsoft NT* illustrating the use of the IDF.

To use this sample model to learn about the IDF:

1. Start the development version of Rational Rhapsody in C and choose **File > Open** and browse to locate the IDF model `Share\LangC\idf\model\idf.rpy`.

2. With that project open, click the GMR button  to generate and make the generic configuration automatically displayed above the window, as shown here.



This generates all core files and `idfFiles.list` with dependencies and rules in the `Share\LangC\idf` directory.

3. Using the same method, open another IDF model, `Share\LangC\idf\Adapters\Microsoft\MicrosoftNT`.
4. Generate and make this model with the GMR button. This builds the library `msidf.lib` in the directory `Share\LangC\lib`.

Saving a project

You can save a project in its current location using **File > Save** or click the **Save** button in the toolbar. The **Save** command saves all modified files in the project repository.

Saving a project in a new location

To save the project to a new location with **Save As**:

1. Select **File > Save As**. The Save As window opens.
2. Use the **Save In** field to locate the folder where you would like to save the project.

3. Type a name for the project file in the window. The file extension `.rpy` (for *repository*) denotes that the file is a Rational Rhapsody model.
4. Click **Save**. All project files are saved in the project repository.

Incremental save

If you want to save only the modified project units and not the entire project:

1. Select **File > Project Properties** and select the **Properties** tab.
2. Navigate to the `General::Model::UseIncrementalSave` property. This property should be checked to use the incremental save feature.

Autosave

By default, Rational Rhapsody performs an incremental autosave every 10 minutes to back up changes made between saves. Modified units are saved in the autosave folder, along with any units that have an older time stamp than the project file. Modifications to the project file are saved in the `<Project>_auto.rpy` file. When you save the project, the autosave files are cleared.

The autosave feature saves files in flat format. All unit files reside in the `<Project>_auto_rpy` folder, regardless of the directory structure of the original model.

To change the autosave interval, use the `General::Model::AutoSaveInterval` property. The interval is measured in minutes. To disable the autosave feature, set the `AutoSaveInterval` property to 0.

Renaming a project

To change the name or location of the project, use the **Save As** command. Do not attempt to edit the project file directly. When you save the project under a different name, the name of the project folder is updated in the browser.

Refactoring or renaming in the user code

To rename an element that is in the user code:

1. In the browser, right-click the element to be renamed and select **Refactor > Rename**.
2. Enter the **New Name** and click **Continue**.
3. Use the Find window to the element name.

Previewing the rename changes

Before the name changes are applied, the Rename Preview lists all of the references in the user code that will be renamed.

1. Select entries in the lower section of the preview window to see the related code displayed above.
2. To remove a reference from the list of changes, click the check box before the reference to remove the green check.
3. To accept the checked changes, click **OK**.

Examining the renamed elements

After accepting the previewed changes, the rename text displays in the selected browser element. To check the accuracy of the change:

1. Expand the browser tree to see the change to the related elements.
2. Display the Features window for the element and check the information to be certain it has been changed as you expected.

Closing all diagrams

To close all the diagrams opened for a project, choose **Window > Close All**.

Closing a project

1. Select **File > Close**.
2. If you have unsaved changes, Rational Rhapsody asks if you would like to save your changes before closing the project. Select one of the following options:
 - ♦ **Yes** saves the changes.
 - ♦ **No** discards the changes.
 - ♦ **Cancel** cancels the operation and return to Rational Rhapsody.

Closing Rational Rhapsody

To exit from Rational Rhapsody, select **File > Exit**.

Creating and loading backup projects

Rational Rhapsody can create backups of your model every time you save your project, allowing you to revert to a previously saved version if you encounter a problem. To use the automatic backup feature, set the `General::Model::BackUps` property to the number of backup projects you want Rational Rhapsody to create. The options are **None** (default), **One**, and **Two**. Leave the default value of **None** if you do not want Rational Rhapsody to create backups.

To set up automatic backups for your project:

1. In the browser, right-click the <project name> at the top of the browser list and select **Features**.
2. On the **Properties** tab, click the **All** radio button to display all of the properties for this project.
3. Expand the **General** and **Model** property lists and locate the **BackUps** property.
4. Select **One** or **Two** from the pull-down menu. With this setting, Rational Rhapsody creates up to one or two backups of every project in the project directory.
5. Click **OK**.

After this change, saving a project more than once creates <projectname>_bak2.rpy contains the most recent backup and the previous version in <projectname>_bak1.rpy. To *restore* an earlier version of a project, you can open either of these backup files.

1. Open the <Project>_bak2.rpy (or the <Project>_bak1.rpy file) in Rational Rhapsody. Do not try to rename the backup file directly.
2. Save the project as <Project>.rpy using the **File > Save As** command.

Archiving a project

At times, you might need to archive a project to send it to another developer or to Customer Support. To create a complete archive, include the following files and directories:

- ♦ <Project>.rpy file
- ♦ <Project>_rpy directory with all subdirectories
- ♦ <Component> directories
- ♦ Any external source files (.h and .cpp) needed to compile the project

Table and matrix views of data

Rational Rhapsody provides these additional methods to view model data:

- ♦ *Table view* performs a query on a selected element type and displays a detailed list of its various attributes and relations.
- ♦ *Matrix view* displays queries showing the relations among selected model elements.

These views provide the following development capabilities:

- ♦ Define and run dynamic queries of model content
- ♦ Provide easy requirements display and analysis
- ♦ Produce exportable and printable tables and data lists

Basic method to create views from layouts

To create either the table or matrix view, follow these general steps:

1. Select any area of the model in the Rational Rhapsody browser where you want to store a table or matrix layout (query design).
2. Define a *layout* for the table or matrix (as described in the following sections) and save it in the selected browser location.
3. Define a *view* of the model data using the previously defined layout. This view also gives you the opportunity to define the *scope* of the query for the view.
4. In the browser, double-click the defined view to display the results of the query in the drawing area.
5. To edit the data displayed in a view, use the Features window for the view and click the **Refresh**  button to update the view.

Note: Data displayed in views cannot be edited directly in a view. You must use the Features window for a view to make your edits.

6. To export the data, right-click the data in the drawing area and select **Copy**.

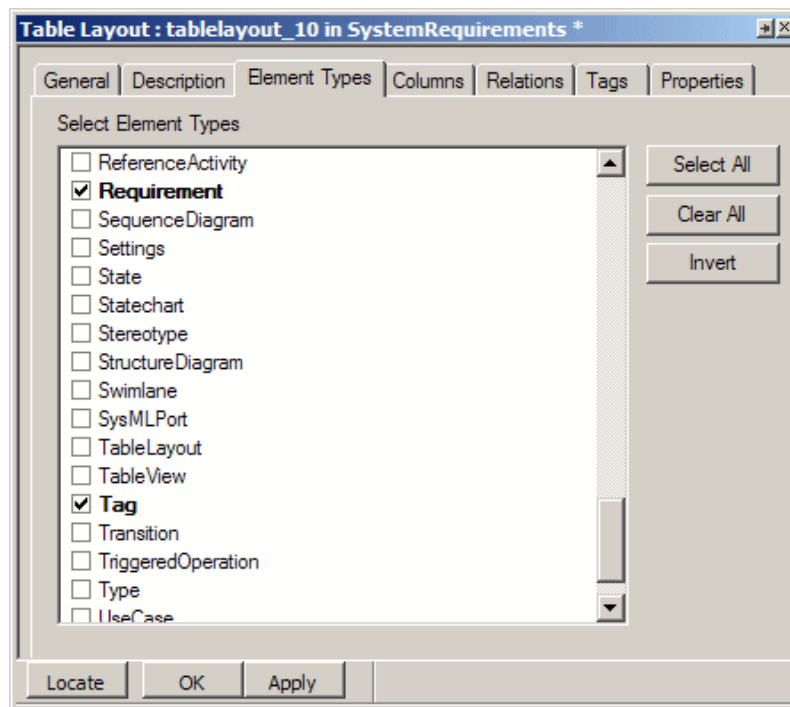
Creating a table layout

To design the structure for your model elements query as a table layout:

1. Right-click the package in the Rational Rhapsody browser where you want to create and store your table layout and select **Add New > TableMatrix > Table Layout**.

Note: **Add New > TableMatrix** is the default menu command structure in Rational Rhapsody. It can be changed by users. This topic assumes that all defaults are in place.

2. In the browser, enter a name for this table design. You might want to include the word “layout” in the name to help identify your defined layouts from their generated views.
3. Double-click the new layout in the browser to open its Features window.
4. On the **Element Types** tab, select the element types you want to be displayed in the table:

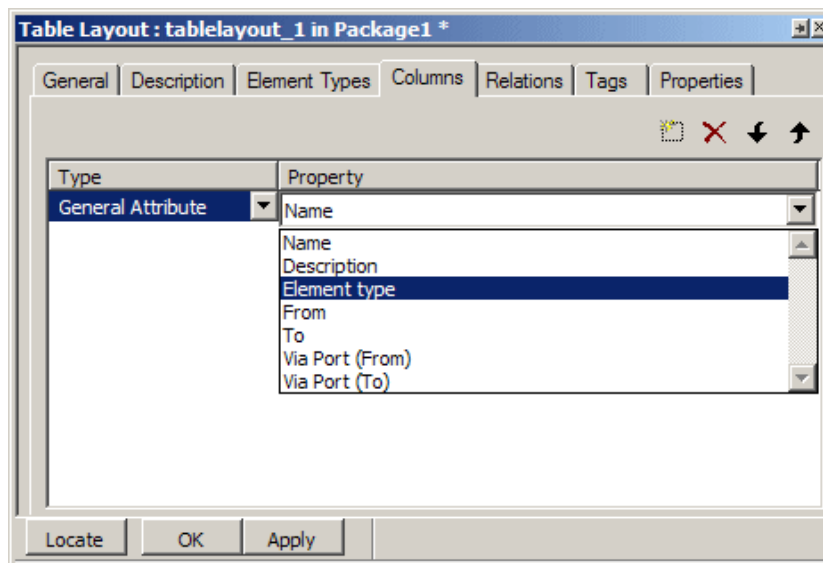


5. Click **Apply** to save your selections without closing the Features window.

Adding a new row to the table layout

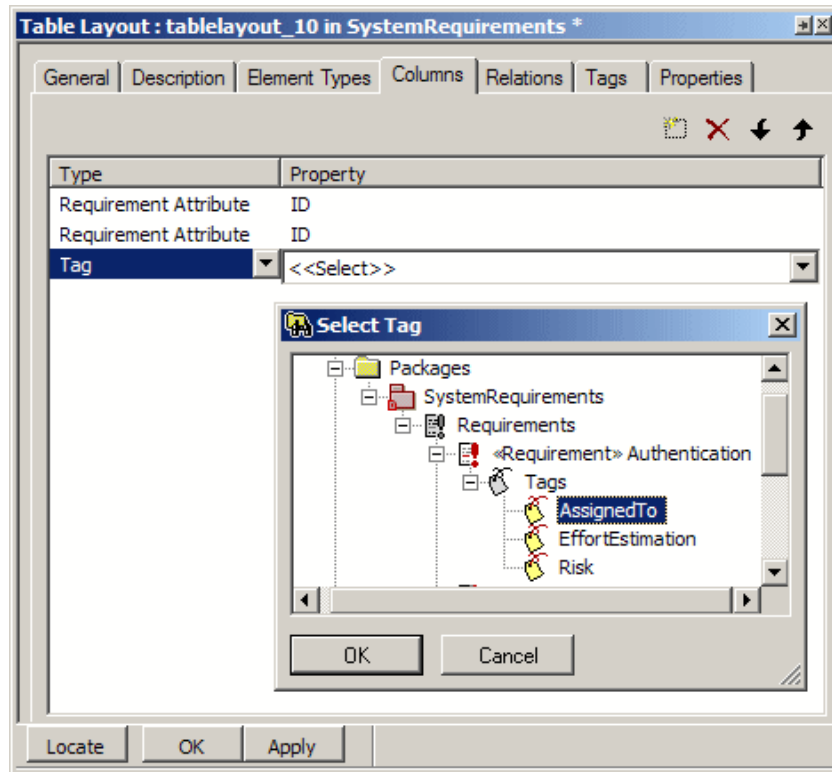
After [Creating a table layout](#), you can begin to create the table design.

1. On the **Columns** tab, click the **New**  button to create a new row in the table layout.
2. For the row, select a **Type** and **Property** from the corresponding menus for your query purpose:
 - ♦ The **General Attribute** type might use one of the following properties (also shown in the following figure) to define it:
 - **Name** displays the name of an element.
 - **Description** displays the description for an element (if there is one).
 - **Element type** displays the element type of an element.
 - **From** displays where an element is from.
 - **To** displays where an element goes to.
 - **Via Port (From)** displays the port from which a relation is connected. Typically used along with **Via Port (To)**. For more information, see [Including ports and multiple relations](#).
 - **Via Port (To)** displays the port to which a relation is connected. Typically used along with **Via Port (From)**. For more information, see [Including ports and multiple relations](#).



- ♦ The **Requirement Attribute** type might have either an **ID** or **Specification** property.
- ♦ The **Flow Attribute** type might have only **Flow items** as its property.

- ♦ The **Tag** type has the <<Select>> property. Click it to open the Select Tag window in which you can identify the information for the tag. Click **OK**.



3. Click the **New**  button to add each row for your table. Use the Move Item Up and Move Item Down buttons   to arrange the order of the rows in your table layout.
4. Optionally, to remove a row from the layout, select it on the **Columns** tab and click the Delete button .
5. When you have completed your design layout, click **OK**.

Creating a table view

After you have created one or more table layouts, to generate a table view of the model element data based on your layout design:

1. Right-click a package on the Rational Rhapsody browser to which you want to add a table view and select **Add New > Table\Matrix > Table View**.

Note: **Add New > Table\Matrix** is the default menu command structure in Rational Rhapsody. It can be changed by users. This topic assumes that all defaults are in place.

2. In the browser, right-click the new table view and select **Features** to open its Features window.
3. Enter a **Name** for the view.
4. Select the name of a previously created table layout from the **Layout** drop-down list.
5. Select the scope for this view by selecting a package from the **Scope** drop-down list.
6. Clear or select the **Include Descendants** check box if you want to exclude or include the descendants for the selected scope. For more information, see [Including and excluding descendants](#).
7. Click **OK** to save your table view.
8. To make changes to the displayed data, double-click a row in the table view and make the changes in the Features window for that element.
9. If you have finished making changes, you can use either of the following actions:
 - ♦ Click **Apply** to save your changes in the Features window and then click the **Refresh**  button to display the new data in the view.
 - ♦ If you are done making changes, click **OK**.

Changing the layout for a generated table

You can also return to the table layout and make design changes. To redisplay the data in the changed table view, click **Refresh**.

Including and excluding descendants

By default, table and matrix views include descendants in their scope because the **Include Descendants** check box on the **General** tab of the table or matrix view's Features window is selected by default. For example, when you produce the table view whose scope is the main package and both packages have objects, the default the view shows all descendants.

To exclude descendants from appearing in your table or matrix view, you need to clear the **Include Descendants** check box on the **General** tab of the Features window for the particular view, as shown in the following figure for a table view. Then (refresh the view if necessary), the view shows without descendants.

Analyzing data in the table

In the browser, double-click the table view name to generate the results of the data query. The query analyzes data for the selected package and all of its nested packages.

Adding elements to a table

After creating a table from the table layout, you might want to add new element type to the table.

1. On the left of the table display, click the **Add model element**  icon.
2. In the Add new element window, select the **Element type** from the pull-down menu.
3. Select the **Location** from the pull-down menu.
4. Select the **number of elements**.
5. Click **OK**.

The revised table displays the new element automatically.

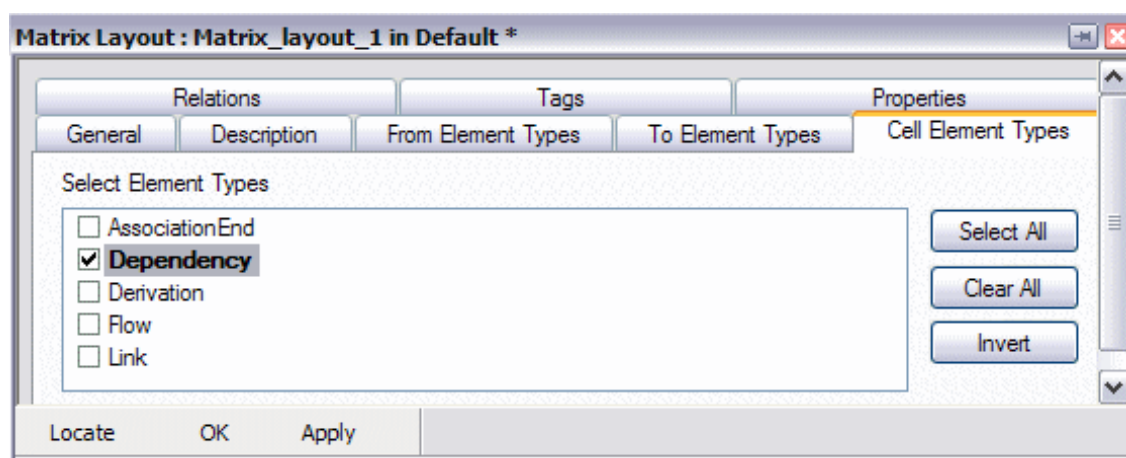
Creating a matrix layout

To create a matrix layout to analyze the relationships among model elements:

1. Right-click a package on the Rational Rhapsody browser where you want to add a matrix layout and select **Add New > Table\Matrix > Matrix Layout**.

Note: **Add New > Table\Matrix** is the default menu command structure in Rational Rhapsody. It can be changed by users. This topic assumes that all defaults are in place.

2. In the browser, enter a name for this new matrix design.
You might want to include the word “layout” in the name to help identify your defined layouts from their generated views.
3. Double-click the new matrix layout to open its Features window.
4. On the **Cell Element Types** tab select only one of the possible element types. *Cell element type* is the relation type between the “From” and “To” elements to be displayed in matrix cells. In the finished matrix, the elements of that type are displayed down the left side of the matrix view to identify a row of data.



5. Click **Apply** to save your selections and keep the Features window open to select the “From” and “To” elements.

Selecting the element types for the matrix layout

After selecting the Cell Element Types in the Features window for the matrix layout, define the basic “from” and “to” structure for the matrix:

1. On the **From Element Types** tab, select the element types from which the connection should be identified.
2. Click **Apply** to save your selections.
3. On the **To Element Types** tab, select the element types to which the connection should be identified.
4. Click **OK**.

Modifying the matrix layout

You can return to the matrix layout and make changes to the design even after the data has been generated in the matrix view. To redisplay the data in a modified view, use the **Refresh** icon on the matrix view.

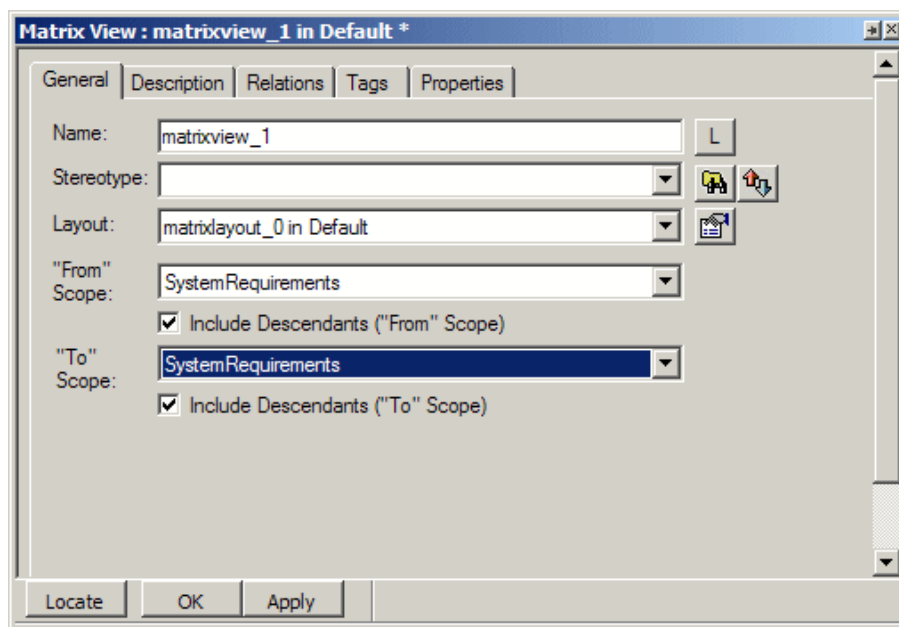
Creating a matrix view

To create a matrix view to analyze the attributes of the model elements you identified in the matrix layout:

1. Right-click the package on the Rational Rhapsody browser to which you want to create and store a matrix view and select **Add New > Table\Matrix > Matrix View**.

Note: **Add New > Table\Matrix** is the default menu command structure in Rational Rhapsody. It can be changed by users. This topic assumes that all defaults are in place.

2. In the browser, right-click the new matrix view and select **Features** to open its Features window.
3. Enter a **Name** for the new matrix view.
4. Select the name of a previously created matrix layout from the **Layout** drop-down list.
5. Select the scope for this view by selecting a package or project in the **“From” Scope** and in the **“To” Scope** drop-down lists, as shown in this example:



6. Clear or select the **Include Descendants** check box if you want to exclude or include the descendants for the selected scope. For more information, see [including and excluding descendants](#).
7. Click **OK** to save your matrix view.

8. In the browser, double-click the matrix view name to generate the results of the data query. The query analyzes data for the selected package and all of its nested packages, as shown in this example:

To: Actor, Object, Package, UseCase		Scope: SystemUseCases
	 Login	 Login Via Web Client
 Non_Authorized_user	 Login	 Login Via Web Client
 Read_Only_User	 Login	 Login Via Web Client
 Read_Write_User	 Login	 Login Via Web Client
 Login Via Web Client	 Login	

9. To make changes to the displayed data, double-click a cell in the matrix view and make the changes in the Features window for that element.
10. Depending on if you are done making changes:
- Click **Apply** to save your changes in the Features window and then click the **Refresh**  button to display the new data in the view.
 - If you are done making changes, click **OK**.

Filtering out rows and columns without data

If you want to remove the rows in a matrix view that do not contain data:

1. Open the Features window for the view or choose **File > Project Properties** to display the properties for your project. For information about using properties in your project, see [Properties](#).
2. On the **Properties** tab, locate the `Model::MatrixView::HideEmptyRowsCols` property and select its check box.
3. Click **OK**.
4. When you display a view, click the **Toggle empty rows filter**  button to show only the rows containing data.

If you select the `Model::MatrixView::HideCellNames` property, the content of a cell is displayed only with an icon.

Including ports and multiple relations

By default, the table view and matrix view show relations between objects (even if ports are involved) and display multiple relations between elements if they exist. This means that you can easily communicate your design information in a tabular or matrix format.

The `Model::TableLayout::ShowContainerElementForPorts` property controls this capability. By default, this property is set to **Checked**. This means that for table views, if ports are encountered when searching for connections of relations, Rational Rhapsody displays the port's parent element instead of the port itself. For matrix views, Rational Rhapsody automatically looks at child ports of elements and finds the relations to other elements.

If you do not want this property to be active, you would clear the check box for it.

To make changes to this property:

1. Open the Features window for the table view or matrix view.
2. On the **Properties** tab, locate the `Model::TableLayout::ShowContainerElementForPorts` property.
3. Clear or select the check box next to the property name.
 - ◆ Clear it to disable this property.
 - ◆ Select it if you want to show relations between objects (even if ports are involved) and display multiple relations between elements if they exist. This is the default.
4. Click **OK**.

Note

When there are multiple relations, options such as **Features**, **Locate**, and so on are disabled.

Adding elements to a matrix

After creating a matrix from the matrix layout, you might want to add new element type.

1. On the left of the matrix display, click the **Add model element**  icon.
2. In the Add new element window, select whether you want the element selection to be **Based on row settings** or **Based on column settings**.
3. Select the new **Element type** from the pull-down menu.
4. Select the **Location** from the pull-down menu.
5. Select the **number of elements**.
6. Click **OK**.

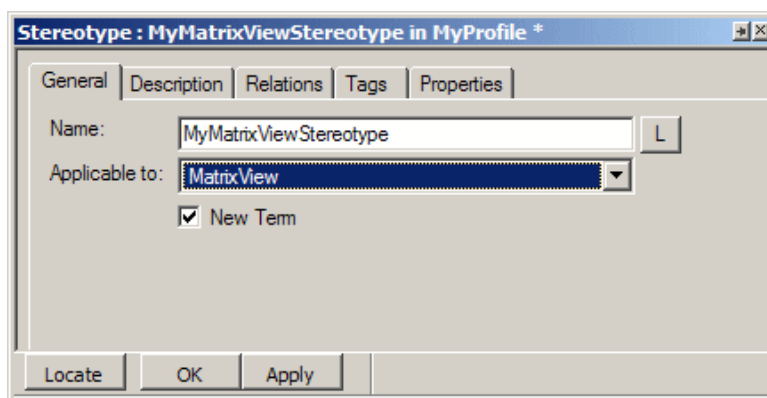
The revised matrix displays the new element type's information automatically.

Setting up an initial layout for table and matrix views

You can bind a view and layout for a table or matrix through the use of a New Term stereotype that has the `Model::Stereotype::InitialLayoutForTables` property set for this purpose for a Rational Rhapsody profile. Once you apply the stereotype to a table or matrix layout, it allows you to set the initial layout for a table or matrix view.

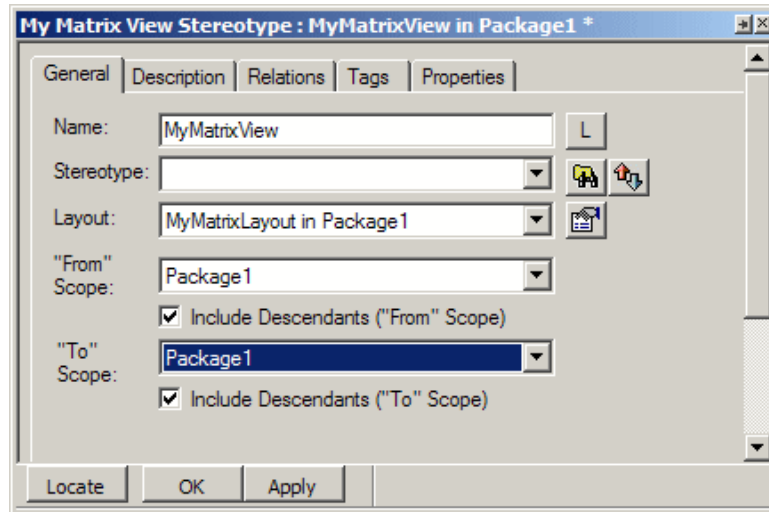
To apply a stereotype to a table or matrix layout for a profile:

1. Create a profile. For more information, see [Creating a customized profile](#).
2. Create a table or matrix layout and design it as you want. See [Creating a table layout](#) and [Creating a matrix layout](#).
3. In your profile, create a stereotype and define it as a New Term and to what it is applicable to (table view or matrix view, as shown in the following figure) and click **Apply** to save but not close the window. For information on stereotypes, see [Stereotypes](#).

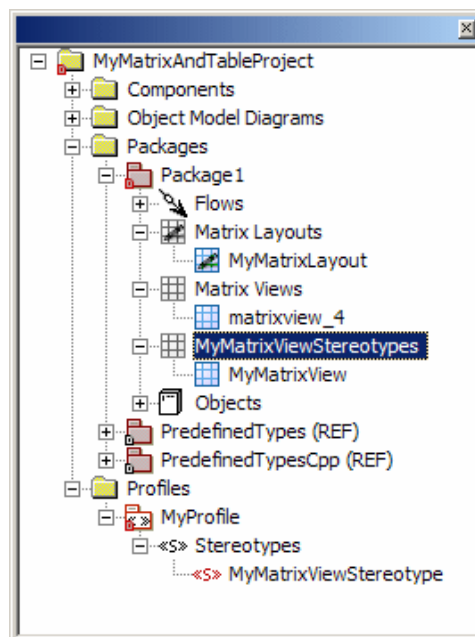


4. On the Properties tab, locate the `Model::Stereotype::InitialLayoutForTables` property and enter the name of the layout you created in Step 2 and click **OK**.
5. Create a table or matrix view through the profile, select **Add New > [name of profile] > [matrix view]**.
6. Open the Features window for the table or matrix view you created in the previous step and note that the layout is already identified (because of the applicable stereotype).

7. Complete your design of the view by filling in the **Scope** boxes, as shown here, and click **OK**.



8. Look at the Rational Rhapsody browser and notice that the view you just created is listed in a View category (in our example, **MyMatrixViewStereotypes**, as shown in the following figure) other than the general **Matrix Views** category, which only lists those views that were created without a stereotype (using **Add New > Table/Matrix > Matrix View**):



9. When you double-click the table or matrix view (**MyMatrixView** in our example above), it shows the query results from the layout that has the specified stereotype.

Managing table or matrix data

After creating a table or matrix, you have additional options to manage a view and its data. Right-click the view in the drawing area to display a menu with these standard Rational Rhapsody options:

- ◆ **Features** to open the Features window. For more information, see [The Features window](#).
- ◆ **Locate** to find the location of the element on the Rational Rhapsody browser.
- ◆ **Add to Favorites** to add to your favorites list in Rational Rhapsody. For more information, see [The Favorites browser](#).
- ◆ **Browse from Here** to open a Rational Rhapsody browser that contains a more-focused Rational Rhapsody browser from whichever point you are at. For more information, see [Opening a Browse From Here browser](#).

The remaining options are standard Windows options. You might find the **Copy** option particularly useful to export the view's data into Microsoft Excel and then save it as a .csv file.

The Rational Rhapsody specialized editions

In addition to the Rational Rhapsody Developer edition that supports projects in C, C++, Java, and Ada, Rational Rhapsody also provides three editions to create specialized projects:

- ◆ Rational Rhapsody Designer for Systems Engineers edition
- ◆ Rational Rhapsody Architect for Systems Engineers edition
- ◆ Rational Rhapsody Architect for Software edition

For information about installing or switching between these editions, see the Rational Rhapsody installation instructions.

Creating projects in Rational Rhapsody Designer for Systems Engineers

To create a project in Rational Rhapsody Designer for Systems Engineers:

1. With Rational Rhapsody Designer for Systems Engineer running, create the new project by either selecting **File > New** or clicking the **New project** button on the main toolbar.
2. Replace the default project name (Project) with <your project name> in the **Project name** field. Enter a new directory name in the **In folder** field or Browse to find an existing directory.
3. The Default **Project Type** provides all of the basic Rational Rhapsody Designer for Systems Engineers edition features. However, you can select one of these specialized [Profiles](#) to supply a predefined, coherent set of tags, stereotypes, and constraints for specific project types:
 - ◆ DoDAF
 - ◆ FunctionalC
 - ◆ Harmony
 - ◆ MODAF
 - ◆ SPARK
 - ◆ SysML
 - ◆ TestingProfile
4. You might want to select one of the **Project Settings**. For example, you might want to work in a code centric environment instead of the default model centered setting.
5. Click **OK**. If the directory does not exist, Rational Rhapsody asks if you want to create it. Click **Yes** to create the new project directory.

The default Designer for Systems Engineers edition project contains a structure diagram as the starting point. For more detailed instructions, see the [Structure diagrams](#) section.

Note

Remember that Designer for Systems Engineers can use any of the supported languages: Ada, C, C++, or Java.

You can add the following profiles to a Designer for Systems Engineers project using the [Adding a Rational Rhapsody profile manually](#) procedure:

- ◆ AdaCodeGeneration
- ◆ [Backward compatibility profiles](#)
- ◆ JavaDoc
- ◆ NetCentric
- ◆ SDL
- ◆ Simulink
- ◆ SPT

For ideas for your project, examine the sample projects in following <Rational Rhapsody installation path>\Samples:

- ◆ Ada, C, C++, or Java Samples
- ◆ Extensibility Samples
- ◆ JavaAPI Samples
- ◆ System Samples

Creating projects in Rational Rhapsody Architect for Systems Engineers

Rational Rhapsody Architect for Systems Engineers does not include code generation or reverse engineering. You use the same method to create projects as you do for Developer projects, but a specialized project structure is created for systems engineers.

To create an Rational Rhapsody Architect for Systems Engineers project:

1. With Rational Rhapsody Architect for Systems Engineers edition running, create the new project by either selecting **File > New** or clicking the **New project** button on the main toolbar.
2. Replace the default project name (Project) with <your project name> in the **Project name** field. Enter a new directory name in the **In folder** field or Browse to find an existing directory.
3. The Default **Project Type** provides all of the basic Rational Rhapsody Architect for Systems Engineers edition features. However, you can select one of this limited set of specialized [Profiles](#):
 - ♦ DoDAF
 - ♦ FunctionalC
 - ♦ Harmony
 - ♦ MODAF
 - ♦ NetCentric
 - ♦ SysML
4. You might want to select one of the **Project Settings**. For example, you might want to work in a code centric environment instead of the default model centered setting.
5. Click **OK**. If the directory does not exist, Rational Rhapsody asks if you want to create it. Click **Yes** to create the new project directory.

The basic default Architect for Systems Engineers edition project contains a structure diagram as the starting point. For detailed instructions, see the [Structure diagrams](#) section.

Note

Remember that Rational Rhapsody Architect for Systems Engineers edition does not use any of the four development languages available in other Rational Rhapsody editions.

Creating projects in Rational Rhapsody Architect for Software

Rational Rhapsody Architect for Software edition allows you to create a C, C++, or Java project and also allows you to generate code frames for the project from a code generation menu. However, the Architect edition does not support animation or code generation for ports and statecharts. Many architects create a Rational Rhapsody Architect project and reverse engineer code to use as a starting point. If a build environment is wanted for the project, the architect can use Eclipse or another IDE.

To create a Rational Rhapsody Architect for Software project:

1. With Rational Rhapsody Architect for Software edition running, create the new project by either selecting **File > New** or clicking the **New project** button on the main toolbar.
2. Select the *primary implementation language* as C, C++, or Java and click **Next**.
3. In the New Project window, replace the default project name (Project) with <your project name> in the **Project name** field. Enter a new directory name in the **In folder** field or Browse to find an existing directory.
4. The Default **Project Type** provides all of the basic Rational Rhapsody Architect for Software edition features. However, you can select one of the specialized [Profiles](#), that provide a predefined, coherent set of tags, stereotypes, and constraints for specific project types. The following profiles available for a new Architect project from the Type selection:
 - ♦ CodeCentricCpp
 - ♦ FunctionalC
 - ♦ Harmony
 - ♦ NetCentric
 - ♦ SysML

You can also add the available profiles to Rational Rhapsody Architect for Software project using the [Adding a Rational Rhapsody profile manually](#) procedure.

5. You might want to select one of the **Project Settings**. For example, you might want to work in a code centric environment instead of the default model centered setting.
6. Click **OK**. If the directory does not exist, Rational Rhapsody asks if you want to create it. Click **Yes** to create the new project directory.

For ideas for your project, you can examine the sample projects in these <Rational Rhapsody installation path>\Samples directories.

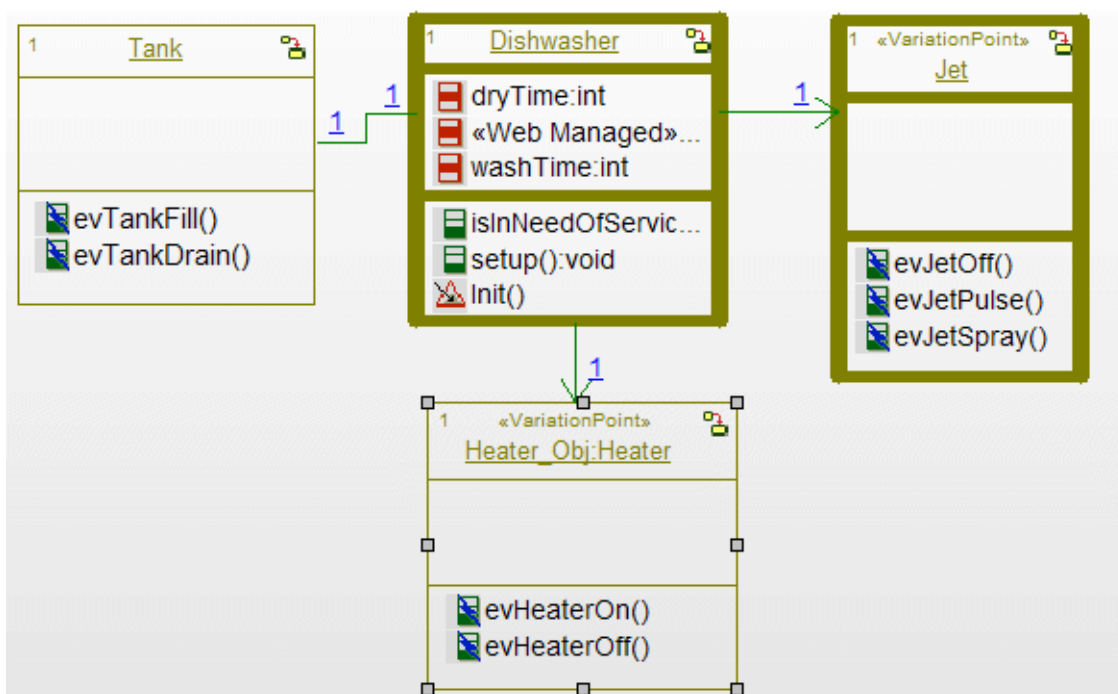
Components with variants for software product lines

If your company uses variations of the software to create a software product line, component variants can manage the reuse of common components and generate code for the product line.

Creating variation points

To create a software product line, identify the Rational Rhapsody components that are the variation points for the system.

1. In a high-level object model diagram, create a class or object for the variable component.
2. In the Features window for each variable component, select **VariationPoint** as the **Stereotype**. In this example, the jet and heater are the variation points for the dishwasher.

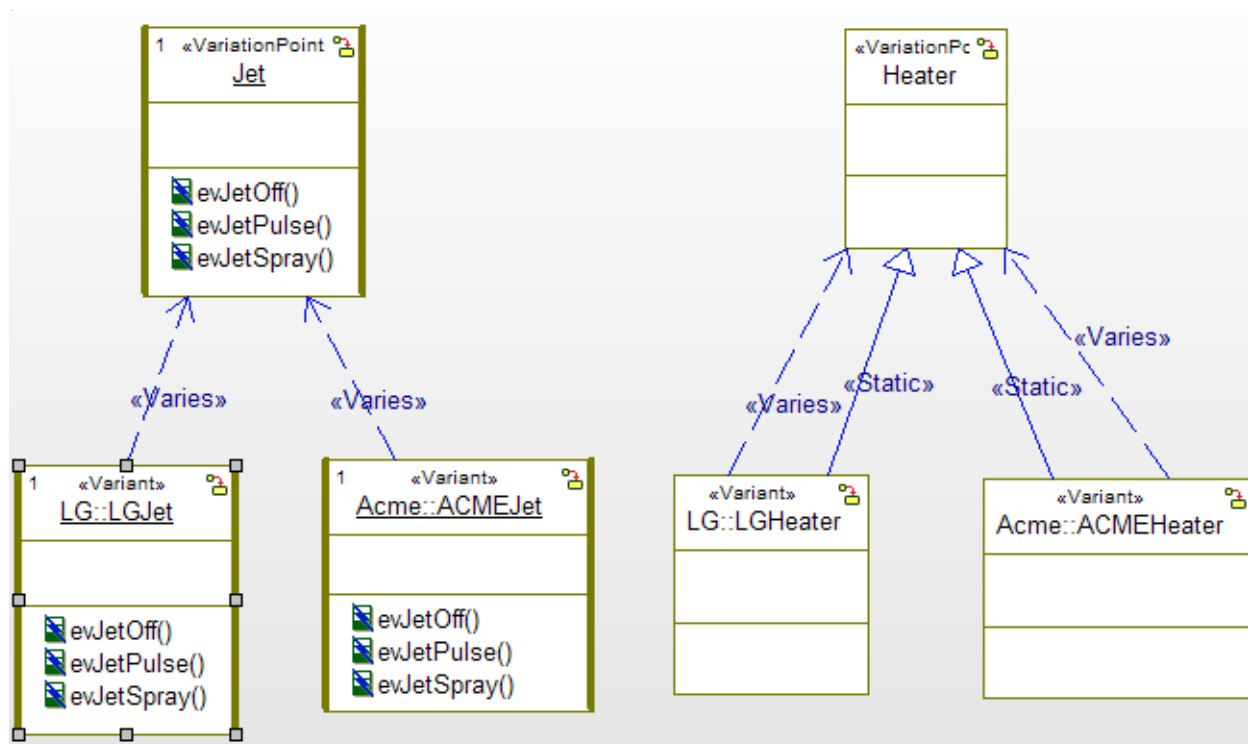


Defining variants

To define one variant for each variation point:

1. For each variation point, draw its dependencies to the variant class or object.
2. For each variant class or object, display the Features window, and select **Variant** as the **Stereotype**.
3. Complete the definition of the variant classes and objects to meet the requirements.

In this example, each variation point has two variants. Note that each of the classes and objects contains a statechart to define the operation states.



The <<Static>> stereotype (on inheritance) duplicates all features of the static base class into the static derive class including types, operations, attributes, associations, inheritances, ports, dependencies, behavior and links.

Selecting a variant

To select a variant:

1. Highlight the component.
2. Display the Features window and click the **Variation Points** tab.
3. Select the variant from the list.

Generating code for software variations

When code is generated, each component generates the specific mapping from the variation points to their selected variants.

Multiple projects

Rational Rhapsody allows you to have more than one project open at a time. When you have more than one project open, you can use the Rational Rhapsody browser to copy and move elements from one project to another.

The following terms are used in Rational Rhapsody in the context of working with multiple projects:

- ♦ *Projects* is the top-level container (or folder) for all open projects in a Rational Rhapsody session (saved as an .rpl file). The **Projects** folder contains the list of projects, which can be saved and reopened.
- ♦ *Active Project* is the project that you can currently be modify. This is also the project to which Rational Rhapsody commands, such as for code generation, are applied (unless the command opens a window that allows you to specify which project to use).

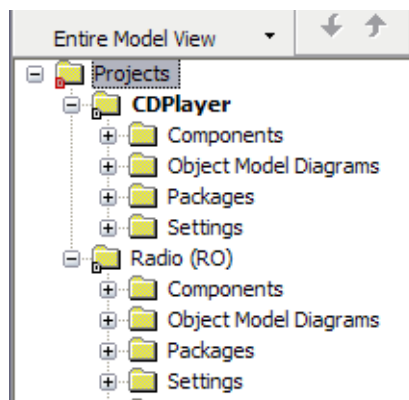
Inserting an existing project

To insert an existing project into an open project:

1. Open the existing Rational Rhapsody project into which you want to add another Rational Rhapsody project.
2. Choose **File > Insert Project**.
3. Select **Existing** to insert a previously created Rational Rhapsody project for a directory.

Note: Rational Rhapsody does not permit two projects with the same name to be incorporated.

The Rational Rhapsody browser displays a project list node called **Projects** that is one level above the open projects. The currently active project name is in bold print, as shown in the following example.



Inserting a new project

To insert a new project into an open project:

1. Open the existing Rational Rhapsody project into which you want to add another Rational Rhapsody project.
2. Choose **File > Insert Project > New** to open the window for [Creating a project](#)

Note: Rational Rhapsody does not permit two projects with the same name to be incorporated.

Setting the active project

When you select **File > Insert Project**, the project you select automatically becomes the active project.

Once you have more than one project open, you can make any project the active project as follows:

1. Right-click the project name in the Rational Rhapsody browser.
2. Select **Set as Active Project**.

Similar to the display of the active component and active configuration, the name of the active project displays in bold in the browser.

You can modify an active project's model elements only. (Rational Rhapsody displays R0, for read-only, next to all project names in the browser that are in non-active projects.)

If you make a project active without first saving changes made to the previously active project, the system asks you if want to save those changes. If you click the **No** button, your changes to the previously active project are not yet lost. This is because when you close all the projects, the system asks if you want to save the project list and all its projects. If you click the **Yes** button, all changes made to all projects that have not been saved are saved at this time.

In general, Rational Rhapsody commands are applied only to the active project. However, the commands **Search** and **Locate in Browser** can be applied across all open projects.

Copy and reference elements among projects

When you have more than one project open, you can use the Rational Rhapsody browser to copy elements from one project to another using either of these methods:

- ♦ **Referencing** an element in another project only “links” that element to the original element in the original project
- ♦ **Copying** creates a element in another project that is the same as the original

Note

Copying and referencing can only be done within the browser. You cannot drag an element from one project to a diagram in another project.

You can use either the standard Windows copying techniques, as described in [Copying elements to other projects](#) or the Shift key method, as described in [Using the Shift key to copy, reference, or move elements](#).

Creating references

Only elements that have been saved as units can be referenced in other projects.

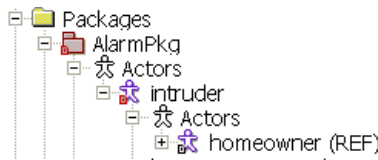
To create a saved unit from an element:

1. In the active project, right-click the element in the browser and select **Create Unit**.
2. Type the **Unit Filename** if you want to use a different name from the displayed name.
3. Click **OK** to create the unit and the icon of the new unit is marked with a red box. In this example, the homeowner and intruder elements are units.



4. In the project that is going to receive the reference, right-click the project name and select **Set as Active Project**.
5. In the browser of the now active project, right-click the element that needs to reference the new unit and then select **Create Unit** to change the receiving element into a unit.

6. In the original project, select the unit that is being referenced. Click and drag that unit to the active project's new unit. When the reference is established, the (REF) symbol displays next to the referenced element, as shown in this example:



Copying elements to other projects

To copy an element to another project:

1. In the browser, right-click the project name in the Rational Rhapsody browser from which you are copying an element and select **Set as Active Project**.
2. Select the element you want to copy into the other project and select **Edit > Copy** or right-click and select **Copy**.
3. Right-click the project name to which you are copying the element and select **Set as Active Project**.
4. Select the folder where you want to store the copied element and select **Edit > Paste**. If the copied element has the same name as an existing element in the new project, the name is appended with “_copy.”

Copied elements with the same name as existing elements should be renamed to avoid confusion.

Using the Shift key to copy, reference, or move elements

To use the pop-up menu to copy, reference, or move a unit:

1. Be certain that the element has been saved as a unit, as described in [Creating references](#).
2. In the non-active project, press and hold the **Shift** key, and click-and-drag the unit you want to copy, reference, or move to the target project.
3. On the pop-up menu that displays, select the applicable command:
 - ♦ **Copy here**
 - ♦ **Reference here, or**
 - ♦ **Move here - leave a reference**

Moving elements among projects

To move an element from one project to another:

1. If the element to be moved is not a saved unit, save it as a unit.
2. Press and hold the **Alt** key, then click-and-drag the element to the other project.
3. Rational Rhapsody checks if there will be unresolved elements in the target project as a result of the move. If there are unresolved elements, a message box displays along with the Output window.
4. If you click **Yes** on the message box, the unit moves to the target project.

Note

The original unit, that was moved, now has a (REF) tag in the source project because the unit has been moved and the moved unit is now the unit of record in the active project.

Closing all open projects

To close all of the projects that are currently open, select **File > Close**.

Managing project lists

When a number of projects are open at the same time, you can save the list of projects as a Rational Rhapsody project list (.rpl) file.

Saving projects in a project list file

To save the project list, select **File > Save All**.

A new .rpl file is created in the current active project folder. The name of the project list file will be `Projects.rpl`. (If a file with this name already exists in the folder, a number will be added to the end of the file name, for example `Project1.rpl`.)

The current active project is saved as an attribute of the project list, so when you reopen a project list, the active project will be the project that was active when the project list was last saved.

Opening a project file list

After a project file list has been saved, you can re-open all the projects in a project list as follows:

1. Open the Open window, choose **File > Open**.
2. If necessary, look in the path for your project, then select the relevant project list file. Typically, the Open window displays all .rpy and .rpl files.
3. Click **Open**.

Adding a project to a project list file

If you open a project list file, its contents are updated each time you select **Save All**. Therefore, to add another project to a project list:

1. Select **File > Insert Project** and choose the project to add.
2. Decide if the just added project or another one should be the current active project.
3. Select **File > Save All**.

Removing a project from a project list file

To remove a project from a list of files:

1. In the Rational Rhapsody browser, select the project to remove. Note that you cannot remove the current active project.
2. Press the **Delete** key on your keyboard.
3. Click **Yes** to confirm your action.
4. Select **File > Save All**.

Project limitations

The following items identify multiple project work and display limitations.

New project

You cannot add a new project to a list of open projects. When you select **File > New**, if there are changes to be saved, you will be asked if you want to save before closing, then all open projects will be closed, and the project list will be saved before the New Project window displays.

Placement of GUI elements

Information regarding the placement of elements such as toolbars and the browser window are stored in the `rhapsody.ini` file, and are, therefore, uniform for all projects and project lists. Information regarding the placement of elements such as windows are stored in a project's workspace file (`.rpw`). Therefore, these elements will change, depending on which project in the list is currently the active project.

References window

The References window includes the references for all of the projects in the project list, and not just those for the active project.

DiffMerge

DiffMerge does not support the comparison of project lists (that is, groups of projects).

Configuration management

Rational Rhapsody does not support the configuration management of project lists but you can use your configuration management tool directly for a project file list.

Properties

When viewing properties, Rational Rhapsody always displays the property values of the selected project. However, for all properties that affect more than one project, Rational Rhapsody uses the settings of the active project. For information about using properties in your project, see [Properties](#).

Components and configurations

When you select a different project to be the active project, the **Code** toolbar displays the active component and configuration for that project. The list of components/configurations available is also updated accordingly.

VBA editor and the active project

When you open the Rational Rhapsody VBA Interface editor, you see only the items belonging to the active project.

Naming conventions and guidelines

To assist all members of your team in understanding the purpose of individual items in the model, it is a good idea to define naming conventions. These conventions help team members to read the diagram quickly and remember the model element names easily.

Note: Remember that usually the names used in the Rational Rhapsody models are going to be automatically written into the generated code. Therefore, the names should be simple and clearly label all of the elements, and they should not use any special characters.

Guidelines for naming model elements

The names of the model elements should follow these guidelines:

- ◆ Class names begin with an upper case letter, such as `System`.
- ◆ Operations and attributes begin with lower case letters, such as `restartSystem`.
- ◆ Upper case letters separate concatenated words, such as `checkStatus`.
- ◆ The same name should not be used for different elements in the model because it will cause code generation problems. For example, you should not have a class, an interface, and a package with the same name of `Dishwasher`.
- ◆ Note the following about special characters:
 - Do not include special characters in an element's *name* if the element is used for code generation.
 - You can use special characters in the *labels* for model elements. See [Descriptive labels for elements](#).
 - The following elements are the only ones for which you can include special characters: Dependencies, Stereotypes, Flows, Links, Configurations, Table layouts, Table views, Matrix layouts, Matrix views, Requirements, Actors, Use Cases, and all diagrams.
 - You can use the `General::Model::NamesRegExp` property to control what special characters are allowed. For detailed information on a property, see the definition displayed in the **Properties** tab of the Features window. For information about using properties in your project, see [Properties](#).
 - While you can use spaces (but not special characters) in the names for actors, it is not typical because the spaces might cause problems during code generation.

Standard prefixes

Lower and upper case prefixes are useful for model elements. The following list shows common prefixes with examples of each:

- ◆ Event names = “ev” (evStart)
- ◆ Trigger operations = “op” (opPress)
- ◆ Condition operations = “is” (isPressed)
- ◆ Interface classes = “I” (IHardware)

Using project units

In Rational Rhapsody, a *unit* is any element of a project that is saved in a separate file. You can partition your model into units down to the class level. Creating units simplifies collaboration in team environments. With this feature, you have explicit control over file names and modification rights, and you can check unit files in and out of a configuration management system.

Note

Association ends and ports cannot be saved as units.

The project and all packages are always units. The following table lists other project elements that can be units.

Element	File Extension	Unit by Default?
Actors	.cls	No
Components	.cmp	Yes
Packages	.sbs	Yes
Classes	.cls	No
Implicit objects (parts)	.cls	No
Files	.cls	No
Diagrams (except statecharts and activity diagrams)	Block definition diagrams (*.omd)	No
	Component diagrams (*.ctd)	No
	Collaboration diagrams (*.clb)	No
	Deployment diagrams (*.dpd)	No
	Internal block diagrams	No
	Object model diagrams (*.omd)	No
	Sequence diagrams (*.msc)	No
	Structure diagrams (*.std)	No
	Use case diagrams (*.ucd)	No

Unit characteristics and guidelines

The following unit characteristics and guidelines might help you use units effectively in your projects:

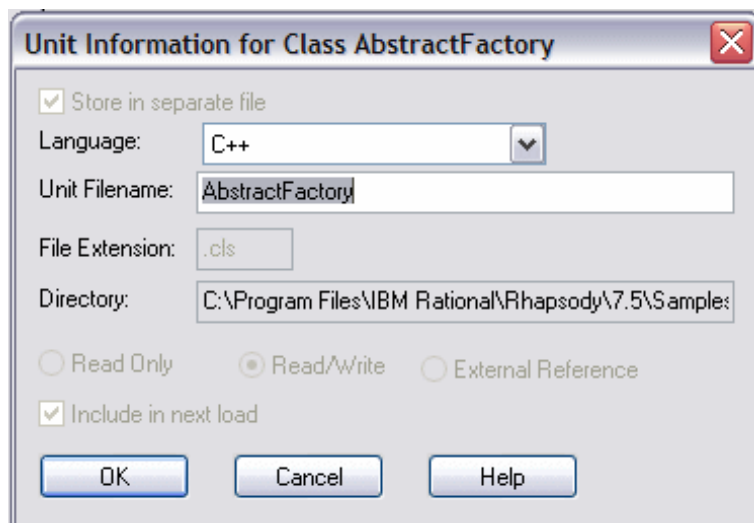
- ♦ A unit can be part of only one Rational Rhapsody model. Therefore, a unit can be modified in only one Rational Rhapsody model.
- ♦ A unit can be referenced as often as necessary.
- ♦ A unit can contain many subunits.
- ♦ Each model element is identified by a Global Unique Identifier (GUID), so each unit is unique in its model.
- ♦ Only model elements that must be shared should be changed into units.
- ♦ A unit's name should be the same as its model element name. This simplifies the association between a unit and its model element.
- ♦ Changing many model elements to units might slow Rational Rhapsody processes.
- ♦ To create diagrams automatically as units, change the `General::Model::DiagramIsSavedUnit` property to be Checked. The default is Unchecked.

Separating a project into units

If you need to share units in one project with another or you need to control some model elements in a configuration management system, you might need to divide the project into units. Keep in mind the [Unit characteristics and guidelines](#).

To change elements to units:

1. Right-click the element and then select **Create Unit**. The Unit Information window for the element opens with fields filled in, as shown in the following figure:



2. By default, the **Store in separate file** check box is selected.
3. Edit the default **Unit Filename**, if wanted. Rational Rhapsody assigns the appropriate file extension for you in the next field. The Rational Rhapsody **File Extension** is displayed and cannot be changed if the unit must have a specific extension for Rational Rhapsody.
4. You change the **Directory** for the new unit.
5. In the access privileges radio button area, you might want to change the default **Read/Write** selection for the new unit to **Read Only** or **External Reference**.
6. The **Include in next load** selection is automatically checked since you are adding a unit.
7. Click **OK**.

The unit in the browser is now marked with a small red icon. Save the unit and the icon turns black.

Modifying units

To modify a unit:

1. In the browser, right-click the unit to modify, then select **Unit > Edit Unit**. The Unit Information for Package window opens.
2. Change the settings as wanted.
3. Click **OK**.

To reduce the amount of time required to save a project, Rational Rhapsody marks all modified units and enables you to save those units without saving the entire project. To indicate that a unit has changed, the unit icon in the browser changes from black to red.

Saving individual units

To save a selected unit:

1. Select the unit in the browser.
2. Right-click the unit and select **Unit > Save Unit**.

Loading and unloading units

When you open a Rational Rhapsody project, the Open window allows you to specify whether or not Rational Rhapsody should load the subunits contained in the project. If you need to locate the unloaded units in a project, you can use either of these tools:

- ♦ [Rational Rhapsody browser menu options](#) for the Unloaded Units
- ♦ [Search and replace facility](#)

If you elected to load the project without its subunits, you can later load individual subunits: Right-click the subunit in the browser to display the menu and select **Load [unit name]**, or select **Load [unit name] with Subunits** to load the unit together with all its subunits.

To unload a unit, right-click the subunit in the browser and select **Unit > Unload [unit name]**.

Units that are not currently loaded are indicated by (U) before the unit name in the browser.

If you do want to unload a unit, but you want to prevent it from being loaded the next time you open the project:

1. In the browser, right-click the unit and select **Unit > Edit Unit**.
2. On the Unit Information, clear the **Include in next load** check box.

Loading units from last session

If you would like to load only those units that were open during your last Rational Rhapsody session, select the *Restore Last Session* radio button when you open the project with the Open window.

If you open the project by selecting its name from the MRU list (under *File*), Rational Rhapsody will also only load those units that were open when you completed your last session.

Note

This applies also to the *Include in next load* check box. If you cleared this check box, Rational Rhapsody will refrain from loading the unit only if you select the *Restore Last Session* radio button when you open the project, or open the project from the MRU list.

Saving packages in separate directories

To assist with configuration management and improve project organization, you might want to store packages in separate subdirectories within a parent folder. Rational Rhapsody has two directory schemes:

- ♦ In *flat* mode, all package files are stored in the project directory, regardless of their location in the project hierarchy.
- ♦ In *hierarchical* mode, a package is stored in a subdirectory one level below its parent. It is possible to have a hybrid project, where some packages are stored in flat mode, and others are organized in a hierarchy of folders.

To set the default so that new packages are stored in separate directories:

1. Select the <ProjectName> at the top of the browser hierarchy.
2. Right-click and select Features.
3. Locate the General::Model properties and select the DefaultDirectoryScheme property.
4. Change the value from flat to PackageAsDirectory.

Flat mode

In flat mode, Rational Rhapsody stores all package files in one directory. This is usually the project directory.

If you are changing modes from hierarchical to flat, Rational Rhapsody maintains the existing directory structure, but does not add any new subdirectories. New packages are stored within the existing structure beneath the directory of their closest parent.

To create a new model that will be in one file, after you create the project, set the following properties' check boxes to be Cleared at the project level:

- ♦ General::Model::BlockIsSavedUnit
- ♦ General::Model::ClassIsSavedUnit
- ♦ General::Model::ComponentIsSavedUnit
- ♦ General::Model::DiagramIsSavedUnit
- ♦ General::Model::FileIsSavedUnit
- ♦ General::Model::ObjectIsSavedUnit
- ♦ General::Model::PackageIsSavedUnit

Hierarchical mode

In hierarchical mode, you can save a package in a unique subdirectory one level below the directory of its parent. All units contained in the package are saved in its subdirectory, along with the package (.sbs) file. Nested packages are further divided into subdirectories.

Consider the example of a project `Home` that contains the package `Family`, which contains the package `Pets`. With each package in its own directory, the path of the `Pets.sbs` file would be:

```
../Home/Family/Pets/Pets.sbs
```

Note

When changing from flat mode to hierarchical mode, Rational Rhapsody does not automatically create folders for existing packages. Instead, it creates a folder for each *new* package within the existing directory structure.

1. Right-click the package and select **Unit > Edit Unit**. The Unit Information for Package window opens.
2. Select the **Store in separate Directory** check box (available only for packages). The name of the separate directory has the same name as the unit.
3. Click **OK**.

Rational Rhapsody creates the new directory and moves the package, along with all of its subunits, into the new folder.

Changing a hierarchical model to a flat model

To change an existing model from hierarchical mode to flat mode, write a VBA script that iterates over the entire model and, for each `IRPUnit`, calls the `setSeparateSaveUnit(true)` method. The only unit that should not activate this method is the project.

Using environment variables with reference units

If you have a reference unit in your model (added using **Add to Model As Reference**), you can edit its location using the **Directory** field of the Unit Information window, and use an environment variable as part of that location.

Note

When you add a reference to your model, Rational Rhapsody adds the packages as top-level packages by default. However, you can move the reference packages so they become nested packages.

For example, you can use a relative path (`.. \`) or the environment variable `$ENV_VAR`. If you set the `General::Model::EnvironmentVariables` property to include the path of this environment variable, Rational Rhapsody parses and executes that environment variable when it opens the project, and then searches for the reference unit in the specified location.

Note

If you use relative paths, note that the path is relative to the `_rpy` folder, not where the `.rpy` file is located.

Preventing unresolved references

When you delete an element that has references in read-only files, those references cannot be updated accordingly on the disk because the files are read-only files. These references become unresolved when the model is reloaded. In order to resolve this situation, read-only files that contain references to elements that you are deleting should be made writable before you perform the delete operation.

Rational Rhapsody automatically opens the References in Read-Only Files Encountered window when it detects that an attempt to delete an element might cause unresolved references. When you delete an element (for example, through the Rational Rhapsody browser or a graphic editor) that has references in read-only files, the References in Read-Only Files Encountered window opens with a list of read-only files that contains references to the deleted element.

The following options are available to you on this window:

- ◆ **Check out the selected and continue** - Use only if you have a configuration management tool set.
 - Note:** This option might fail if the unit you are trying to check out is not already checked in.
 - Note:** This option is unavailable in the Rational Rhapsody Platform Integration.
- ◆ **Make the selected files read/write and continue**
- ◆ **Ignore the references in read-only files and continue.** Using this option means that unresolved references might be created when the model is reloaded.

Using workspaces

Workspaces enable you to work with selected units of a project without having to open the entire model. This feature supports component-based development and collaboration among teams. It also reduces the time required for routine operations, such as saving and code generation, by enabling you to load only the units currently under development.

In addition, workspaces save viewing preferences, including window size, position, status of feature windows, and the scaling or zoom factor of open diagrams.

Rational Rhapsody automatically saves workspace information in a separate file named `<Project>.rpw`. Rational Rhapsody saves the `.rpw` file whenever a project is closed, regardless of whether you save the project itself.

Creating a custom Rational Rhapsody workspace

1. Choose **File > Open**. The Open window displays.
2. In the **Look in** field, browse to the location of the project, then select the `.rpy` file.

Alternatively, type the name of the project file in the **File name** field.
3. Select the **Without Subunits** check box. This prevents Rational Rhapsody from loading any project units. All project units will be loaded as stubs.
4. Click **Open**. The project file opens with no units loaded. This empty project acts as a starting point for you to create your workspace.
5. Add units to your workspace to customize it for your needs.

Adding units to a workspace

1. Select the unit in the browser.
2. Right-click the unit, then select **Load Unit**.
3. Select the unit in the browser.
4. Right-click the unit, then select **Load Unit with Subunits**.

Unloaded units

Units of your project that have not been loaded into your workspace are marked with the letter “U.” This designation means that the unit is a *stub* unit, and was either excluded from the project intentionally, or was not found when Rational Rhapsody attempted to load the unit.

In diagrams, a “U” located at the destination end of a relation, dependency, or generalization means that the target is an unresolved element. In this case, the originator of the relation, dependency, or generalization is loaded, but the unresolved element has been either intentionally or accidentally excluded from the project.

You can use the [Advanced search and replace features](#) to locate any unloaded units and load them.

Opening a project with workspace information

To open a workspace:

1. Select **File > Open**. The Open window displays.
2. In the **Look in** field, browse to the location of the project, then select the .rpy file. Alternatively, type the name of the project file in the **File name** field.
3. Select the **Restore Last Session** check box. The project opens with the workspace information that was saved during your last Rational Rhapsody session.

To open a project without loading workspace information, see [Opening an existing Rational Rhapsody project](#).

Controlling workspace window preferences

Workspaces save information about your window preferences. These preferences include window size, position, status of feature windows, and the scaling or zoom factor of open diagrams.

To prevent Rational Rhapsody from saving graphic editor settings, set the `OpenDiagramWithLastPlacement` property under `General::Workspace` `BlockIsSavedUnit` check box to `Cleared`. With this setting, Rational Rhapsody does not save the position of graphic editor windows. Instead, it uses the default window settings the next time you open a graphic editor.

You can prevent Rational Rhapsody from saving any window preferences by setting the `OpenWindowsWhenLoadingProject` property under `General::Workspace` check box to `Cleared`.

Project files and directories

The project directory is the top-level directory for a project. It is the folder entered in the New Project window when you create a new project. Rational Rhapsody creates a number of project files with matching directories. Each project file/directory pair shares the same name.

The following table lists project files created by Rational Rhapsody.

File Name	Description
<Project>.rpy	The project file or model repository. Requires the repository files in the <Project>.rpy directory to be a complete model.
<Project>_rpy	Directory containing unit files for the project, including: • Components (*.cmp) • Packages (*.sbs) • Classes (*.cls) • Use case diagrams (*.ucd) • Sequence diagrams (*.msc) • Object model diagrams (*.omd) • Component diagrams (*.ctd) • Collaboration diagrams (*.clb) • Deployment diagrams (*.dpd) • Structure diagrams (*.std) • Table of files contained in the project (filesTable.dat)
<Project>_auto.rpy	A backup file of the model, created during autosave. This file is written only if the <Project>.rpy has been modified since it was last saved.
<Project>_auto_rpy	Directory containing a backup of project files modified since the last save. All autosave files are stored in flat mode. For more information, see Saving packages in separate directories .
<Project>_bak1.rpy	A backup of the model, created when the project is first saved. Requires the repository files in the <Project>_bak1_rpy directory to be a complete model.
<Project>_bak1_rpy	Directory containing a backup of the project files for the <Project>_bak1.rpy repository. This folder can contain the same types of files as the <Project>_rpy folder.
<Project>_bak2.rpy	A backup of the model, created when the project is saved a second time. Contains the most recent backup of the project. Requires the repository files in the <Project>_bak2_rpy directory to be a complete model.
<Project>_bak2_rpy	Directory containing a backup of the project files for the <Project>_bak2.rpy repository. This folder can contain the same types of files as the <Project>_rpy folder.

File Name	Description
<Project>.rpw	Workspace settings file. Preserves the workspace settings for the project. For more information, see Using workspaces .
<Project>.ehl	Events history list. Stores events and breakpoints during animation.
<Project>.vba	VBA project file. Contains VBA macros, modules, user forms, and so on.
ReverseEngineering.log	A log of reverse engineering activity containing messages reported in the output window during reverse engineering.
<Project>_ATG	Directory that holds any tests created using the Rational Rhapsody Automatic Test Generation add-on (if you added the product).
<Project>_RTC	Directory that holds any tests created using the Rational Rhapsody TestConductor™ add-on (if you added the product).
load.log	A log of when various repository files were loaded into Rational Rhapsody, including any errors that might have occurred during the loading and resolution phases.
store.log	A log recording when the project was saved.
<Component>	Directory for each component in the project. Organizes files generated for each configuration in the component. Each configuration is placed in a subdirectory that contains the source and binary files for a build.

Parallel project development

Many companies use Rational Rhapsody to create large models developed by multiple users, who are often working in parallel in distributed teams. These teams might use a source control tool or configuration management (CM) software, such as Rational ClearCase, to archive project units, but not all files might be loaded into CM during development.

Engineers in the team need to see the differences between an archived version of a unit and another version of the same unit or a similar unit that might need to be merged. To accomplish these tasks, they need to see the graphical differences between the two versions, as well as the differences in the code. However, source control software does not support graphical comparisons.

The Rational Rhapsody DiffMerge tool supports team collaboration by showing how a design has changed between unit revisions and then merging units as needed. It performs a full comparison including graphical elements, text, and code differences.

Unit types

A Rational Rhapsody unit is any project or portion of a project that can be saved as a separate file. These are some examples of Rational Rhapsody units with the file extensions for the unit types:

- ◆ Class (.cls)
- ◆ Package (.sbs)
- ◆ Component (.cmp)
- ◆ Project (.rpy)
- ◆ Any Rational Rhapsody diagram

DiffMerge tool functions

The DiffMerge tool can be operated inside and/or outside your CM software to access the units in an archive. It can be launched from inside or outside Rational Rhapsody. It can compare two units or two units with a base (original) unit.

The units being compared only need to be stored as separate files in directories and accessible from the PC running the DiffMerge tool. In addition to the comparison and merge functions, this tool provides these capabilities:

- ◆ Graphical comparison of any type of Rational Rhapsody diagram
- ◆ Consecutive walk-through of all of the differences in the units
- ◆ Generate a Difference Report for a selected element including graphical elements
- ◆ Print diagrams, a Difference Report, Merge Activity Log, and a Merge Report

Note

Many of the DiffMerge tool's operations can be run from a command-line interface to automate some of the tasks associated with software development (for example, to schedule nightly builds).

Project migration and multi-language projects

When you run Rational Rhapsody, you select a specific language version of Rational Rhapsody. This determines the language that is associated with your Rational Rhapsody projects.

You can, however, open Rational Rhapsody projects that were created with other language versions of Rational Rhapsody.

In addition, Rational Rhapsody allows a single model to contain units that are associated with different languages. A model can include units associated with C, C++, or Java. Code can then be generated in the appropriate language for each unit.

Note

While project migration is a built-in feature of Rational Rhapsody, you can only have multi-language projects if you possess the special license required for this feature.

Opening models from a different language version

Rational Rhapsody allows you to open models created in a different language version of Rational Rhapsody. This is also referred to as *migration* of projects.

When you try to open a project that was created in a different language version of Rational Rhapsody, you are notified that the project will be converted to the language of the current version, and you are asked whether you would like to continue with the conversion of the project.

Note

When you migrate a project, you do not lose any language-specific features of model elements that are not supported in the language version of Rational Rhapsody that you are running. These language-specific characteristics will not be displayed, for example, in the Features window, and any code generation will be in the language of the current version not the version with which the model was originally created. However, Rational Rhapsody maintains this information. If, at a later stage, you reopen the model in the original language, you will once again see these language-specific characteristics.

When a project is migrated, bodies of operations and any other code entered manually in Rational Rhapsody are not converted to the target language. If you already have such code in your model before the migration, make sure to convert the code in order to avoid compilation errors.

If you use *Add by reference* to add a unit whose language differs from that of the version of Rational Rhapsody you are running, a non-persistent conversion is performed (since these elements are read-only). This non-persistent conversion will be performed each time you open the model.

Note

If you have a license for multi-language projects, no conversion is performed when you open a model from another language version of Rational Rhapsody. If you would like to convert an entire project, just change the unit language at the project level. For details, see [Determining language of a unit in multi-language projects](#).

Multi-language projects

Rational Rhapsody uses units to permits projects to contain components from different development languages. Each unit is associated with a specific language.

Determining language of a unit in multi-language projects

When you create a new unit, the Unit Information window provides a list that allows you to select a specific language for the unit. The default language for a new unit is the language of its owner unit.

To change the language of an existing unit:

1. Right-click the unit in the browser, and then select **Unit > Edit Unit**.
2. When the Unit Information window is displayed, select the language from the list.
3. Click **OK**.
4. When you are asked to confirm the change, click **Yes**.

If you are changing the language of a unit that contains subunits, Rational Rhapsody will ask you if you also would like to change the language of all of the contained subunits.

Note

As is the case for project migration, if you change the language of a unit, you do not permanently lose any language-specific features of the unit. These language-specific characteristics will not be displayed, and any code generation will be in the new language. However, Rational Rhapsody maintains this information. If, at a later stage, you switch the unit back to the original language, you will once again see these language-specific characteristics.

When you move units of one language to a package of another language, Rational Rhapsody will inform you that they are different languages and will ask you to confirm the move.

If you try to add a unit that is associated with another language, Rational Rhapsody will ask you to confirm the addition of the unit. You will also be prompted for confirmation if you add “by reference” a unit that is associated with another language.

Code generation

In each of the language versions of Rational Rhapsody, you can generate code for units in each of the three languages - C, C++, Java.

For code generation to work properly, you have to adhere to the following rules:

- ♦ To generate code for units in a certain language, the appropriate language must be specified at the component level.
- ♦ Elements included in the scope of a component must be of the same language as the component. (If other language elements are included in the scope, a warning will be issued during code generation.)

Note

If you select **All Elements** as the scope, Rational Rhapsody will automatically include only those units whose language matches that of the component. If you choose **Selected Elements**, Rational Rhapsody will only display those units whose language matches that of the component. However, if you selected specific elements, and then changed the language of the component, Rational Rhapsody will not deselect these non-matching units. When you attempt to generate code with such a component, you will receive error messages. The same principle applies to the *Initial Instances* specified for the configuration.

Language-specific differences in Rational Rhapsody

The Features window differs from language to language. For each unit in your model, the appropriate Features window is displayed.

Similarly, the properties displayed reflect the language of the selected unit.

Non-unit elements

If you try moving an element not saved as a unit to a package with a different language, the language of the element will be changed to that of the receiving package after you confirm that you want to move the element.

Reverse engineering

The reverse engineering mechanism always uses the language of the active component. When you use the Reverse Engineering window to add files to reverse engineer, the default file filter used will reflect the language of the active component, for example, *.java if the active component is associated with Java.

Miscellaneous issues

- ◆ Rational Rhapsody API: The interface `IRPUnit` allows recursive changing of unit language.
- ◆ ReporterPLUS can query the language of an element.
- ◆ The Rational Rhapsody internal reporter shows the language of each saved unit.
- ◆ XMI: Language of each unit is exported and imported.
- ◆ Graphic Editor: Changes to language of a unit do not affect the depiction of the unit in the graphic editor. For example, if you change the language of a template class to C, it will still look like a template class in the graphic editor.
- ◆ DiffMerge checks for language differences.
- ◆ PredefinedTypes package: These packages are language-dependent. When you create a unit whose language differs from that of the Rational Rhapsody version being used, the relevant package of predefined types for that language will be loaded.

Domain-specific projects and the NetCentric profile

The Rational Rhapsody NetCentric Profile builds domain-specific projects for a Service Oriented Architecture (SOA) or to support Network Centric Warfare (DoDAF) Applications. In SOA projects, developers begin by writing the interface specifications using Web Service Description Language (WSDL), a complex XML schema.

Unfortunately, WSDL requires that the data types be fully defined and contain legal XML types. WSDL also requires the developers to define the interface calls in terms of where the services are to be deployed (bindings and namespaces). These details are not always known during the initial design phase. The Rational Rhapsody NetCentric profile helps the engineer bridge this gap from the design concepts to the WSDL file production.

SOA or NetCentric application model roles

SOA and NetCentric application models have two roles:

- ◆ Service provider or the service itself
- ◆ Service consumer is the user of the service

Service consumers

A service consumer can be any actor, application, or any other receiver of the deployed service. For example, a weather station might receive the weather information provided by a SOA application.

Using the platform independent WSDL files for application output, service users gain access to the output more easily than from output in a proprietary format.

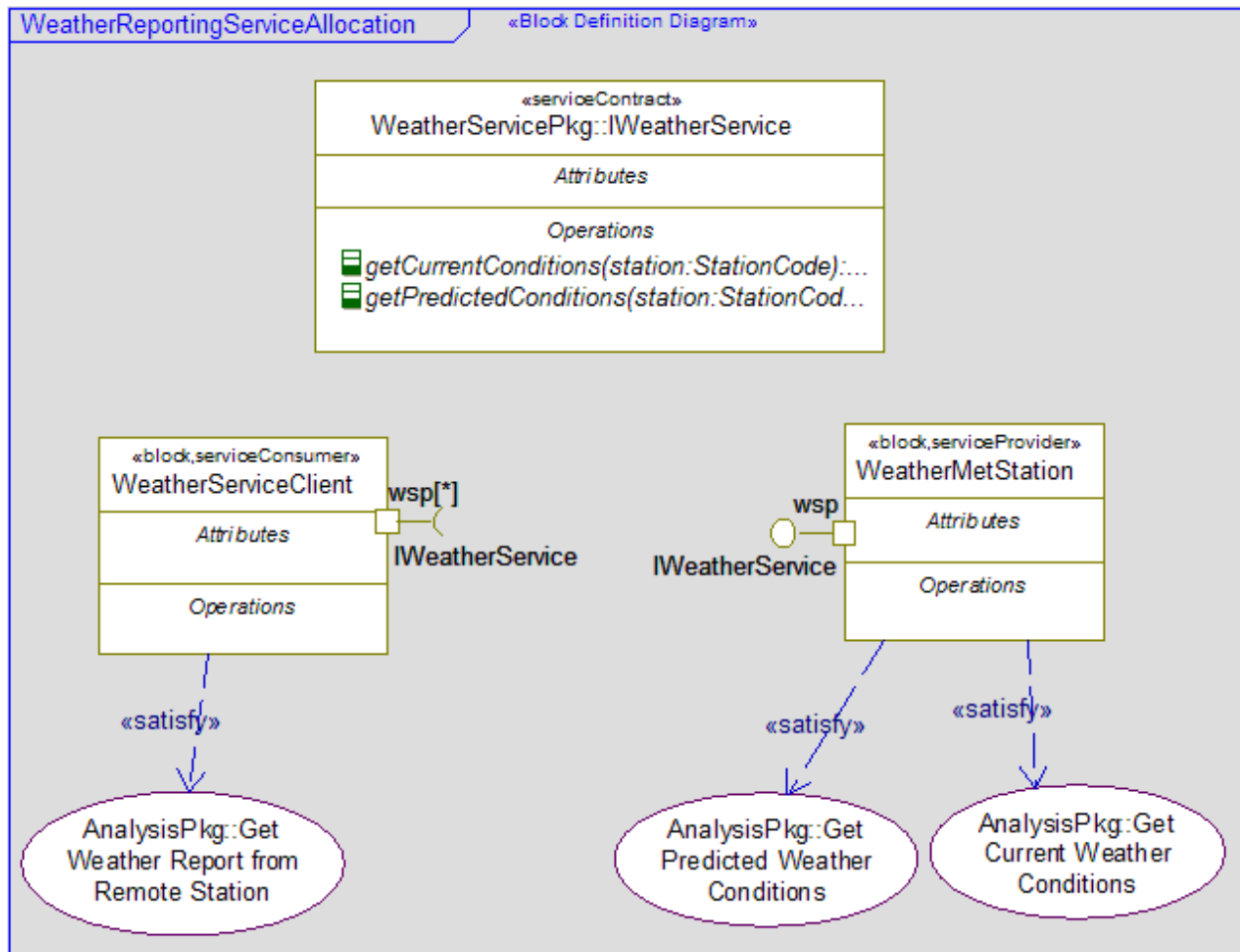
Service provider

The service provider is the code that implements the service. The service provider includes the interface specification (also called the service contract) and the related WSDL file. The service provider also includes everything necessary to produce the WSDL file:

- ◆ Service contract (interface class)
- ◆ Realization of this contract (classes that implement the service)
- ◆ Data types
- ◆ WSDL file.

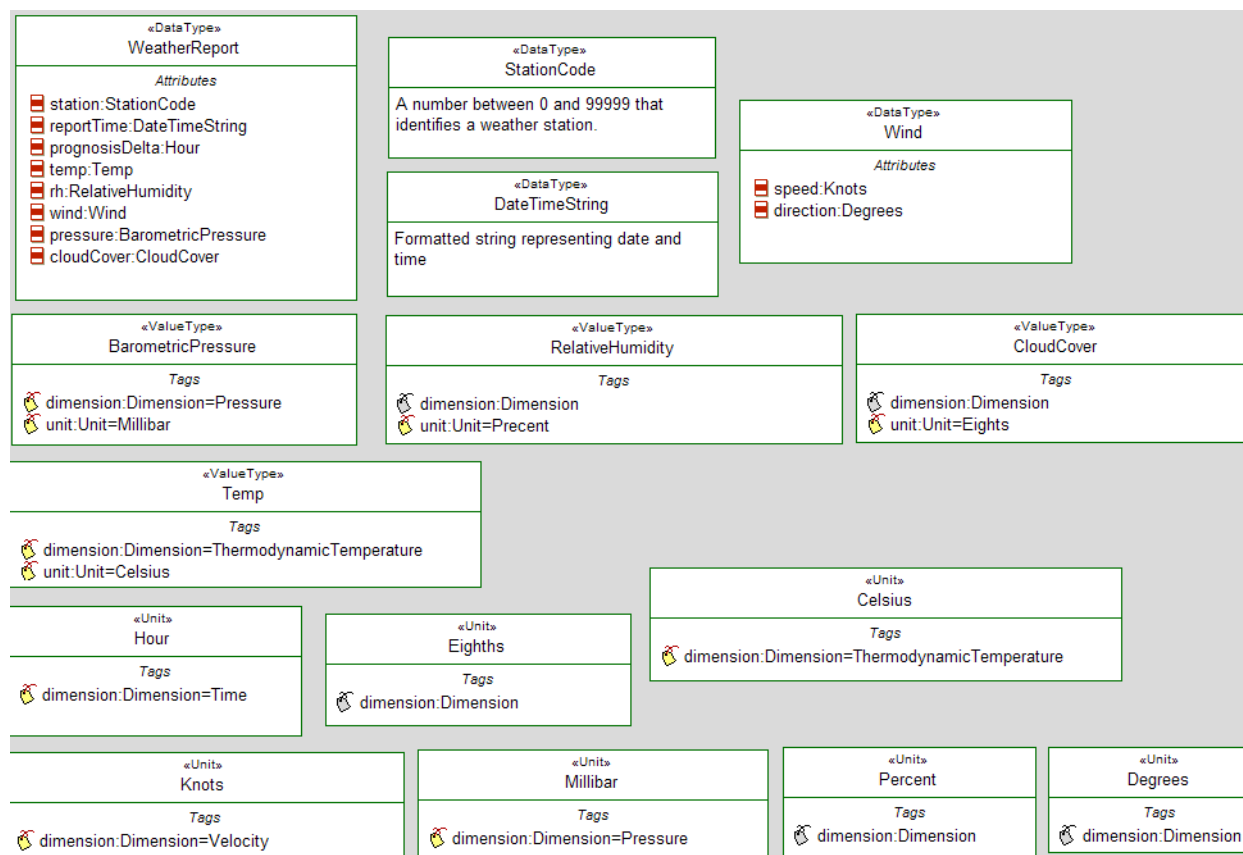
Rational Rhapsody uses the stereotype <<servicePackage>> to indicate a UML/SysML package that contains the model elements necessary for creating the WSDL file that includes the package with the service provider and the service contract. The stereotypes <<serviceProvider>> and <<serviceContract>>, respectively, indicate these classes. To define the data types, the WSDL files also include XML schemas and XSDs.

This Block Definition diagram example shows a typical collaboration in which the stereotypes indicate the service contract, service consumer, and service provider.



Collaborations in a top level or System of Systems model allow the systems engineers to confirm the data types, interfaces, and basic block behavior before generating the WSDL file. Executing the model ensures that both sides of the interface interpret messages the same way. This avoids consistency errors that, otherwise, might not be discovered until late in the integration phase.

The systems engineers and designers might model data types using the SysML units and value types, as shown the Block Definition diagram example (from Rational Rhapsody System samples “NetCentricWeatherService” project):



The Generate WSDL Specification tool uses these data types to create the schema information within the WSDL file.

Creating a NetCentric project

The NetCentric profile is available as a standard profile selection in the following Rational Rhapsody editions:

- ◆ Developer edition for C, C++, Java, and Ada projects
- ◆ Architect for Systems Engineers edition
- ◆ Architect for Software edition

To create a NetCentric project from one of these editions:

1. With your Rational Rhapsody edition running, create the new project by either selecting **File > New**, or clicking the **New project** button on the main toolbar.
2. Replace the default project name (Project) with <your project name> in the **Project name** field. Enter a new directory name in the **In folder** field or Browse to find an existing directory.
3. Select the NetCentric **Project Type**.
4. You might want to select a different option for the **Project Settings**.
5. Click **OK**. If the directory does not exist, Rational Rhapsody asks if you want to create it. Click **Yes** to create the new project directory.

To add the NetCentric profile manually to an existing project:

1. With the Rational Rhapsody project open, choose **File > Add Profile to Model**.
2. In the Add to Profile Model window, select the Package (*.sbs) file for the NetCentric profile. (If the NetCentric.sbs file is not displayed in the Profiles directory, the project you have created is not compatible with the NetCentric profile. The SysML project profile is often used with the NetCentric profile.)
3. Click **Open** to add the profile to your project.

Creating a service contract to export as WSDL

In order to create a service contract to export as a WSDL file, you must assign the set of NetCentric stereotypes to the service package in your Rational Rhapsody project.

To create a service contract to export as WSDL:

1. In a Rational Rhapsody NetCentric project, define an interface block or class. This interface contains the methods that define the service that is called by the service consumer.
2. In the browser, right-click the interface block or class and select **Features**.

3. In the **Stereotype** field, select <<New>> and enter serviceContract. Click **OK**.
4. Create the block or class that realizes the serviceContract. Apply the stereotype <<serviceProvider>> to it.
5. Add a standard port to the <<serviceProvider>>. Set the port to provide the service contract interface. Stereotype this port <<servicePort>>.
6. Define all of the data types needed by the interface.
7. Set the stereotype on the package to <<servicePackage>>.

Exporting a WSDL specification file

To export a WSDL specification file from your Rational Rhapsody project:

1. Open the NetCentric project containing the WSDL specification. It is stored as a <<wsdlDefinitionDocument>> stereotyped package.
2. Select that package in the project browser.
3. Select **Tools > Generate WSDL Specification**.
4. Enter the name and location for the output of WSDL specification file. Specify the target output file with the .wsdl extension.
5. Click **OK**.

When the export is complete, you can open the WSDL file in any editor.

Importing a WSDL specification

To import a WSDL specification file into a Rational Rhapsody project:

1. With the Rational Rhapsody project open, choose **Tools > Import WSDL Specification**.
2. Select the previously generated WSDL file and click **Open**.

Schedulability, Performance, and Time (SPT) profile

The SPT profile (also known as the UML Real-Time profile) is a standard UML profile adopted by the OMG. The profile has the following uses:

- ◆ Enable the construction of models that could be used to make quantitative predictions regarding these characteristics.
- ◆ Facilitate communication of design intent between developers in a standard way.
- ◆ Enable interoperability between various analysis and design tools.

The SPT.rpy model provided in <Rational Rhapsody installation path>\Share\Profiles\SPT is the Rational Rhapsody implementation of the standard profile that you can use in any Rational Rhapsody model.

Note

This functionality can only be used with Rational Rhapsody in C++.

Manually adding the SPT profile to your model

If you have not created your project using the SPT profile as the Type, you can add the SPT profile to your model:

1. Open your Rational Rhapsody model.
2. Choose **File > Add Profile to Model**.
3. On the Add Profile to Model window, select the **SPT.sbs** unit.
4. Click **Open**.
5. On the Add To Model From Another Report window, select the **As reference** radio button.
6. Click **OK**.

Rational Rhapsody adds the SPT profile as a reference profile to your model. As a result, the stereotypes and tagged values of the profile become available.

Using the stereotypes and tagged values

1. Select the model element (for example, a class named MyClock).
2. Assign the stereotype you want to the model element (for example, RTClock).
3. Set tag values through the **Tags** tab of the Features window for the element.

Changing the profile

To change the profile, edit the SPT model.

Note

Deleting a stereotype or changing its metaclass might result in unresolved references in the models using it (that is, the models that are referencing the SPT model and have elements with this stereotype).

Rational Rhapsody with IDEs

You can connect Rational Rhapsody to another development tool or IDE (integrated development environments) to use features of both. You can use Rational Rhapsody with these IDEs:

- ◆ Eclipse
- ◆ Visual Studio
- ◆ Tornado

Rational Rhapsody provides some standard menu commands to setup and work within both environments.

IDE options

Use the **Code > IDE Options** on the Rational Rhapsody menu to set the ports to receive and send messages for the IDE and Rational Rhapsody and synchronize the IDE with Rational Rhapsody.

Locating Rational Rhapsody elements in an IDE

If you want to locate an element from a Rational Rhapsody project in another development tool (IDE), select the element in Rational Rhapsody and select **Edit > Locate in IDE** or press **Ctrl+Alt+K**. The program, such as Eclipse or Visual Studio, opens and displays the element in its project environment.

Opening the IDE

To work in the IDE, open Rational Rhapsody and choose **Code > Open IDE**. Your configured IDE, such as Eclipse or Visual Studio, launches.

Creating an IDE project

After you have created a configuration for the IDE in your Rational Rhapsody project, you can create a project in the IDE. Right-click the IDE configuration and select **Create IDE Project**. Your configured IDE launches.

Using the Rational Rhapsody Workflow Integration with Eclipse

The Rational Rhapsody plug-in for Eclipse has two implementations:

- ♦ **Rational Rhapsody Workflow Integration** allows the software developer to work in Rational Rhapsody and use some Eclipse features through Rational Rhapsody menu commands. This integration can be used for C and C++ development in either Windows or Linux environments. Both Eclipse and Rational Rhapsody must be open when the developer is using this integration.
- ♦ **Rational Rhapsody Platform Integration** permits developers to work on a Rational Rhapsody project completely within Eclipse. Rational Rhapsody does not need to be open for this implementation. This integration can be used for C, C++, or Java development in a Windows environment only.

The Rational Rhapsody *Workflow integration with Eclipse* allows software developers to work on Rational Rhapsody C or C++ projects in Eclipse version 3.3 or WindRiver's Eclipse Workbench 2.6 to perform these tasks:

- ♦ Create a new IDE (integrated development environment) configuration in Rational Rhapsody in order to perform the following tasks:
 - work in Eclipse on a new Rational Rhapsody project
 - attach an existing Rational Rhapsody project to an existing Eclipse project
- ♦ Import legacy Eclipse models into Rational Rhapsody UML models
- ♦ Make changes in Eclipse and rebuild the project automatically in both Rational Rhapsody and Eclipse
- ♦ Use the debugging facilities in Rational Rhapsody and Eclipse in a synchronized manner
- ♦ Use Rational Rhapsody reverse engineering with an active Eclipse configuration
- ♦ Disconnect an Eclipse project from the associated Rational Rhapsody configuration

Converting a Rational Rhapsody configuration to Eclipse

If you created a configuration in Rational Rhapsody, but now you want to convert it into an Eclipse configuration:

1. Open Rational Rhapsody and in the Rational Rhapsody browser, right-click the configuration that you want to convert to be an Eclipse configuration and then select the **Change to > Eclipse Configuration**.
2. From this point, the process is the same as creating a new Eclipse configuration.
 - ♦ Outline
 - ♦ Navigator

Importing Eclipse projects into Rational Rhapsody

Rational Rhapsody makes it easy for you to import your Eclipse projects quickly. When you import an Eclipse project, all elements contained in the project are reverse engineered and added as elements of a Rational Rhapsody model. The imported elements are added in a new package which uses the name of the original Eclipse project. To import an Eclipse project:

1. Create a new Rational Rhapsody project, or open an existing Rational Rhapsody project.
2. Select **Tools > Import from Eclipse**.
3. If Eclipse is not currently open, Rational Rhapsody will launch it (after first asking you to confirm that you want Eclipse opened), and the Export Rhapsody Model window will be displayed in Eclipse. (If the window does not appear, check the port settings used; select **Code > IDE Options** from the main Rational Rhapsody menu.)
4. Select the projects that you would like to export to Rational Rhapsody. (If an Eclipse project has already been imported into Rational Rhapsody, it will not appear in the list of available Eclipse projects.)
5. If you want the elements in the project to be added to the Rational Rhapsody model as external elements, select **Export as External**.
6. If you want to fine-tune the reverse engineering options that will be used for importing your Eclipse project, select **Open Reverse Engineering options dialog in Rhapsody before export**.
7. Click **Finish**. All of the elements in the Eclipse project will be imported into the open Rational Rhapsody project.
8. When the import process has been completed, look in the Rational Rhapsody browser for a package that has the same name as the Eclipse project you imported. You will also

notice that a new component has been added to your Rational Rhapsody model, containing an Eclipse configuration named after your Eclipse project.

Creating a new Eclipse configuration

If the developer or designer prefers to use the Eclipse IDE (integrated development environment) to work on a project previously created in Rational Rhapsody:

1. Display the existing Rational Rhapsody C or C++ project in Rational Rhapsody.
2. In the Rational Rhapsody browser, right-click the Component in the Rational Rhapsody project for which you want to create an Eclipse configuration and then select **Add New > Eclipse Configuration**. The system displays a window, asking whether or not the user wants to launch the IDE if it is not running.

Note: If IDE is already running, be certain that the ports for Eclipse and Rational Rhapsody match by selecting the **Code > IDE options** from the Rational Rhapsody menu and making any changes required. If the ports do not match, a new IDE might open even if the user meant to switch to the running IDE!

3. For the WindRiver version of Eclipse, the Workspace Launcher displays so that you can select a directory for your Eclipse project workspace. Click **OK** to save the selected directory.
4. Then the Rhapsody Project Wizard displays. It lists the name of the Rational Rhapsody project and the component you selected for the new Eclipse configuration. Select whether you want to create a **New Project** or an **Existing Project**. Click **Finish**.
5. The New Project window displays Wizard types. However, currently only the vxWorks Downloadable Kernel Module Project is supported. Select it and click **Next**.
6. Type a **Project name** and select a workspace in this window. Complete setting up the Workbench project, as described in the Workbench documentation.
7. The Application Development interface displays the Eclipse configuration of the selected Rational Rhapsody component.
8. Back in Rational Rhapsody, the browser now contains the new Eclipse configuration.

Troubleshooting your Eclipse installation with Rational Rhapsody

If after installing Eclipse, working to set up a project as described previously, and you have not been successful, do the following troubleshooting to check your installation:

1. Check the path to Eclipse in the rhapsody.ini file to be certain that it is the actual path on your system. The following example shows a typical path:

```
[IDE]
EclipsePath=C:\eclipse\eclipse.exe
```

Note: If it does not match your path to Eclipse, make the necessary changes.

2. Check to be certain that the c:\cygwin\bin is in your environment PATH variable.
3. Start Rational Rhapsody and create or open a project.
4. Right-click the component and choose **Add New > Eclipse Configuration**.
5. You are prompted for a workspace. Then the Rhapsody Project Wizard asks you to either specify a project or create a new one. Click **Next**.
6. Specify the new project name and click **Next**.
7. Click **Finish**.
8. If the program asks if you want to associate with a C++ Perspective, click **Yes**. Now the Eclipse CDT IDE is open and linked to your Rational Rhapsody project.
9. Switch to Rational Rhapsody and make sure your Eclipse configuration is active. Generate code.
10. Right-click a model element in the Eclipse version of the code and select **Locate in Rhapsody**. This change should trigger a roundtrip in the Rational Rhapsody version of the code if the change is not one of the [Workflow integration with Eclipse limitations](#).

Switching between Eclipse and Wind River Workbench

When Rational Rhapsody launches an Eclipse-based IDE, whether the generic Eclipse or Wind River Workbench, it checks the `rhapsody.ini` file to determine the path of the IDE.

Then entry that stores this information is called `EclipsePath` and it is located in the section `[IDE]`. During installation, you will be asked to provide the path for Eclipse, and this information is copied to the `rhapsody.ini` file, for example:

```
EclipsePath=l:\windriver\workbench-2.4\wrwb\2.4\x86-win32\bin\wrwb.exe
```

If you want to switch between the generic Eclipse and Wind River Workbench, change the value of this entry in the `rhapsody.ini` file.

Rational Rhapsody tags for the Eclipse configuration

After creating an Eclipse configuration, the Rational Rhapsody model elements are coupled with an Eclipse-based project. The definition of the Eclipse project is stored in Rational Rhapsody in these *Tags* (displayed in the Rational Rhapsody browser):

- ♦ **IDEName** is the name of the type of integrated development environment (for example, Eclipse, Workbench).
- ♦ **IDEProject** is the project name entered while creating the Eclipse configuration.
- ♦ **IDEWorkspace** is the workspace directory for the Eclipse-based project.

The values of the tags are set automatically after the IDE project is created or mapped to the configuration. You do not need to modify the values of these tags, unless you change the IDE workspace location in the file system. If you make that change, then you must update the `IDEWorkspace` tag manually. For instructions to make this change if necessary, see [Configuring Rational Rhapsody for Eclipse](#).

Configuring Rational Rhapsody for Eclipse

To examine the features of an Eclipse configuration in Rational Rhapsody:

1. Right-click the Eclipse configuration in the Rational Rhapsody browser.
2. Select **Features** from the menu to display this window. This version of the **Features** window contains the **IDE**, **Tags**, **Properties**, and **Settings** tabs for use with Workbench projects.
3. The **IDE** tab provides two important features:
 - ♦ **Open in IDE** launches Eclipse with the corresponding project or simply bring the IDE forward.

- ♦ **Build configuration in IDE** sets the build to be performed via the IDE (by sending a request to the IDE)
- 4. However, if you want to perform the build in Rational Rhapsody, use the **Settings** tab to make the build selections.
- 5. Generally, you do not need to change the values for the **Tags**, unless you change the *IDEWorkspace directory location*. In that case, you must make the change to the path in the window and click **OK**.

Eclipse workbench properties

To define the configuration and environment for the IDE project, the following Rational Rhapsody properties are available in the [Configuring Rational Rhapsody for Eclipse](#):

Subject	Metaclass	Property Name	Default Value	Description
Eclipse	Configuration	InvokeExecutable	\$executable	Points to the executable. Keywords:\$executable - the IDE executable as read from Rhapsody.ini
Eclipse	Configuration	InvokeParameters	-data \$workspace -vmargs - DRhpClientPort=\$ RhpClientPort - DRhpServerPort=\$ RhpServerPort	Parameters for the command-line. Keywords: \$workspace as specified in the Rational Rhapsody tags for the Eclipse configuration . \$RhpClientPort: the port number that Rational Rhapsody uses to be a client to Eclipse, as specified using Rational Rhapsody menu Code > IDE options . \$RhpServerPort: the port number that Rational Rhapsody uses to be a server to Eclipse
Eclipse	DefaultEnvironments	Eclipse	Cygwin	The default environment in the settings tab for generic Eclipse (CDT) projects
Eclipse	DefaultEnvironments	Workbench	WorkbenchManaged	The default environment in the settings tab for generic Eclipse (CDT) projects

Editing Rational Rhapsody code using Eclipse

To edit code from Rational Rhapsody using Eclipse:

1. Open the Rational Rhapsody project that contains the Eclipse configuration and make it the active configuration.
2. Launch Workbench.
3. In Rational Rhapsody, right-click a class and select **Edit code in Eclipse** from the menu.
4. The implementation of that class is then displayed in Eclipse, and Eclipse automatically generates a file in DMCA mode in Rational Rhapsody.
5. Edit the code as needed. The updates are recorded in the implementation of the class.

Locating implementation code in Eclipse

If you want to examine the implementation code for model elements or errors using Eclipse:

1. Open the Rational Rhapsody project that contains the Eclipse configuration and make it the active configuration.
2. Launch Workbench.
3. In Rational Rhapsody, right-click a model element, such as an attribute for a class in the browser, and select **Locate in Eclipse** from the menu.
4. The implementation code displays in Eclipse.

Opening an existing Eclipse configuration

After creating Eclipse configurations for the Rational Rhapsody components that you want to work on in Eclipse, to open Eclipse:

1. Start Rational Rhapsody.
2. Choose **Code > IDE Open**.
3. Eclipse launches. If the active configuration in the Rational Rhapsody browser is an Eclipse configuration, that configuration is automatically displayed in Eclipse. If the active configuration is not an Eclipse configuration, the system displays a window asking the *Eclipse workspace directory*. Then it displays the Eclipse configuration in the workbench interface.

Note: Only those elements that can be edited in Rational Rhapsody can be opened for editing in Eclipse.

Disassociating an Eclipse project from Rational Rhapsody

To disassociate an Eclipse project from Rational Rhapsody:

1. In the list of C/C++ or Java projects in Eclipse, right-click the project and then select **Rhapsody > Disconnect from Rhapsody**.
2. Click **OK**.

This removes the Rational Rhapsody characteristics from the project.

When you return to Rational Rhapsody, you are asked whether you want to delete the corresponding Eclipse configuration from your model.

Note

In Rational Rhapsody, if you delete the Eclipse configuration from the model, the Eclipse project will automatically be disconnected from the model.

Workflow integration with Eclipse limitations

When using the Rational Rhapsody workflow integration with eclipse, operations originating in Rational Rhapsody and continuing into Eclipse cannot be undone. In addition, the \$executable is mapped to a single IDE. To work with several IDEs, the user must set the properties.

Visual Studio IDE with Rational Rhapsody

You can use Microsoft Visual Studio 2008 (standard or professional edition) with Rational Rhapsody for C and C++ projects.

Changing an existing Rational Rhapsody configuration to Visual Studio

To convert your Rational Rhapsody configuration into a Visual Studio configuration:

1. Open a Rational Rhapsody project.
2. Select the active Rational Rhapsody configuration in the browser.
3. Select **Change to > Visual Studio Configuration**.

Adding a new Visual Studio configuration

To add a Visual Studio configuration to your Rational Rhapsody project:

1. Open a Rational Rhapsody project.
2. Select the Rational Rhapsody configurations in the browser.
3. Select **Add New > Visual Studio Configuration**.

Creating a new Visual Studio project

After creating a Visual Studio configuration in Rational Rhapsody, you can perform the following tasks:

- ♦ work in Visual Studio on a new Rational Rhapsody project
- ♦ attach an existing Rational Rhapsody project to an existing Visual Studio project

To create a new Visual Studio project, right-click the Visual Studio configuration and select **Create IDE Project**.

Co-debugging with Tornado

Rational Rhapsody enables you to connect to the Tornado IDE, download an executable component to the target, and perform source-code level debugging on the target while simultaneously performing design-level debugging on the Rational Rhapsody host.

Integration of the Tornado IDE provides a seamless development workflow between Rational Rhapsody and Tornado through the following functions:

- ◆ Downloading and reloading an image directly to the target from Rational Rhapsody.
- ◆ Synchronizing Rational Rhapsody breakpoints and Tornado breakpoints:
 - Tornado (gdb) is aware of Rational Rhapsody-based breakpoints (break on state).
 - Rational Rhapsody is alerted for source-level breakpoints from Tornado.

This appendix provides information on the following topics:

- ◆ [Preparing the Tornado IDE](#)
- ◆ [IDE operation in Rational Rhapsody](#)
- ◆ [Co-debugging with the Tornado debugger](#)
- ◆ [IDE properties](#)

Preparing the Tornado IDE

To prepare the Tornado IDE for integration with Rational Rhapsody:

1. Open Tornado, then select **Tools > TargetServer > (server name)**.
2. In Rational Rhapsody, make sure the active configuration is set to VxWorks.

IDE operation in Rational Rhapsody

Rational Rhapsody directly supports the following operations from the IDE submenu of the **Code > Target** menu.

- ♦ **Code > Target > Connect** opens a connection to the IDE server. In the case of Tornado, this is the Tornado target server. This should be applied once during a session. The connection is disconnected either explicitly or when the project is closed. To connect, you must specify a target server name, typically <targetName>@<hostName>. Once specified, the name is stored for future sessions.
- ♦ **Code > Target > Download** downloads an image to the targets through the IDE. Download is available only if an image exists.
- ♦ **Code > Target > Run.** Runs the executable on the target starting from a designated entry point (VxMain in Tornado), as specified by the <lang>_CG::<Environment>::EntryPoint property. You can also run the executable from the Code menu or toolbar.
- ♦ **Code > Target > Unload** clears the target from all tasks and data allocated by the application. This is an important feature because RTOSes generally do not have a process concept that cleans up after termination of the application. Unload is always available and causes execution to stop if the application is running.
- ♦ **Code > Target > Disconnect** disconnects from the IDE server.

Co-debugging with the Tornado debugger

Before using the Tornado debugger, make sure to compile the generated file using debug flags (normally -g).

To use the Tornado debugger:

1. In Rational Rhapsody, connect the application by selecting **Code > IDE > Connect**.
2. Download the application by selecting **Code > IDE > Download**.
3. Select **Code > IDE > Run**, or click **Run**.
4. In the **Animation** toolbar, select **Go Idle** (or **Go Step** several times) so the tRhp task is created.

Note: You must run the application before attaching a debugger; otherwise, there will be no tasks to which to attach the debugger.

5. In Tornado, start the debugger by selecting **Tools > Debugger**.
6. Attach the debugger to the main thread (tRhp) by selecting **Debug > attach**.

7. From the debugger, change directory to the generated code directory (using the `cd` command in the **gdb** prompt).
8. From the debugger, load the symbols of the executable (using the `add-symbol-file` command at the **gdb** prompt).

Now you can use **gdb** to debug the application, set breakpoints, and so on.

Before quitting animation on Rational Rhapsody, you must detach the debugger using **Debug > Detach**. Failing to detach the debugger might block the session once Rational Rhapsody attempts to unload the image.

Note

Do not download the executable to the target using the debugger. Rational Rhapsody will not function properly if you use this method.

IDE properties

The following Rational Rhapsody properties (under `<lang>_CG::<Environment>`) determine IDE settings:

- ◆ `HasIDEInterface`

If this property is set to `Cleared`, no IDE services are attempted and IDE support is disabled. If the property is `Checked`, it is expected that the `IDEInterfaceDLL` property points to an IDE adapter that provides connection to the IDE. This property can be used to disable the IDE connection. By default, it is set to `Checked` only for the `VxWorks` environment.

- ◆ `IDEConnectParameters`

This property specifies the IDE connection parameters. If this property is defined, Rational Rhapsody will use the connection parameters from this property instead of the `.ini` file.

- ◆ `IDEInterfaceDLL`

This property points to the IDE adapter DLL. Currently, there is no reason to modify the value of this property.

Creating Rational Rhapsody SDL blocks

Systems engineers often use the System Design Language (SDL) to model discrete (event driven) algorithms. The SDL Suite also generates C code for its models. Rational Rhapsody in C++ is integrated with the SDL Suite (version 5.0 or greater) to enable system simulation based on Rational Rhapsody and the SDL Suite's discrete behavior. Engineers can import an SDL model into Rational Rhapsody. Rational Rhapsody manages the imported model as a class, stereotyped with the **SDLBlock**.

Note

The naming convention for an SDL signal adds the “_” prefix to the signal's original name. This prefix can be modified by changing the `SDLSignalPrefix` property in the `Model::Profile` group.

By default the **SDLBlock** uses behavioral ports. This configuration can be changed to use a rapid port instead by selecting the `UseRapidPorts` property for the package. This property is also stored in the `Model::Profile` group that you access from the **Properties** tab of the Features window.

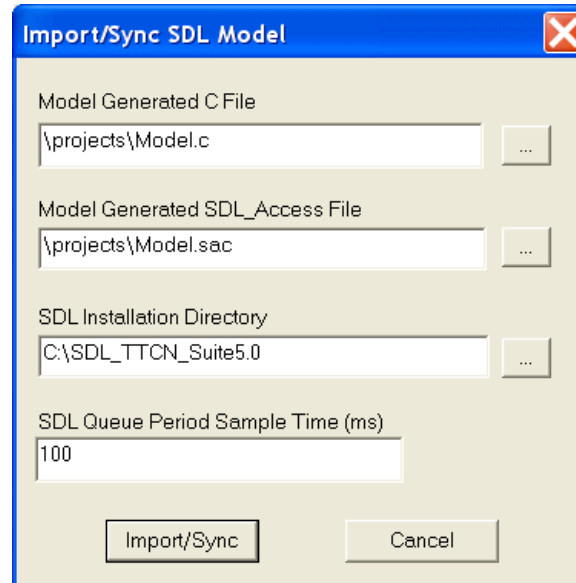
Note

The SDL models you import into Rational Rhapsody cannot contain more than a single instance of any given process.

To import an SDL model into Rational Rhapsody:

1. In the SDL Suite, open the SDL model. Mark the System level rectangle.
2. From the main menu select **Generate > Make**.
3. Select the **CAdvanced** Code Generator configuration.
4. Select the Generate environment header file check box.
5. Activate the “Make” to generate the model C file (modelname.c) and environment header file (modelname.ifc).
6. Select **SDLAccess** Code Generator configuration and activate the **Full Make** to generate the model `SDL_Access` file (modelname.sac).
7. Open Rational Rhapsody and choose **File > New**.
8. Select the **SDL_Suite** for the project **Type**.
9. Create a new block/class and select the **SDLBlock** class stereotype.
10. Right-click this block and select **Import/Sync SDL Model**.

11. Enter the locations of the SDL model files you created previously, as shown in the following example:



12. Click **Import/Sync**.
13. To connect the Rational Rhapsody block to an SDLBlock, create a user class with behavior ports and a statechart. The statechart controls the user class' sending and receiving of events to and from the SDLBlock.
14. Create objects from the SDLBlock and the Rational Rhapsody block and connect their ports via links using the interfaces that were created by the import.
15. To create an executable, perform a code generation and build on the entire Rational Rhapsody model. Code generation scope should contain only one SDLBlock.

Note

Since the SDLBlock is imported as a “black box,” no animation is provided with this block. There is an option to view the behavior of the SDLBlock as a wrapper via a sequence diagram. This can be done by checking the `AnimateSDLBlockBehavior` property, located in the `Model::Profile` property group.

Model elements

The Rational Rhapsody browser lists all the design elements in your model in a hierarchical, expandable tree structure, enabling you to easily navigate to any object in the model and edit its features and properties. The Rational Rhapsody browser also takes part in animation by displaying the values of instances as they change in response to messages and events.

To help you manage large and complex Rational Rhapsody projects, and to be able to focus on and easily access model elements of particular interest to you, you can filter the Rational Rhapsody browser or create other browser views.

Browser techniques for project management

The Rational Rhapsody browser displays a list of project elements organized into folders. You can choose to have all elements displayed in a single folder, regardless of their position in the model, or have them displayed in subfolders based on the model hierarchy. The browser provides several views so you can filter the display of elements by different design categories. The project is the top-most folder in the Rational Rhapsody browser. It contains the following top-level folders:

- ◆ **Components**, which contains one or more configurations and files
- ◆ **Packages**, which contains actors, classes, events, globals, diagrams, types, use cases, and other packages
- ◆ **Diagrams**, which contains any of the UML diagrams that Rational Rhapsody supports

You can organize large projects into package hierarchies that can be viewed easily by nesting packages, components, and diagrams inside other packages, and nesting classes and types inside other classes.

Because a Rational Rhapsody project can get quite large and complex, you might want to filter what you see on the Rational Rhapsody browser or otherwise create other browser views.

Opening the Rational Rhapsody browser

By default, the Rational Rhapsody browser is displayed the first time you open a project. In subsequent work sessions, Rational Rhapsody consults your workspace file (<project name>.rpw) to determine whether to open the browser when it opens a project. For more information on workspaces, see [Controlling workspace window preferences](#).

To open the Rational Rhapsody browser manually, use any of the following methods”

- ◆ Click the Show/Hide Browser button  on the Rational Rhapsody **Windows** toolbar.
- ◆ Select **View > Browser**.
- ◆ Press **Alt+0** (zero).

Browser display options

The browser has two display modes: Flat and Categories. In Flat mode, only the components, packages, and diagrams within each package have separate categories. In Categories mode, all elements are organized into categories based on their position in the project hierarchy. The default display mode is Categories mode.

The display mode for the project is stored in the `Browser::Settings::DisplayMode` property. Set the property to `Meta-class` for Categories mode or `Flat` for Flat mode.

Setting the Organize Tree mode to flat

To hide the categories in the browser:

1. Position the cursor in the Rational Rhapsody browser.
2. Choose **View > Browser Display Options > Organize Tree > Flat**.

Expanding a category while in Flat mode reveals a flat list of elements of all types included under that category. For example, expanding a package category reveals a simple list of the elements contained in the package, such as actors, classes, and events, arranged alphabetically by name.

Setting the Categories mode

In Categories mode, each metatype displays in its own category. Individual items, such as components, diagrams, packages, classes, and actors, are displayed under the appropriate category.

1. Position the cursor in the Rational Rhapsody browser.
2. Choose **View > Browser Display Options > Organize Tree > Categories**.

Displaying model element labels

To display the labels defined for model elements (instead of their names), choose **View > Browser Display Options > Show Labels**.

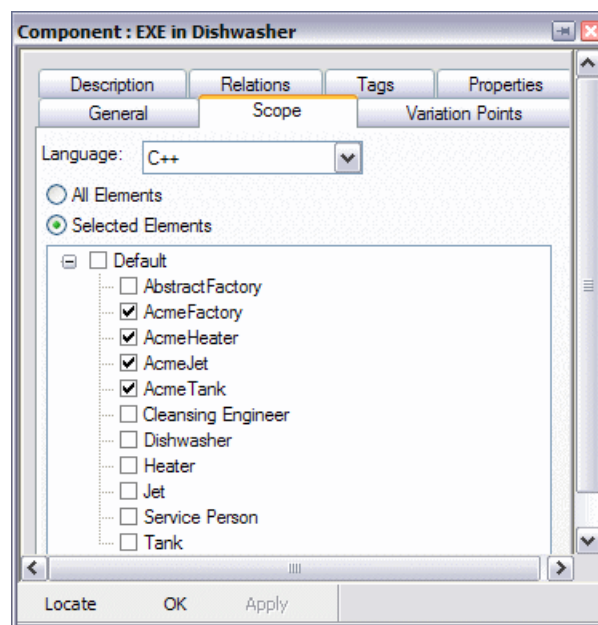
Showing the implementation arguments

To display the implementation arguments in the browser, choose **View > Browser Display Options > Show Implementation Argument**.

Setting the project scope

You might want to divide the system into multiple components, so that each component represents a physical subsystem. To select the model elements for each component:

1. In the Components project folder, right-click the component.
2. Select **Features** and the **Scope** tab
3. To select all of the elements in the component, click the **All Elements** radio button.
4. To select individual elements to be included in the component, click the **Selected Elements** radio button and click the check boxes of the elements, as shown below.

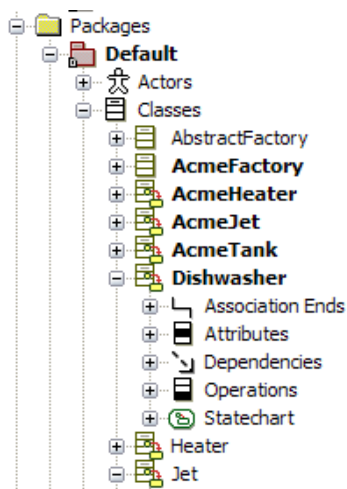


If you select a check box for an element, all of the elements that it contains are included in the component scope (for example, all of the classes in a package). If you want to be able to select sub-elements individually, right-click the check box of the parent element.

Showing the active component elements within the project scope

To show the active component elements in bold type in the browser:

1. Open the Features window. Choose **File > Project Properties**.
2. Navigate to the `General::Model::HighlightElementsInActiveComponentScope` property and select the check box.
3. Click **OK**. The elements of the active component within the scope are shown in bold type, as shown in the following figure:



Basic browser icons

The icons before elements listed in the Rational Rhapsody browser provide additional information about the elements so that you can quickly identify items you want to access. The following table summarizes the standard icons for the Rational Rhapsody browser, but not the icons exclusive to the speciality profiles, such as DoDAF. See the sections describing special features for the browser icons that are available.

Icons	Name (if available)	Description
	Folder	Used to group and organize project elements.
	Folder that is a unit	The square in the lower left corner indicates that this folder is a unit. For information, see Using project units .
	Folder of hyperlinks	For more information, see Hyperlinks .
	Component	A physical subsystem in the form of a library or executable program or other software components such as scripts, command files, documents, or databases.
	Component that is a unit	The square in the lower left corner indicates that this component is a unit.
	Active component	Component that is a unit and is set as the active component with a red check. To be the active component, a component must be either an Executable or a Library build type. For more information, see Active component .
		Executable <i>application</i> or a <i>library</i>
	Actor	Represents an end user of the system, or an external component that sends information to or receives information from the system. For more information, see Actors .
	Class	Defines the attributes and behavior of objects. For more information, see Classes .
		Class with a <i>statechart</i>
		Class <i>attributes</i>
		Class <i>operations</i>
	File	Indicates an imported file.
	Part	A component or artifact of a system.

Icons	Name (if available)	Description
	Event	An asynchronous, one-way communication between two objects (such as two classes). For more information, see Events and operations .
	Use case	Captures scenarios describing how the system could be used. For more information, see Use cases .
	SuperClass	Marks a class that inherits from another class. For more information, see Inheritance .
	Dependency	Notes the relationship of a dependent class to a model element or external system that must provide something required by the dependent class. For more information, see Dependencies .
	Constraint	Supplies information about model elements concerning requirements, invariants in a text format.
	Stereotype	A type of modeling element that extends the semantics of the metamodel. Stereotypes must be based on certain existing types or classes in the metamodel. For more information, see Stereotypes .
	Type	A stereotype of class used to specify a domain of instances (objects) together with the operations applicable to the objects. A type cannot contain any methods. For more information, see Types .
	State	An abstraction of the mode in which the object finds itself. It is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. For more information, see States .
	Initial Connector	Marks the default state of an object. For more information, see Transitions .
	Controlled file	Files produced in other programs, such as Word or Excel, that are added to a project for reference purposes and then controlled through Rational Rhapsody. For more information, see Controlled files .
	Association	Defines a semantic relationship between two or more classifiers that specify connections among their instances. It represents a set of connections between the objects (or users). For more information, see Creating associations .
	Flow port	Represents the flow of data between blocks in an object model diagram (OMD) without defining events and operations. Flowports can be added to blocks and classes in object model diagrams.
	Profile	Applies domain-specific tags and stereotypes to all packages available in the workspace. For more information, see Profiles .
	Requirement	Is a wanted feature, property, or behavior of a system. Requirements can be imported or created in Rational Rhapsody.
		Requirement verification
	Tags	Adds information to certain kinds of elements to reflect characteristics of the specific domain or platform for the modeled system. For more information, see Use tags to add element information .

Icons	Name (if available)	Description
		Flow item
	Comments	Marks text added to a model element.
	Table layout	Shows the design for a table view of project data. For more information, see Table and matrix views of data .
	Table view	Displays project data in the predefined table layout.
	Matrix layout	Shows the design for a matrix view of project data.
	Matrix view	Displays project data in the predefined matrix layout.
	Activity diagram	Shows the lifetime behavior of an object, or the procedure that is executed by an operation in terms of a process flow, rather than as a set of reactions to incoming events. For more information, see Activity diagrams .
	Collaboration diagram	Displays objects, their messages, and their relationships in a particular scenario or use case. This diagram is also a unit. For more information, see Collaboration diagrams .
	Component diagram	Specifies the files and folders that components contain and defines the relations between these elements. For more information, see Component diagrams .
	Deployment diagram	Shows the configuration of run-time processing elements and the software component instances that reside on them. For more information, see Deployment diagrams .
	Object model diagram	Shows the static structure of a system: the objects in the system and their associations and operations, and the relationships between classes and any constraints on those relationships. For more information, see Object model diagrams .
	Requirements diagram	Shows requirements imported from other software products or created in Rational Rhapsody and illustrate the relationships between requirements and system artifacts. For more information, see Creating Rational Rhapsody requirements diagrams .
	Sequence diagram	Describes message exchanges within your project. For more information, see Sequence diagrams .
	Statechart	Defines the behavior of objects by specifying how they react to events or operations. For more information, see Statecharts .
	Structure diagram	Models the structure of a composite class; any class or object that has an object model diagram can have a structure diagram. For more information, see Structure diagrams .
	Use case diagram	Illustrates scenarios (use cases) and the actors that interact with them. The icon in this example indicates that this use case diagram is also a unit. For more information, see Use case diagrams .

Rational Rhapsody browser menu options

The large size and nested hierarchy of a Rational Rhapsody project might complicate the process of locating and working with model elements. To help you navigate the Rational Rhapsody browser more easily, the browser has a filtering mechanism that you can use to display only the elements relevant to your current task.

To display the filter menu, click the down arrow button at the top of the browser. Whatever view you have selected is reflected in the label to the left of the arrow button.

Select one of these views:

Note

If the browser is filtered, you can add only elements that appear in the current view.

- ♦ **Entire Model View** is the default view and it does not use a filter. It displays all model elements in the browser.
- ♦ **Use Case View** displays use cases, actors, sequence diagrams, use case diagrams, and relations among use cases and actors.
- ♦ **Component View** displays components, nodes, packages that contain components or nodes, files, folders, configurations, component diagrams, and deployment diagrams. This view helps you manage different components within your project and assists with deploying them.
- ♦ **Diagram View** filters out all elements except diagrams. It displays all the diagrams, including statecharts and activity diagrams.
- ♦ **Unit View** displays all the elements that are also units. This view assists with configuration management. For information on creating and working with units, see [Using project units](#).
- ♦ **Loaded Units View** displays only the units that have not been loaded into your workspace. For more information, see [Loading and unloading units](#) and [Unloaded units](#).
- ♦ **Requirement View** displays only those elements with requirements.
- ♦ **Overridden Properties View** displays only those elements with overridden properties.

To learn about other browser views, see [The Browse From Here browser](#) and [The Favorites browser](#).

Deleting items from the Rational Rhapsody browser

You can delete items from your project through the main Rational Rhapsody browser and the Browse From Here browser (see [The Browse From Here browser](#)).

To delete an item from your project:

1. Select the items in the main Rational Rhapsody browser or the Browse From Here browser.
2. Right-click and select **Delete from Model** or choose **Edit > Delete** from the main menu.

The system asks for confirmation of the deletion operation.

3. Click **OK**.

Note

Delete from Model is not available for the Favorites browser (see [The Favorites browser](#)). You can select one or more items and choose **Edit > Delete** from the main menu to delete items from the Favorites browser (which is the same if you right-click and select **Remove from Favorites** from the pop-up menu). However, this only means that you are removing items from the Favorites browser. You cannot delete any model elements through the use of the Favorites browser.

The Browse From Here browser

Rational Rhapsody projects can become very large and complex, making it difficult to find commonly used model elements in the Rational Rhapsody browser. To help limit the scope of the current view of the browser, you can open a Browse From Here browser that contains the view of the browser that you want. The Browse From Here browser is similar to the main Rational Rhapsody browser except that it typically shows a more focused area of the main Rational Rhapsody browser.

The Browse From Here browser operate in the same manner as the main Rational Rhapsody browser, and it has the same look-and-feel. You can drag-and-drop between the main Rational Rhapsody browser and one or more Browse From Here browsers.

The one feature that a Browse From Here browser has that the main Rational Rhapsody browser does not is the Up One Level button .

For another method to help you view and access only those model elements you are most interested in, see [Rational Rhapsody browser menu options](#) and [The Favorites browser](#). Note that you can have the main Rational Rhapsody browser, the Favorites browser, and one or more Browse From Here browser open at any time.

Opening a Browse From Here browser

Note that you can open multiple Browse From Here browsers.

To open a Browse From Here browser, right-click a model element in the main Rational Rhapsody browser, the Favorites browser, on a diagram, or another Browse From Here browser and select **Browse from here**.

Note

Browse From Here is not available for elements inside sequence diagrams and collaboration diagrams that are view-only elements.

Closing a Browse From Here browser

To close a Browse From Here browser, click the Close button  for that browser.

Note that **View > Browser** is only for the main Rational Rhapsody browser.

Navigating a Browse From Here browser

You navigate a Browse From Here Browser as you would the main Rational Rhapsody browser.

To set the root to the parent of the current root, click the Up One Level button . Note that, if you want, you can click this button as many times as needed to go to the project root folder (so that your Browse From Here Browser ends up looking exactly like your main Rational Rhapsody browser).

Deleting items from the Browse From Here browser

Just like the main Rational Rhapsody browser, you can delete items from your project through the Browse From Here browser. See [Deleting items from the Rational Rhapsody browser](#).

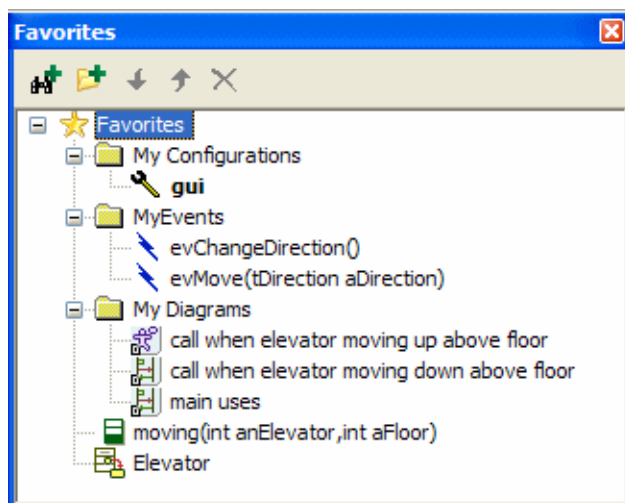
Browse From Here browser limitations

Browse From Here browsers are not saved when you close your project. Meaning that they will not be opened or available when you open your project the next time.

The Favorites browser

You can use the Favorites browser to create a favorites list, which is a list of items (model elements) that you are most interested in for the opened Rational Rhapsody model. This is analogous to the favorites functionality for a Web browser. You might find the Favorites browser most useful with Rational Rhapsody models that are very large, which can make it difficult to find commonly used model elements in the Rational Rhapsody browser. The Favorites browser should help you manage large and complex projects by making it easier to focus on and easily access model elements of particular interest to you.

The following figure shows a sample Favorites browser:



Note

The Favorites browser is available only for the stand-alone version of Rational Rhapsody

While the Favorites browser resembles the Rational Rhapsody browser and has some of its functionality (for example, double-clicking an item on the Favorites browser opens the applicable Features window, the view is refreshed when an item is renamed, it can be docked and undocked, and so on), the Favorites browser has limited functionality (for example, there is no filtering mechanism, certain commands, such as **Cut**, **Copy**, **Paste**, **Add New**, and **Unit** are not available, and while you can remove items from the Favorites browser, you cannot use it to delete a model element from your model).

Note

You cannot use the Favorites browser as a replacement for the Rational Rhapsody browser.

Your favorites list is saved in the <projectname>.rpw file, while the visibility and position of the Favorites browser are saved in the Rhapsody.ini file, so that when you open the project the next time, your settings will automatically be in place. When multiple projects are loaded, the Favorites browser shows the favorites list for the active project.

Favorites toolbar

The **Favorites** toolbar provides tools for the Favorites browser. To display or hide this toolbar, choose **View > Toolbars > Favorites**.

The **Favorites** toolbar includes the following tools:

Tool Button	Name	Description
	Show/Hide Favorites	Toggles between showing and hiding the Favorites browser.
	Add to Favorites	Select a model element and then click this button to add what you selected to your Favorites browser.

Showing and hiding the Favorites browser

To show the Favorites browser, use any of the following methods:

- ◆ Click the Show/Hide Favorites button  on the **Favorites** toolbar. The button acts as an on/off toggle.
- ◆ Select **View > Favorites**. This menu command acts as an on/off toggle.
- ◆ Add an item to your favorites list. This automatically opens the Favorites browser. See [Creating your Favorites list](#).

To hide the Favorites browser, use any of the following methods:

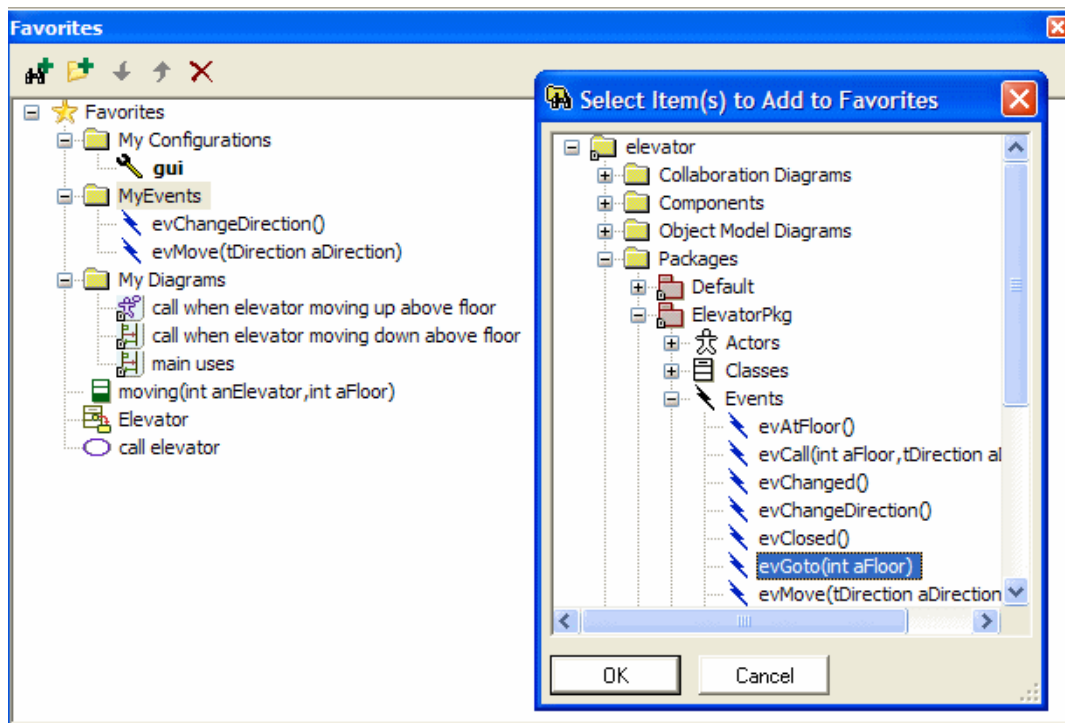
- ◆ Click the Close button  for the browser.
- ◆ Click the Show/Hide Favorites button  on the **Favorites** toolbar.
- ◆ Select **View > Favorites**.

Creating your Favorites list

The Favorites browser automatically opens when you add an item to it.

To create your favorites list, use any of the following methods:

- ◆ Select a model element in the Rational Rhapsody browser, the Browse From Here browser, or on a diagram and click the Add to Favorites button  on the **Favorites** toolbar. (This toolbar should display by default. If it does not, choose **View > Toolbars > Favorites**).
- ◆ Right-click a model element in the Rational Rhapsody browser or on a diagram and select **Navigate > Add to Favorites**.
- ◆ Select a model element in the Rational Rhapsody browser, the Browse From Here browser, or on a diagram and press **Ctrl+d**.
- ◆ With the Favorites browser open, click the Add to Favorites button  and then select one or more items to add from the Select Items to Add to Favorites window. Note that with this method, if you have a folder structure for your favorites list, you can specify where to put a new favorite by selecting that folder (or the root node, **Favorites**) first in the Favorites browser. Otherwise, your favorite is added to below the non-folder item you have highlighted on the Favorites browser. See [Creating a folder structure for your Favorites](#).



- ◆ With the Favorites browser open, right-click the root node (**Favorites**) or a folder and choose **Select Item to Add**. See [Creating a folder structure for your Favorites](#).
- ◆ Click a model element on the Rational Rhapsody browser, the Browse From Here browser, or on a diagram and drag it onto the Favorites browser. You can drop the dragged item anywhere on the Favorites browser. See [Re-ordering the items on your Favorites list](#).

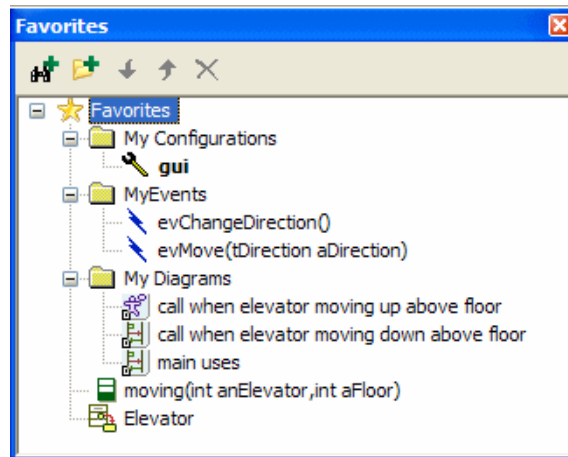
Note

You cannot put a model element on your favorites list more than once. For example, if you have ElementA in FolderA and you add ElementA to FolderB, the element is removed from FolderA so that it can reside in FolderB. See [Creating a folder structure for your Favorites](#).

In addition, **Add to Favorites** is not available for elements inside sequence diagrams and collaboration diagrams that are view-only elements.

Creating a folder structure for your Favorites

You can have a flat file structure for your favorites or you can have a hierarchical folder structure.



To create a folder structure for your favorites list:

1. Open the Favorites browser (see [Showing and hiding the Favorites browser](#)).
2. Wherever you want to add a folder, click the New Folder button  or right-click an item on the Favorites browser and select **Add New Folder**. At this time, you can name your folder.
3. Once you have added an item (at any level), you can add a subfolder for that item if you want.

Re-ordering the items on your Favorites list

To re-order the items on your favorites list, use any of the following methods:

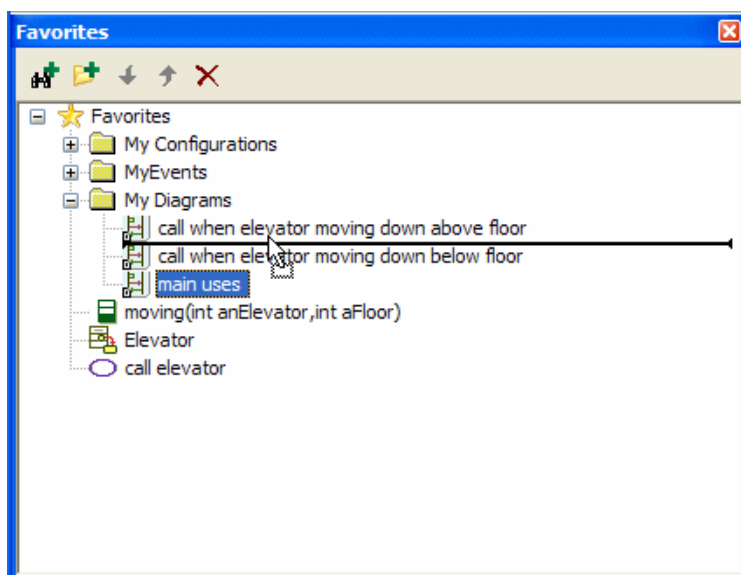
- ◆ Select an item on your favorites list and click the Move Up  or Move Down  buttons on the Favorites browser.

Note: This method is not available for folders.

- ◆ Right-click an item on your favorites list and select **Move Up In Favorites** or **Move Down in Favorites**.

Note: This method is not available for folders.

- ◆ Select one or more items on your favorites list and drag it to where you want to drop it. An insertion line displays to indicate where you can drop it.



- ◆ To drop an item into an empty folder or the root node (**Favorites**), move your pointer over the folder so that it becomes highlighted and then drop your item.

Removing items from your Favorites list

When you remove items from your favorites list, you are only removing them from the Favorites browser. You are not deleting them from your model.

Note

The item is removed once you execute the action. If you change your mind, you can add the item to your favorites list again (see [Creating your Favorites list](#)).

To remove items from your favorites list, select one or more items on the Favorites browser and then use any of the following methods:

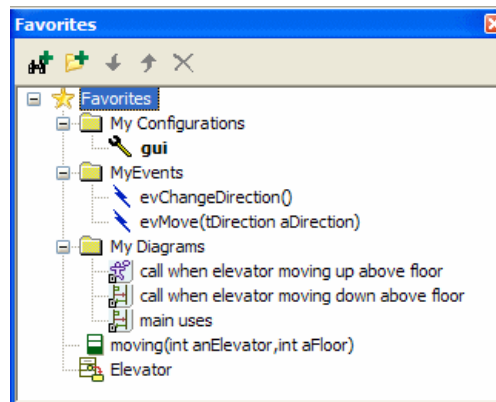
- ◆ Click the Delete Favorites button  on the Favorites browser.
- ◆ Press the **Delete** key.
- ◆ Right-click and select **Remove from Favorites**.
- ◆ Choose **Edit > Delete**.

If there are items in a folder to be removed, the system informs you of this and ask if you want to remove the folder (and its items).

Favorites browser limitations

Note the following limitations for the Favorites browser:

- ♦ You cannot give a name that is different from the referenced element to a favorite (like you can for a hyperlink or a favorite in a regular Web browser).
- ♦ Since you can put different model elements with the same name on your favorites list, there is a chance for confusion. To try to avoid this, notice the icon to the left of the model element name. The icons provide you with a clue as to what type of model item you have on your favorites list.



Elements

Primary model *elements* within the browser are packages, classes, OMDs, associations, dependencies, operations, variables, events, event receptions, triggered operations, constructors, destructors, and types. Primary model elements in OMDs are packages, classes, associations (links), dependencies, and actors.

Rational Rhapsody in C and C++ classes and their instances are replaced by C equivalent object types and objects, respectively. Similarly, class constructors and destructors are replaced by initializers and cleanup operations.

Adding elements

In the browser, you can add new elements to the model either from the Edit menu or from the pop-up menu. The location you select in the browser hierarchy determines which model elements you can add. The new element is added in the scope of the current selection.

- ◆ Select the project folder or a package, then select **Edit > Add New > Component**.
- ◆ Right-click the project folder or a package and then select **Add New > Component**.

Note

If the browser is filtered, you can add only elements that appear in the current view.

When you add a new association end, a window opens so you can select the related model element.

1. Right-click the actor and then select **Add New > Association End**. The Add Association/Aggregation window opens.
2. From the list, select the actor (or class) with which the current actor needs to communicate.
3. Click **OK**.

When you create a new element, Rational Rhapsody gives it a default name. You can edit the name of your new element by typing a new name directly in name field in the browser, or by opening the Features window and entering a new name in the **Name** field.

Naming new elements in the browser

To change the automatically generated name of a new element:

1. Click the new element in the browser to open the name for editing.
2. Type the element name that you want to use.

You can also select one of these options for the menu that is available when you are renaming an element:

- ◆ Right to left Reading order
- ◆ Show Unicode control characters
- ◆ Insert Unicode control character (for example, LRM - Left-to-right mark or RLM - Right-to-left mark are available in a selection list)

Browser settings

When you open an existing project in a newer version of Rational Rhapsody, the system adds a compatibility profile in the Settings folder, as shown in this example.



For more information about the Settings folder and profiles, see [Profiles](#).

Components

A *component* is a physical subsystem in the form of a library or executable program. It plays an important role in the modeling of large systems that contain several libraries and executables. For example, the Rational Rhapsody application has several dozen components including the graphic editors, browser, code generator, and animator, all provided in the form of a library.

A component contains configurations and files. In the Rational Rhapsody hierarchy, a component can be saved at the project level, or grouped within a package.

For instructions on editing components, see [Component diagrams](#).

Configurations

A *configuration* specifies how the component is to be produced. For example, the configuration determines whether to compile a debug or non-debug version of the subsystem, whether it should be in the environment of the host or the target (for example, Windows versus VxWorks), and so on. For more information, see [Component diagrams](#).

Configuration files

Configurations manifest themselves as *files*. The ability to map logical elements to files enables you to better specify implementations for code generation, and to capture existing implementations during reverse engineering. Just as it might be desirable to map several classes into a single package, so might it be desirable to map one or more packages into a single subsystem. You control where to generate the source files for the classes (or packages) in a given subsystem, either into a single directory or separate directories.

- ◆ Which logical elements, or classes, to map into which files
- ◆ The order of the classes in the file
- ◆ Verbatim code chunks, such as macros and `#define` statements, that should be included in generated source files
- ◆ Whether to map each class to its own specification and implementation files, or to map multiple classes to the same files
- ◆ Whether specification and implementation files generated for a class, or set of classes, have the same name or different names
- ◆ Dependencies between classes
- ◆ File scope definitions for user-defined code

For more information on files, see [Component diagrams](#).

Packages

A Rational Rhapsody project contains at least one package. *Packages* divide the system into functional domains, or *subsystems*, which can consist of objects, object types, functions, variables, and other logical artifacts. Packages do not have direct responsibilities or behavior; they are simply containers for other objects. They can be organized into hierarchies, providing the system under development with a high level of partitioning. When you create a new model, it will always include a default package, called “Default,” where model elements are saved unless you specify a different package.

These elements are described in detail in subsequent sections.

Classes (C++/J)	Collaboration diagrams
Comments	Sequence diagrams
Constraints	Structure diagrams
Dependencies	Object model diagrams
Flow charts	Statecharts
FlowItems	Tags
Flows	Stereotypes
Functions (C/C++)	Events
Hyperlinks	Component diagrams
Object _types (C)	Activity diagrams
Objects	Actors
Packages	Deployment diagrams
Receptions	Files
Requirements	Use case diagrams
Types (C/C++)	Nodes
Variables	Use cases

Package design guidelines

Packages provide a way to group large systems into smaller, more manageable subsystems. Packages are primarily a means to group classes together into high-level units, but they can also contain diagrams and other packages.

When creating packages, follow these basic guidelines:

- ◆ Do not allow packages to become unmanageably large. Break them into subpackages.
- ◆ Limit dependencies across packages. One way to do this is to use interfaces in their own packages.

Creating a package

To create a package:

1. Right-click the project or the package category, then select **Add New > Package**.
2. Edit the name of the package using the package label in the browser.
3. Double-click the new package to open the Features window.
4. Select a stereotype for the package (if necessary) from the **Stereotype** list.
5. Select an object model or use case diagram as the main diagram for a package from the **Main Diagram** list.
6. Enter a description for the package in the **Description** box.

In the generated code, this text becomes a comment located after the `#define` statements and before the `#include` statements at the top of the `.h` file for the package.

Using functions

The *Functions* category lists global functions, which are visible to all classes in the same package.

Creating a global function

1. Right-click the name of a package or the **Functions** category in the browser.
2. Select **Add New > Function**. A new function displays under the Functions category of the selected package.
3. Edit the name of the new function in the browser.

Changing what a function returns

When a function returns an «existing type», it is returned by type pointer. You can set if the function returns by "value"(), by "pointer"(*) or by "reference"(&).

To change what a function returns:

1. Right-click the function/operation and choose **Features** to open the Features window.
2. On the **Properties** tab, find the `CPP_CG::Type::ReturnType` property.
3. Change the value to any of the following choices:
 - `$Type*` to return by "pointer"(*)
 - `$Type` to return by "value"()
 - `$Type&` to return by "reference"(&)
4. Click **OK**.

Using objects

Objects are instances of classes in the model and can be used by any class. They are typically created in OMDs, but can be added from the browser. For instructions on creating an object in an OMD, see [Object model diagrams](#).

To create an object

1. Right-click the name of a package or the **Objects** category in the browser.
2. Select **Add New > Object**. The new object is displayed in the browser with the default name `object_n`.
3. Optionally, rename the new object.

Using variables

Global variables are visible to all classes in the same package.

Note

Exercise caution when using global variables. If a global variable is used for a counter in a shared library and multiple threads or processes can access the counter, inaccurate results might occur if the global counter is not protected by a mutex or semaphore.

Creating a variable

1. Right-click the name of a package or the Variables category in the browser.
2. Select **Add New > Variable**.

The new global variable is displayed in the Variables category.

Variable ordering in C++

Consider the case where your model has variables defined directly in a package. These variables define various constants (such as `PI` and `DEGREES_PER_RADIAN`). Some of these variables are defined in terms of others, such that the dependent variable must be declared before the others for the application to compile. However, Rational Rhapsody will not allow you to override the default alphabetical order of the variable declarations.

There are at least two ways to solve this problem:

- ◆ Define your constants using types. For example, assume a type named `PI` with the following declaration:

```
const double %s = 3.14
```

In this syntax, the `%s` is replaced with the name `PI`.

A `DEGREES_PER_RADIAN` type would have the following declaration:

```
const double %s = 180.0 / PI
```

In this syntax, the `%s` is replaced with the name `DEGREES_PER_RADIAN`.

Because Rational Rhapsody allows you to change the order of the type declarations such that `PI` is generated first, the compilation is successful.
- ◆ Create a variable called `PI` of type `const double` with an initial value of `3.14`. Create a second variable called `DEGREES_PER_RADIAN` of type `const double` with an initial value of `180.0 / PI`. This will not compile because Rational Rhapsody generates the `DEGREES_PER_RADIAN` variable before the `PI` variable.

On the `DEGREES_PER_RADIAN` variable, set the `CPP_CG::Attribute::VariableInitializationFile` property to `Implementation to`

initialize the variable in the implementation file. The default setting (Default) causes the initialization to be put in the specification file if the type declaration begins with `const`; otherwise, it is placed in the implementation file.

Now, your application will compile correctly.

Dependencies

Dependencies are relations in which one class (the dependent) requires something provided by another (the provider). Rational Rhapsody supports the full dependency concept as defined in the UML. Dependencies can exist between any two elements that can be displayed in the browser. Dependencies can be created in diagrams or in the browser.

Constraints

UML *constraints* provide information about model elements concerning requirements, invariants, and so on. Rational Rhapsody captures them in text format. For more information, see [Annotations for diagrams](#).

Classes

Classes define the attributes and behavior of objects. Classes can also be created in the object model or collaboration diagram editors. For more information, see [Creating classes](#).

Types

The Types category displays user-defined, rather than predefined, data types. Rational Rhapsody enables you to create types that are modeled using structural features instead of verbatim, language-specific text.

Receptions

A *reception* specifies the ability of a given class to react to a certain event (called a *signal* in the UML). Receptions are inherited. If you give a trigger to a transition with a reception name that does not exist in the class but that exists in the base class, a new reception is not created.

Events

An *event* is an asynchronous, one-way communication between two objects (such as two classes). Events inherit from the `OMEvent` abstract class, which is defined in the Rational Rhapsody framework. Events can be created in sequence diagrams, collaboration diagrams, statecharts, or the browser.

Actors

An *actor* represents an end user of the system, or an external component that sends information to or receives information from the system. Actors can be created in UCDs, OMDs, collaboration diagrams, and the browser.

Use cases

A *use case* captures scenarios describing how the system could be used. It usually represents this scenario at a high conceptual level. Use cases can also be created in the use case diagram editor.

Nodes

A *node* represents a computer or other computational resource used in the deployment of your application. For example, a node can represent a type of CPU. Nodes store and execute the run-time components of your application. In the model, a node can belong only to a package, not to another node (that is, nodes cannot be nested inside other nodes).

Files

A *file* is a graphical representation of a specification (.h) or implementation (.c) source file. This new model element enables you to use functional modeling and take advantage of the capabilities of Rational Rhapsody (modeling, execution, code generation, and reverse engineering), without radically changing the existing files. For more information, see [Configuration files](#).

Diagrams

A project has separate categories in the browser for object model, sequence, use case, component, deployment, structure, and collaboration diagrams. Additionally, all of these diagrams (except collaboration diagrams) can be grouped under a package.

Note: Statecharts and activity diagrams are displayed within the class they describe. To add new statecharts and activity diagrams, select the class name in the browser instead of the package.

Adding diagrams

To add a new diagram to an existing package:

1. Right-click the package in the browser.
2. Select **Add New > Package**.
3. Type the name of the new package in the highlighted area in the browser.
4. Right-click the new package and select the **Add New > Diagrams** submenu.
5. Select the type of diagram you would like to add. The New Diagram window opens.
6. Type a name for the new diagram in the **Name** field.
7. If you want to populate the new diagram automatically with existing model elements. Click the **Populate Diagram** check box.
8. Click **OK**. If you selected **Populate Diagram**, another window displays to allowing you to select which model elements to add to the diagram. Rational Rhapsody automatically lays out the elements in an orderly and easily comprehensible manner.
9. The new diagram displays in the drawing area of the Rational Rhapsody window.

Locating an element on a diagrams

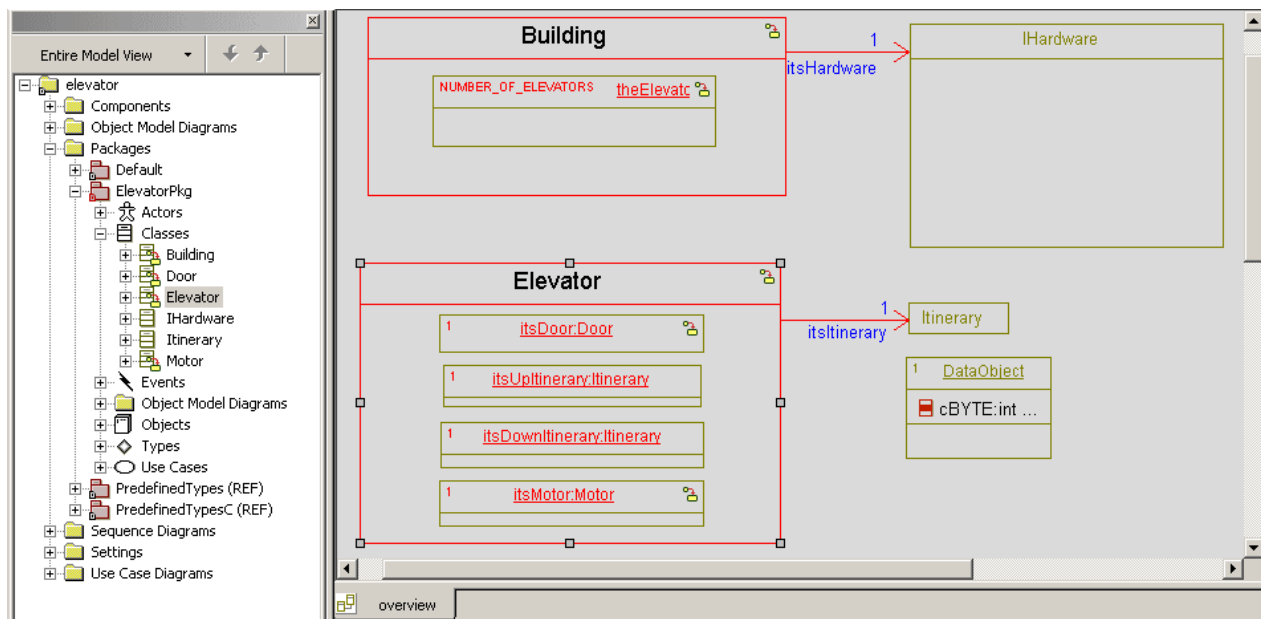
You can use the Rational Rhapsody browser and any code view window to quickly locate an element on a diagram by using **Locate On Diagram** from the pop-up menu.

To use the Rational Rhapsody browser to locate an element on a diagram:

Note: These steps are the same from any code view window, except that you would select the element from the code. For an example, see [Example of Locate On Diagram from code view](#).

1. Right-click an element in the Rational Rhapsody browser and select **Locate On Diagram**.
2. Notice that Rational Rhapsody opens a diagram in which the element (**Elevator**) is displayed.

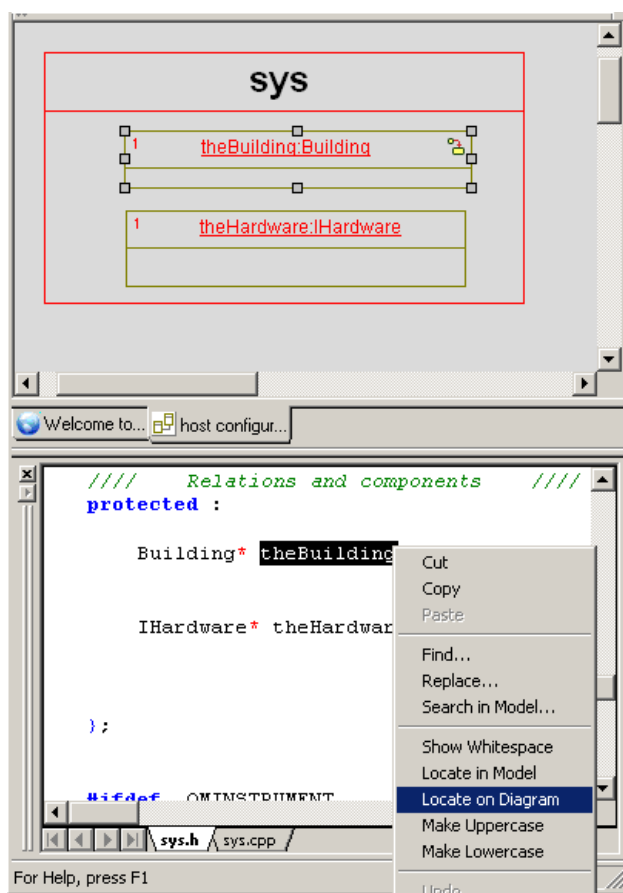
Note: If no diagram exists, a No Diagram Found message displays instead.



Example of Locate On Diagram from code view

The following figure shows an example of selecting **Locate On Diagram** from a code view (as shown in the lower half of the figure) and the diagram it found (as shown in the upper half of the figure).

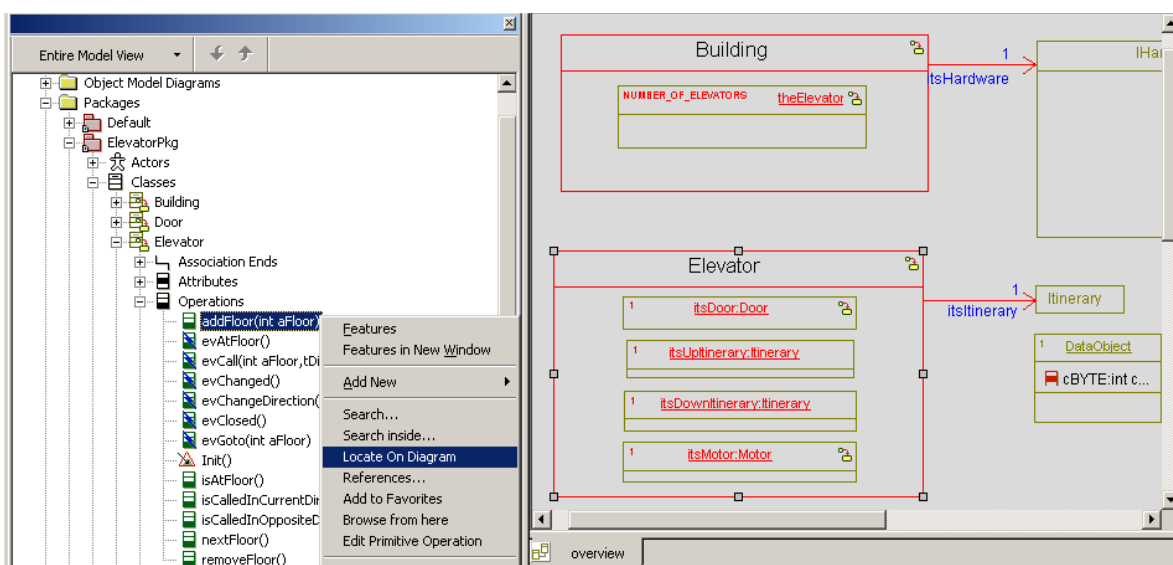
Note: The **theBuilding** element in the code view half of the following figure is selected for illustrative purposes. You can place your cursor within the letters of the element name or that particular line of code and right-click to select **Locate On Diagram**.



Locate On Diagram rules

Locate On Diagram uses the following rules to determine what to open, in order of priority:

1. Open the main diagram of the element if the main diagram is set and does show the element (as illustrated in the previous figure).
2. For an object, show the main diagram of its class if the main diagram is set and does show the object.
3. Show a randomly chosen diagram that shows the element.
4. Look for its container only if it does not find the element on the diagram. For example, if you try to locate an operation, Rational Rhapsody looks for a diagram that shows the class for the operation.



Locate On Diagram limitations

The ability to use **Locate On Diagram** from the code view is based on the annotations in the code. If the code is not annotated, **Locate On Diagram** will not be able to find the diagram. Note also in this case that it will not display a No Diagram Found message.

Element identification and paths

Packages and classes serve as containers for primary element definitions, in other words, recursive composites or namespaces. Each primary model element is uniquely identified by a path in the following format:

`<ns1>::<ns2>::...::<nsn>::<name>`

In this format, <ns> can be either a package or a class. Primary model elements are packages, classes, types, and diagrams. Names of nested elements are displayed in combo boxes in the following format:

`<name> in <ns1>::<ns2>::...<nsn>`

You can enter names of nested elements in combo boxes in the following format:

`<ns1>::<ns2>::<name>`

Descriptive labels for elements

Element naming conventions often result in complicated and difficult-to-use names for elements, especially in large projects developed by many different individuals. For that reason, Rational Rhapsody enables you to assign a descriptive label to an element. A label that does not have any meaning in terms of generated code, but enables you to easily reference and locate elements in diagrams and windows. A label can have any value and does not need to be unique, making it possible for you to use non-English labels, reuse labels, and use labels that include spaces on Solaris systems.

Once you have assigned a label, Rational Rhapsody uses it in place of the element name in diagrams and dialogs by default. Reports display both the element name and the element label.

Note

The `General::Graphics::ShowLabels` property controls whether labels are displayed in diagrams and dialogs.

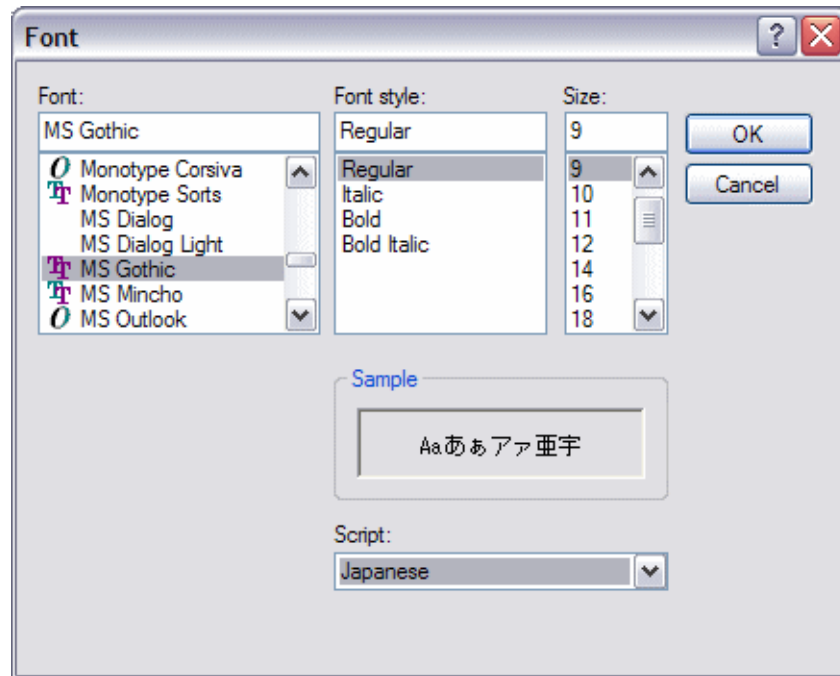
Setting properties for Asian languages

The element labels support Asian language text in Chinese, Korean, Taiwanese, and Japanese. The text input can be through the Features window, browser, or graphic editor, and all non-ASCII characters are stored as RTF instead of plain text.

To set the project properties to use an Asian language font for text input:

1. Right-click on the project name in the browser and select **Features**.
2. Select the Properties tab.
3. From the View menu, select **Filter** and search property names for “font.”
4. Right-click on the font name and description that needs to be changed and click the browse button to display the Font window.
5. Check to be certain that the font supports the Asian language you want to display. If the font does not support the required character set, select a new font from the **Font** list, such as MS Gothic that supports Kanji characters required for Japanese.

6. Also in the Font window, check the **Script** selection and use the pull-down menu to select the correct language, as shown in this example.



7. Click **OK** save the change. Make this same change to any other font properties that are required.

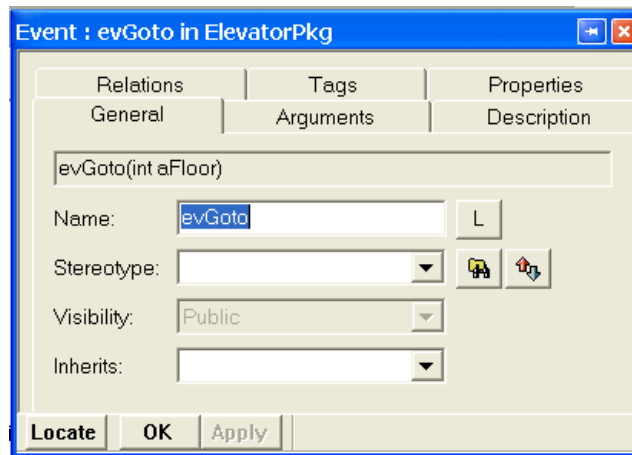
Adding a label to an element

This chart lists all of the element types that can be labeled.

Packages	Association/Aggregation role names
Use cases	Events
Objects	Types
Operations	Activity Flows
Attributes	Constraints
States	Comments
Classes	Requirements
Actors	Files

To add a label to an element:

1. Right-click the selected element in the browser. (You can double-click to display the Features window immediately.)
2. Select **Features** from the menu to display the Features window for the selected element. In this example, the developer is labeling an event.



3. Click the **L** button next to the **Name** field. The Name and Label window displays.
4. Type the text in the **Label** field. The read-only **Name** field displays the name of the element for reference.
5. Click **OK** twice.

Note

Remember that these labels do not have any meaning in the generated code and are used only as references to locate elements in diagrams and windows.

Removing a label from an element

To remove a label from an element:

1. Right-click the selected element in the browser. (You can double-click to display the Features window immediately.)
2. Select **Features** from the menu.
3. Click the **L** button next to the **Name** field to open the Name and Label window.
4. Clear the **Label** field.
5. Click **OK** twice.

Alternatively, you can set the label to have the same value as the Name. In this case, the label becomes tied to the name. In this way, any changes made to the element name also affect the label.

Label mode

Rational Rhapsody differentiates between element names, which are used in the generated code, and element labels, which are just used to represent elements in a user-friendly way within the Rational Rhapsody interface.

Ordinarily, you must provide an element name, even if you are providing a label, and the name must satisfy certain criteria, such as not containing spaces or special characters.

If you would like to deal exclusively with labels, you can select **View > Label Mode**. In this work mode, you only have to provide a label for elements. Rational Rhapsody automatically converts the label to a legal element name, replacing any characters that cannot be used in element names. The text that you entered as the label will be shown in the browser, graphic editor, and the Features window.

Note

When Label Mode is selected, Rational Rhapsody ignores the radio button selected under **Display Name** in the Display Options window. Label Mode is a per-user setting. If the project is opened on a different user's computer, it is displayed in normal mode unless that user has selected Label Mode.

Modify elements

You can move, copy, rename, delete, and re-order elements. In addition, you can display stereotypes of model elements, create graphical elements, and associate an image file with a model element.

Moving elements

To move elements, use any of the following methods:

- ◆ You can move all elements within the browser by dragging-and-dropping them from one location to another, even across packages.
- ◆ You can move a group of highlighted items to a new location in the project by dragging and dropping them into that location in the browser.
- ◆ In addition, you can drag-and-drop packages, classes, actors, and use cases from the browser into diagrams.
- ◆ If code has already been generated for a class or package, you can open the code by dragging and dropping the class or package from the browser into an open editor, such as Microsoft Word™, Visual C++, Borland C++, or Notepad. If the code exists, the standard “+” icon displays as you drag the element into the editor. If the “+” icon does not appear, most likely code has not yet been generated for the element. If the element has both a specification and an implementation file, both files are opened.

See [Smart drag-and-drop](#).

Note

Do not try to open the code for a class or package by dragging it from an object model diagram because this removes the element from the view.

Copying elements

To copy elements, use any of the following methods:

- ◆ You can copy all elements within the browser by dragging-and-dropping.
- ◆ To copy an element, press the **Ctrl** key while dragging the element onto the new owner.

See [Smart drag-and-drop](#).

Copying an element into the same namespace, such as within its own package, creates a copy with the suffix `_copy` appended to its name. For example, copying a class `Sensor` into the same package as the original creates a class named `Sensor_copy`. Subsequent copies have a suffix of `_copy_<n>`, where `<n>` is an incremental integer starting with 0.

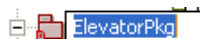
Copying an element into a different namespace creates a copy with exactly the same name.

Renaming elements

Renaming requires a valid name that does not already exist. An appropriate message is displayed if either condition is not met, and the name reverts to its previous value.

There are two methods for renaming elements. You can edit the **Name** box of the Features window or you can edit the name directly in the browser.

1. Click once on the element name to select it.
2. Click once again to open the name field for editing. An edit box opens with the name selected, as shown in the following example.



3. Edit the name and press **Enter**, or click once outside the edit field to complete the change.

The other method you might use is the Features window.

1. Open the Features window for the element.
2. Type the new name in the **Name** box.
3. Click **Apply**, or anywhere outside of the Features window, to apply the new name.
4. Click **OK**.

Note

Do not use this method to rename the project.

Deleting elements

The Edit menu on the diagram editors contains the **Remove** and **Delete** menu commands. **Edit > Remove** removes an object from a particular view but not from the model, whereas **Delete** deletes the item from the entire model. In the browser, **Delete** always deletes an object from the entire model, including all diagrams.

To delete an object from the entire model while in the browser, use any of the following methods:

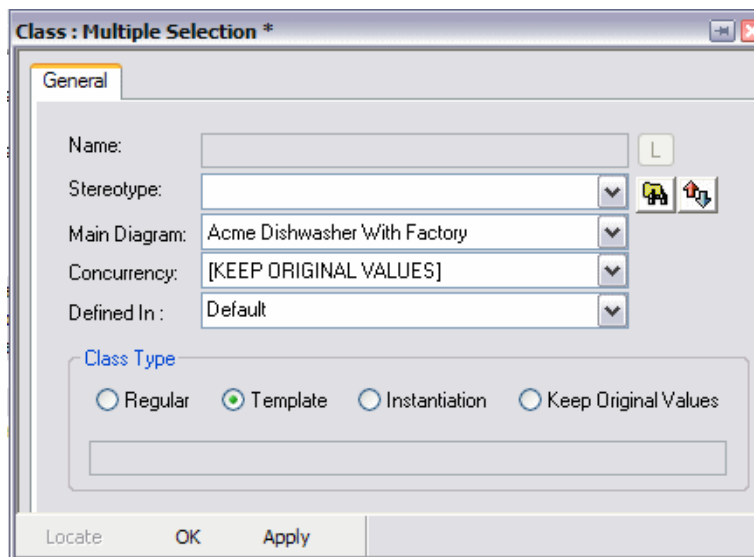
- ◆ Select the item and then choose **Edit > Delete**.
- ◆ Right-click the item, then select **Delete from Model**.
- ◆ Select the item and press the **Delete** key.

If you delete the only item in a category, the entire category displays with it. For example, if you delete the only object model diagram, the entire Object Model Diagrams category displays along with the specific diagram you deleted. The category does not return to the browser until you create at least one object model diagram.

Editing multiple elements

To edit multiple elements of the same metatype at the same time:

1. In the browser, highlight the elements that are all members of the Attributes, Types, Dependencies, Packages, Operations, or Classes metatype.
2. Right-click and select **Features**. The Features window displays as **Multiple Selection** with only the **General** tab accessible. (The Name and Label fields are disabled.)



3. Type the change to common element items such as the stereotype, description, or type definitions.

Note: If all of the selected elements have the same values, the new values are applied to all. However, if the selected elements have different values, the Multiple Selection window uses the **Keep Original Values** option to pinpoint those areas containing varying values. Select this option to keep the different values in the selected elements when the change is applied.

4. Click **OK**.

Re-ordering elements in the browser

By default, model elements are ordered alphabetically within each category. However, Rational Rhapsody also allows you to manually reorder elements within each category.

You might want to reorder elements in the browser in order to:

- ♦ move important items higher in the list
- ♦ control the order of model elements in the generated code. The order of elements in the generated code is determined by the order of display in the browser. To control the order of elements in the generated code, reorder the relevant elements in the browser before generating the code.

Note

Currently, Rational Rhapsody allows you to rearrange all model elements except for statechart-related elements and activity diagram-related elements.

By default, the ability to reorder elements in the browser is disabled. To enable element reordering, select **View > Browser Display Options > Enable Order** from the menu.

When the re-ordering option is available, you can reorder elements:

1. Select the element you would like to move.
2. Click the Up or Down arrow in the browser toolbar to move the element.

Note

The Up and Down arrows are only displayed after you have selected a movable model element. If the element is at the top of a category, only the Down arrow is accessible when you are selecting the element. If the element is at the bottom of the category, only the Up arrow is available upon selection.

Displaying stereotypes of model elements

Rational Rhapsody provides the option of displaying the stereotypes applied to a model element, alongside the name of the element in the browser.

To display elements' stereotypes in the browser, use one of these methods:

- ◆ From the menu, select **View > Browser Display Options > Show Stereotype > All**
- ◆ At the project level, modify the property `Browser::Settings::ShowStereotypes`. The possible values for this property are `No`, `Prefix`, `Suffix`.

Note

When **All** is selected, the property is assigned the value `Prefix`. When **None** is selected, the property is assigned the value `No`. If you want the stereotype name displayed after the element name, this can only be done by changing the property directly to `Suffix`.

If you select the option **First**, then for elements with more than one stereotype, only the first stereotype is displayed in the browser.

The default setting for the property `Browser::Settings::ShowStereotypes` is `Prefix`. For projects created with Rational Rhapsody 6.0 or earlier, this property is overridden and set to `No`.

This settings does not apply to stereotypes that are defined as “new terms.” When a stereotype is defined as a “new term,” it is given its own category in the browser, and any elements to which this stereotype is applied are displayed under the new category.

Creating graphical elements

You can create graphical representations for elements in a model and place them in diagrams. This technique is often used to represent all visible behavior in a use case diagram. This feature is often used to represent stereotypes and tags in diagrams.

To create a graphical representation for an element that is listed in the browser, but not shown in the diagram:

1. Locate the element in the browser that is not currently graphically represented in a diagram. Highlight and drag the element into the diagram.
2. The element displays in the diagram, where you can use the Features window to make any changes and additions to the description for the element.
3. If they can be connected semantically, you might also link the new graphical element to other diagram elements in the diagram. For example, a stereotype can connect to another stereotype for inheritance.

Moving the element into the diagram functions as a move of a class or object.

Supported image formats

The following graphic formats are supported for associated image files: jpeg, tiff, bmp, ico, emf, tga, pcx.

Associating an image file with a model element

Rational Rhapsody allows you to associate an image file with a model element. This image can then be used to represent the element in diagrams in place of the standard graphic representation.

To associate an image file with a model element:

1. Right-click the relevant element in the browser, and then select **Add Associated Image**.
2. When the file browser window is displayed, select the appropriate image file.

Displaying associated images

Once an image file has been associated with a model element, it can be displayed in diagrams that contain that element.

To display the image associated with an element in a diagram, rather than the regular graphical representation:

1. Open the Display Options window.
2. Select **Enable Image View**.

3. Choose **Use Associated Image** or, alternatively, choose **Select an Image** and click ... to open the file browser.

To enable the display of associated images at the diagram level, set the property `General::Graphics::EnableImageView` to `Checked`. The default value is `Cleared`.

When using an associated image in a diagram, there are a number of ways the image can be displayed:

- ♦ **Image Only** displays the image instead of the regular graphic representation.
- ♦ **Structured** displays the image within the regular graphic representation.
- ♦ **Compartment** displays the image within a compartment together with the other items that are displayed in compartments, for example, attributes and operations.

To select one of these layout options:

1. In the Display Options window, click **Advanced**.
2. When the new window opens, select one of the layouts.
3. Click **OK**.

To set the image view layout at the diagram level, select the appropriate value for the property `General::Graphics::ImageViewLayout`. The default value is `ImageOnly`.

Restoring image size, proportions

When **Image Only** has been selected as the image layout option, the context menu displayed when right-clicking the image contains two additional options under the menu item **Image View**:

Restore Initial Size and **Restore Initial Proportions**.

Modifying, replacing, and removing associated image files

Once an image has been associated with a model element, a menu item called **Associated Image** will be available in the context menu that is displayed when the element is selected in the browser. This item contains the following options:

- ♦ **Open Image** allows you to view the image in an external viewer. The application that is opened is determined by two properties. If the property `General::Model::ImageEditor` is set to the value `AssociatedApplication`, the default application for this type of file will be launched. However, if this property is set to `ExternalImageEditor`, the application launched will be the application specified with the property `General::Model::ExternalImageEditorCommand`. The value of this property should be the command-line for launching the relevant application.
- ♦ **Replace Image** displays a file browser, allowing you to select a new image file to associate with the model element, replacing the current associated image.

- ♦ **Remove Association** breaks the association between the model element and its associated image file.

Compatibility with previous image association mechanism

While the procedure described in this section is the current way to associate an image file with a model element, Rational Rhapsody still supports the old approach of associating a stereotype with a bitmap file. If you have defined such stereotypes, simply select **Enable Image View** in the Display Options window to display the image.

Controlled files and image association

If you add an image file as a controlled file beneath a model element, the image file is automatically associated with the model element.

Smart drag-and-drop

This section describes how to operate the Rational Rhapsody smart drag-and-drop feature. It provides step-by-step instructions for performing smart drag-and-drop operations.

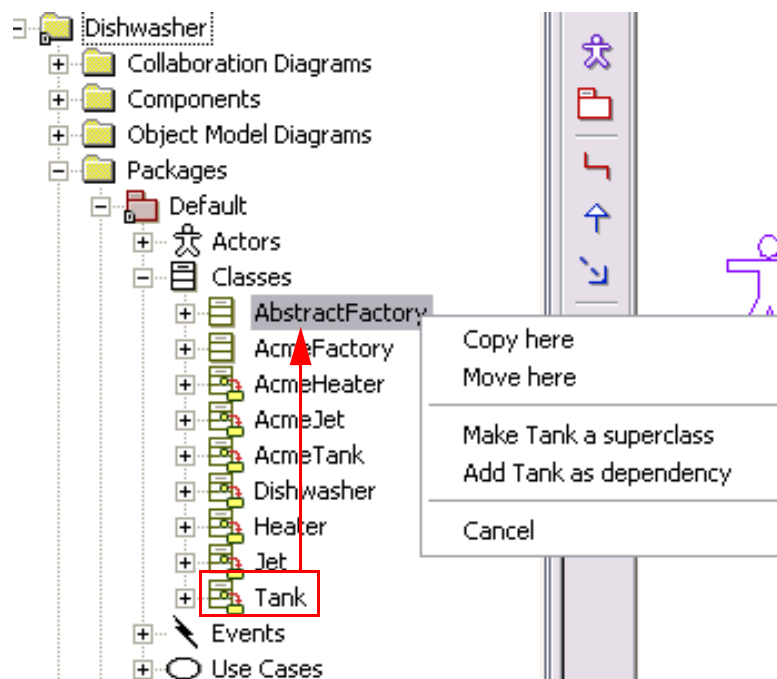
To drag-and-drop a class:

1. Expand your browser view so that all classes are viewable.
2. Click a class in the browser and while holding the mouse button down, drag the class onto another class; before releasing the mouse button press the **Shift** key.

Note

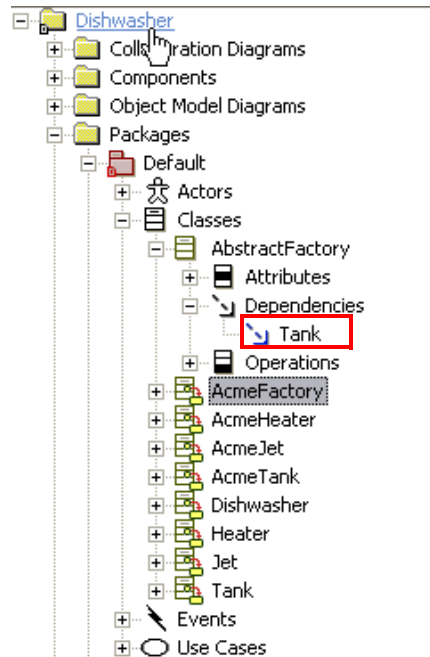
You must press and hold the **Shift** key before releasing the mouse button when dragging-and-dropping a class.

3. A context submenu displays. Select the wanted option. In the example, the Tank class is being dragged onto the AbstractFactory class.

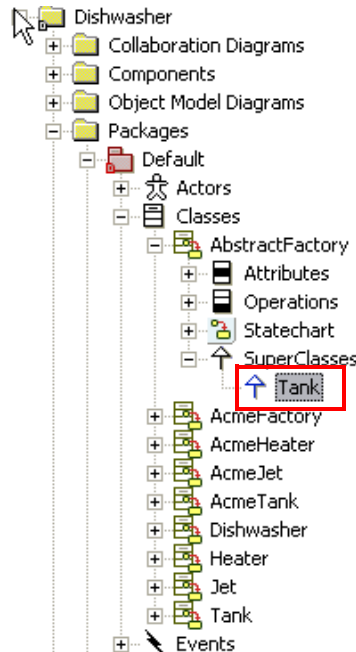


4. In this example the Tank class will be added as a dependency.

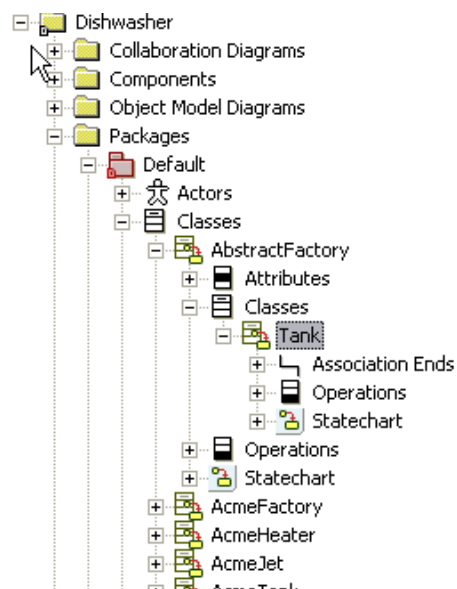
5. Once a selection is made from the submenu, the browser opens the feature added (in this case, the dependency).



6. If the class selection (Tank) were being made a *superclass*, it would appear a dependency added as a superclass.

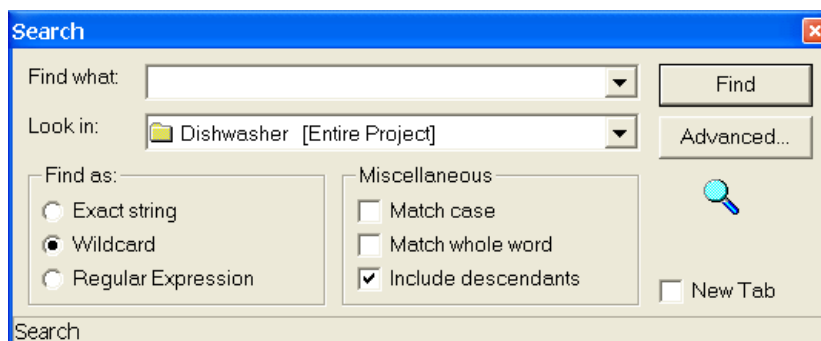


7. If the class selection (Tank) were being copied to another class (Abstract Factory), it would appear as shown in the following figure. Note that the class attributes for Tank remain intact.



Searching in the model

Engineers and developers can use the Rational Rhapsody Search and Replace facility for simple search operations. To perform quick searches with commonly used filters such as wildcards, case, and exact string, select **Edit > Search** to display this window.



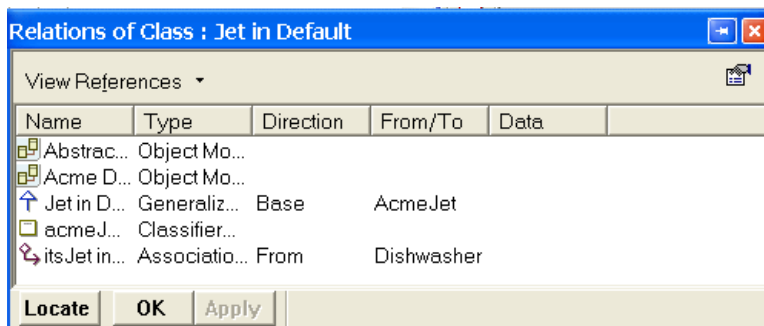
To display new search results in a separate tab of the Output window, select the **New Tab** check box and enter new search results and click **Find**.

You can also highlight an item in the browser and search only in that part of the project by selecting **Edit > Search In**. If you want to perform more complex searches, click **Advanced** in this window.

Finding element references

The References feature enables you to find out where the specified element is used in the model.

1. Highlight an element in the browser.
2. Select **Edit > References**. The Relations window opens, listing the locations of the element in the model.



If the element is not used anywhere in the model other than in the browser, Rational Rhapsody displays a message stating that the element is not referenced by any element.

References option returns only model elements that actually *reference* the specified element; it does not return references to the textual name of the element. To do a text-based search for the element name, use the search and replace functionality described in [Searching in the model](#).

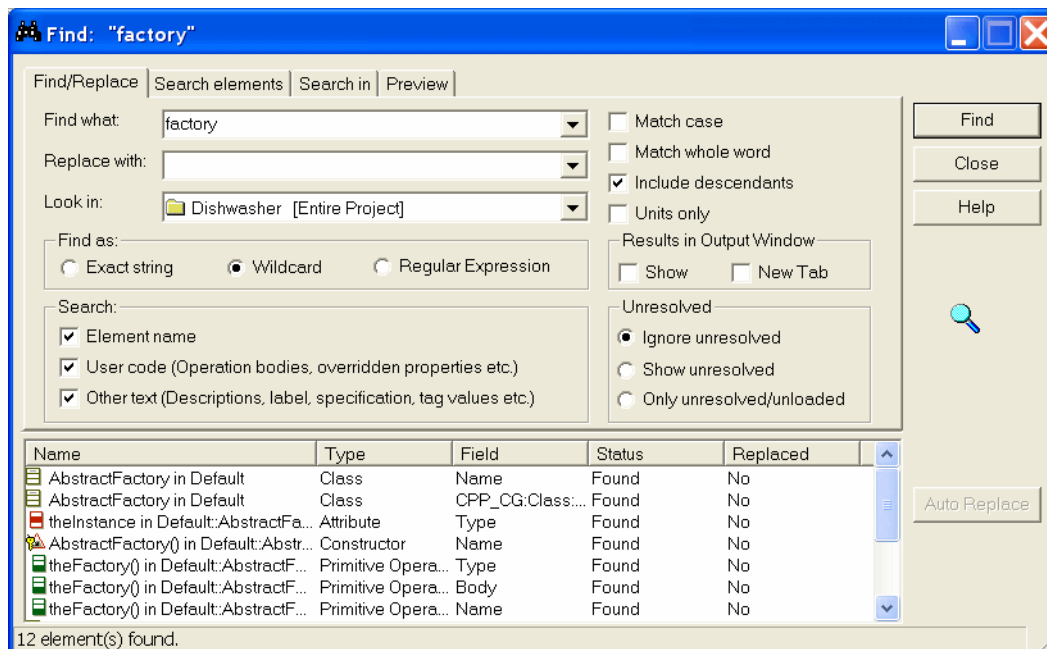
Note

You can also display the list of references by using the keyboard shortcut **CTRL+R**.

Advanced search and replace features

To manage large projects and expedite collaboration work, complex searches are often needed. For those types of searches, select **Edit > Advanced Search and Replace**. The advanced facility provides the following additional capabilities:

- ◆ Locate unresolved elements in a model
- ◆ Locate unloaded elements in a model
- ◆ Identify only the units in the model
- ◆ Search for both unresolved elements and unresolved units
- ◆ Perform simple operations on the search results
- ◆ Create a new tab in the Output window to display another set of search results



All search results display in the Output Window with the other tabbed information. Search and replace results display at the bottom of the Find window, as shown in the example. For more information about the uses of this facility, see [Advanced search and replace features](#).

Using the auto replace feature

To perform a search and replace operation with an automatic replace:

1. Choose **Edit > Advanced Search and Replace**.
2. Enter the **Find what** and **Replace with** text in those two fields.
3. Click **Auto Replace**. The changes display at the bottom of the window.

This automatic replacement also performs an automatic commit of the changes into the Rational Rhapsody model so that you do not need to apply the changes manually. However, it does not display any ripple effects of the change.

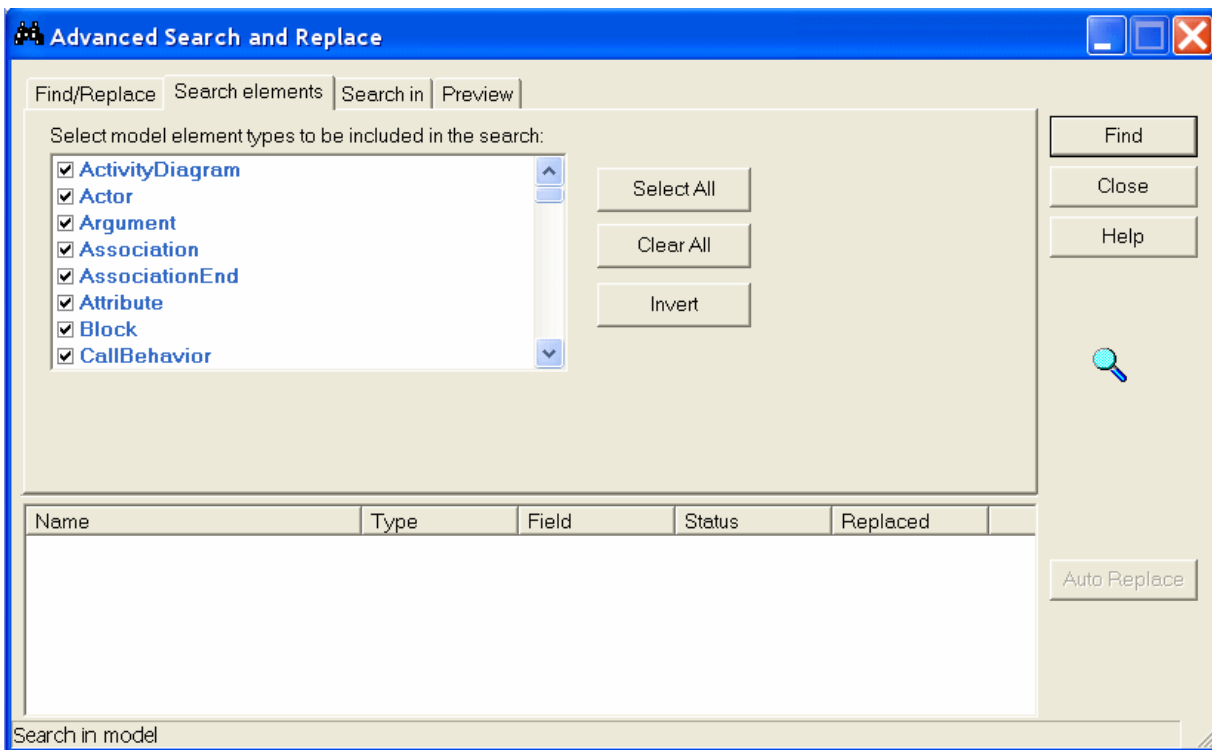
The Auto Replace feature cannot be used on the following items because they do not qualify to be replaced automatically:

- ◆ Read-only elements.
- ◆ A description that includes hyperlinks
- ◆ Stub or unresolved element names
- ◆ The name of an event reception, constructor, destructor, generalization, or hyperlink

Searching for elements

To search for specific elements in the model:

1. Choose **Edit > Advanced Search and Replace** to display the Advanced Search and Replace window.
2. Click the **Search elements** tab to display this window.



3. Select any or all of the element types listed in the scrolling area. Use the **Select All**, **Clear All**, or **Invert** to select element types quickly.
4. Click **Find**. The results display at the bottom of the window.
5. Highlight an item in the search results and right-click to perform either of these operations:
 - ♦ **References** displays all of the relations to the selected element.
 - ♦ **Delete from model**.
6. If you double-click an element in the list, the Features window opens so you can see and change the description and other characteristics for the element.

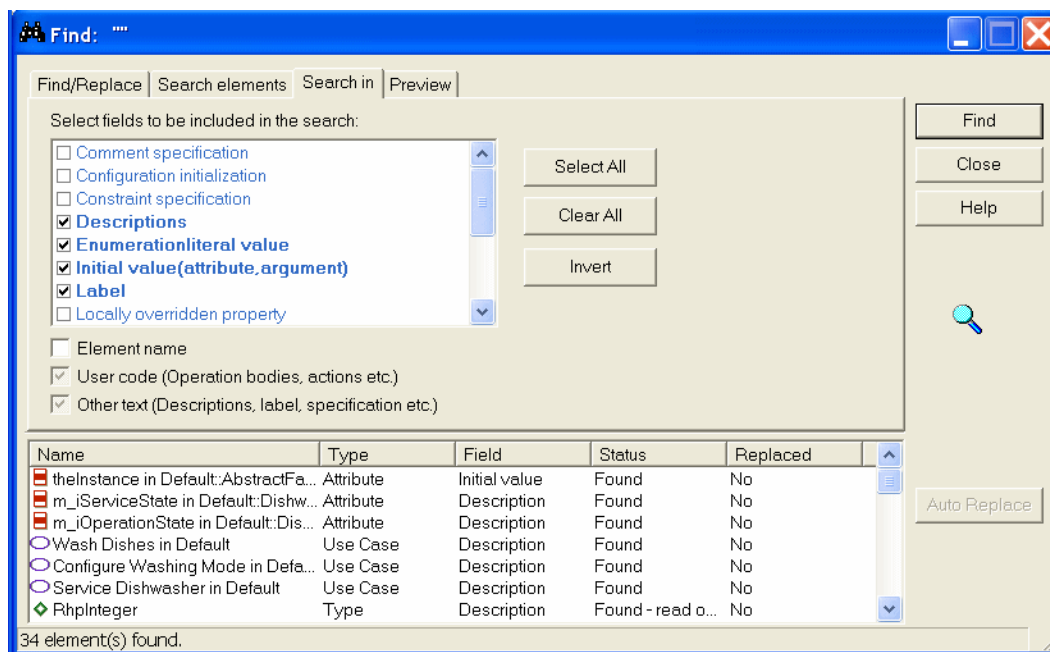
7. You can also highlight an item and perform an **Auto Replace**, as described in [Using the auto replace feature](#).

Searching in field types

To search in specific fields within the model:

1. Choose **Edit > Advanced Search and Replace** to display the Advanced Search and Replace window.
2. On the **Search in** tab, select the types of fields you want to find in the model by checking them in the list at the top of the window.
3. The **Element name**, **User code**, and **Other text** check boxes display the fields associated with them. For example, if only **User code** is checked, only the code-related fields (Actions, Configuration initialization, Initial value, Operation bodies, and Type declarations) are checked. Similarly, if only **Other text** is checked, only the text-related fields are checked. The possible values are as follows:
 - ♦ **Comment specification** scans all the comment specifications for the specified string
 - ♦ **Configuration initialization** scans all the configuration `init` code segments for the specified string
 - ♦ **Constraint specification** scans all the constraint specifications for the specified string (Chinese, Korean, Taiwanese, and Japanese are supported in this field and stored as RTF.)
 - ♦ **Descriptions** scans all the descriptions for the specified string
 - ♦ **Enumeration literal value** scans the literal value of the enumeration type for the specified search string.
 - ♦ **Initial value** scans the initial value fields for attributes and arguments for the specified search string
 - ♦ **Label** scans all the labels for the specified search string
 - ♦ **Locally overridden property** scans the locally overridden properties for the specified search string.
 - ♦ **Multiplicity** scans the Multiplicity field of associations for the specified search string.
 - ♦ **Name** scans all the element names for the specified search string
 - ♦ **Operation bodies** scans operation bodies, including operations defined under classes, actors, use cases, and packages for the specified search string
 - ♦ **Requirement specifications** scans all requirement specifications for the specified search string. (Chinese, Korean, Taiwanese, and Japanese are supported in this field and stored as RTF.)

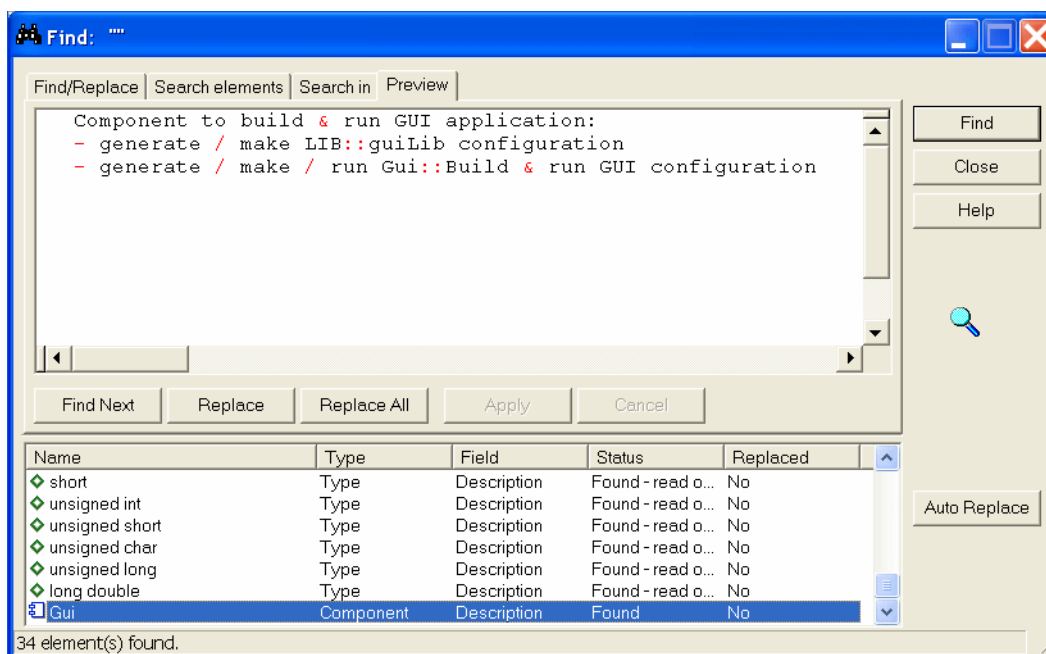
- ♦ **Tag Value** scans the value of the tags for the specified search string.
 - ♦ **Activity Flow Label** scans all activity flow code (action, guard, trigger) for the specified search string.
 - ♦ **Type declarations** scans all the user text in “on-the-fly” types for the specified search string.
4. You can also specify that the **Element name**, **User code**, and **Other text** be included or excluded from the search.
 5. Click **Find** to list the fields that meet your search criteria at the bottom of the window.



You can highlight an item in the list and replace it.

Previewing in the search and replace facility

After performing a search, you can highlight an item in the search results and click the **Preview** tab. This displays the occurrence of the highlighted item so that you can see and perhaps change the string.



Use the following buttons to make changes to the selected occurrence:

- ◆ **Find Next** highlights the next occurrence of the string in the same file.
- ◆ **Replace** replaces the highlighted occurrence of the original string with the replacement string specified on the **Find/Replace** tab.
- ◆ **Replace All** replaces all occurrences of the string.
- ◆ **Apply** applies the changes done in the **Preview** tab (editing or replacing text) to the Rational Rhapsody model.
- ◆ **Cancel** cancels the search and replace.
- ◆ **Auto Replace** automatically replaces all occurrences of the highlighted string with a replacement string.

Note

If you make a mistake, select **Edit > Undo** to undo the changes.

Controlled files

Rational Rhapsody allows you to create *controlled files* and then use their features.

The model in this section uses the “Cars” C++ sample (available in the <Rational Rhapsody installation path>\Samples\CppSamples\Cars) to demonstrate how to use controlled files.

- ♦ Controlled Files, such as project specifications files (for example, Microsoft Word, Microsoft Excel files) are typically added to a project for reference purposes and can be controlled through Rational Rhapsody.
- ♦ A controlled file can be a file of any type (.doc, .txt, .xls, and so on).
- ♦ Controlled files are added into the project from the Rational Rhapsody browser.
- ♦ Controlled files can be added to diagrams via drag-and-drop from the browser.
- ♦ Currently, only Tag and Dependency features can be added to a controlled file.
- ♦ By default all controlled files are opened by their Windows-default programs (for example, Microsoft Excel for .xls files).
- ♦ The programs associated with controlled files can be changed via the **Properties** tab in the controlled files window.

Creating a controlled file

Controlled files can be created using many methods. A controlled file can be created by browsing and selecting a file from the file system or by using a command. To select a method:

1. To select the way a controlled file is created, double-click a package in the Rational Rhapsody browser and select the **Properties** tab from the Features window.
2. With the **All** filter selected, expand the section **Model** and expand the **ControlledFile** subsection. The property **NewPolicy** controls the way a controlled file is created. The default selection is **SystemDefault**. The property can be set to **UseOpenCommand** or **UseRhapsodyCodeEditor**.
 - ♦ **SystemDefault:** This setting lets the program (as done in hyperlink, for example), open the file according to the MIME-type mapping.
 - ♦ **UseOpenCommand:** This setting means that double-clicking on the controlled file runs the open command. The open command is set in the **OpenCommand** property (**Model/ControlledFile/OpenCommand**). This is a string property that expands the keyword **\$fileName** to the full path of the Controlled File. In the event of property inconsistency, where the program is set to run the **UseOpenCommand** but the command contains no setting, the **SystemDefault** policy will be used.
 - ♦ **UseRhapsodyCodeEditor:** This setting allows the file to be opened within the internal Rational Rhapsody code editor.

If you want to open the controlled file after it is created, set the value of the property **OpenFileAfterCreation** to **yes**. The file will open the file immediately after it is created.

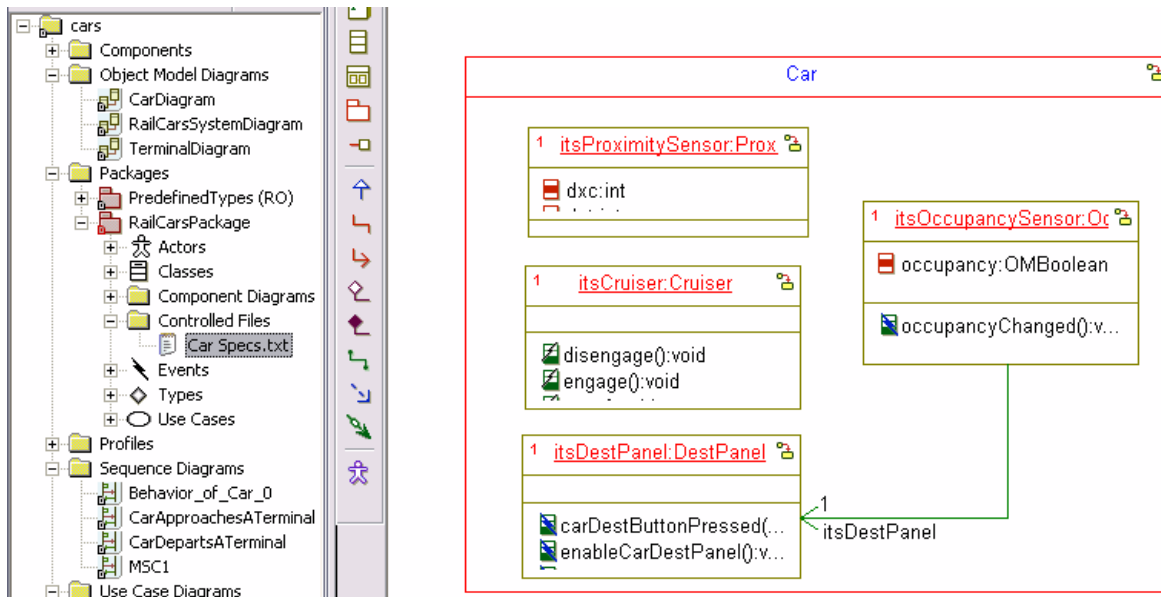
Browsing to a controlled file

This section describes how to create a controlled file by browsing to it.

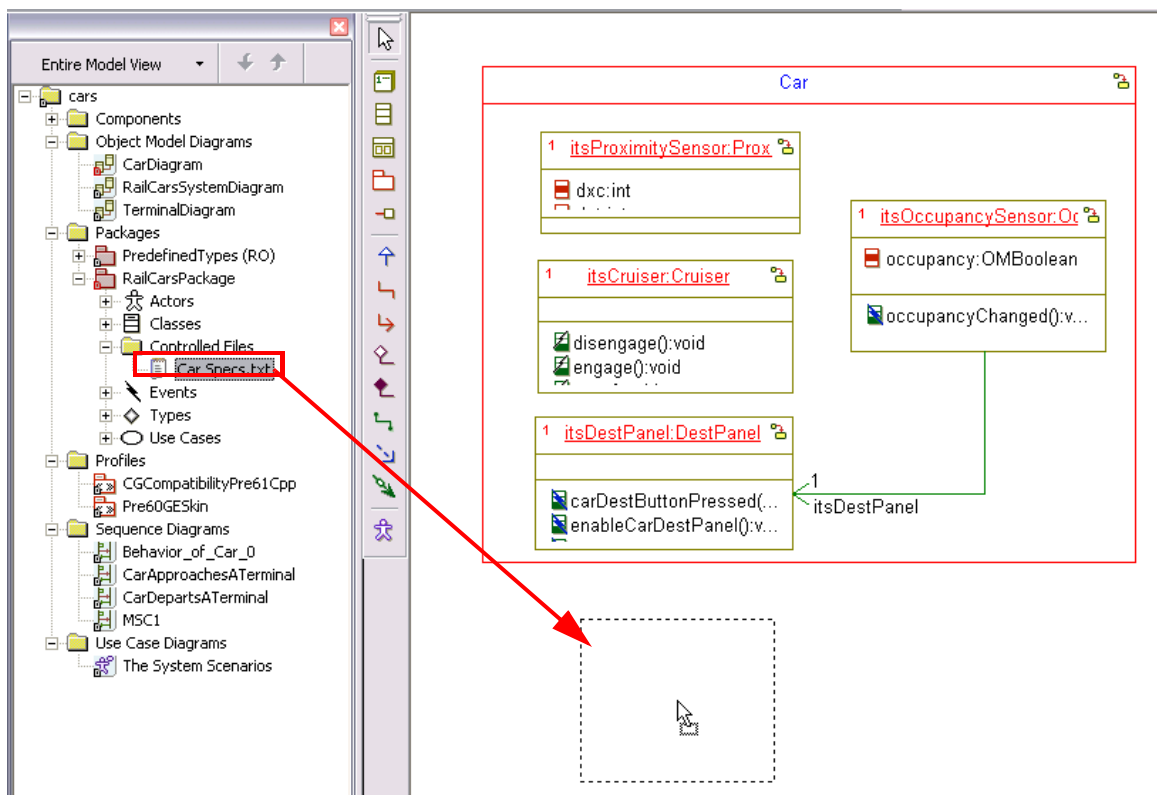
1. To create a controlled file, right-click a non-read-only element, such as a package, in the browser and select **Add New > Annotations > Controlled File**. The Select Controlled File open window opens.

Note: **Add New > Annotations** is the default menu command structure in Rational Rhapsody. It can be changed by users. This topic assumes that all defaults are in place.

2. Navigate to the wanted file and click the **Open** button. Unless the controlled file being selected is located in the right location under the `_rpy` directory (cars_rpy in the example), a message displays asking you for permission to copy the file. If you do not want the file copied into the `_rpy` directory, you must place the original into the `_rpy` directory before selecting it from the Rational Rhapsody browser.
3. A window displays asking you for permission to copy the file into the `_rpy` directory. Click **OK**.
4. The controlled file now displays in the Rational Rhapsody browser.



- Controlled files can be added to the graphic interface (right window pane) via drag-and-drop from the browser.



Controlled file features

Controlled files can be any standard *Windows* type, such as .doc, .txt, and .xls. When those files are opened in Rational Rhapsody, they are displayed in their Windows-default programs, such as for Microsoft Excel for .xls files. In addition, the programs associated with controlled files can be changed via the **Properties** tab in the controlled files window.

Adding dependencies to controlled files

Dependencies can be added to controlled files to display what object depends on the controlled file or what object the controlled file is dependent upon.

To add a dependency to a controlled file (through the Rational Rhapsody browser):

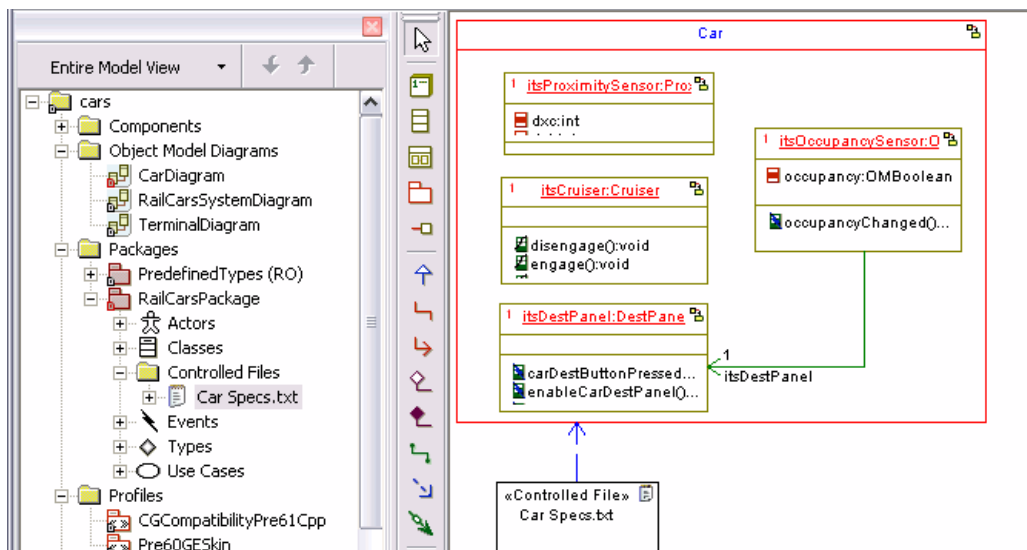
1. To add a new dependency, right-click the controlled file and select **Add New > Relations > Dependency**.

Note: **Add New > Relations** is the default menu command structure in Rational Rhapsody. It can be changed by users. This topic assumes that all defaults are in place.

2. Make a selection in the window that displays and click **OK**.
3. Type the name of the dependency, or leave the default name.
4. Double-click the dependency in the browser to modify it.

To add a dependency to a controlled file (via Graphic Pane):

1. Drag the controlled file from the browser into the graphic pane, if you have not already done so.
2. Using the dependency icon , draw a dependency line from the controlled file to the wanted object.



Adding tags to controlled files

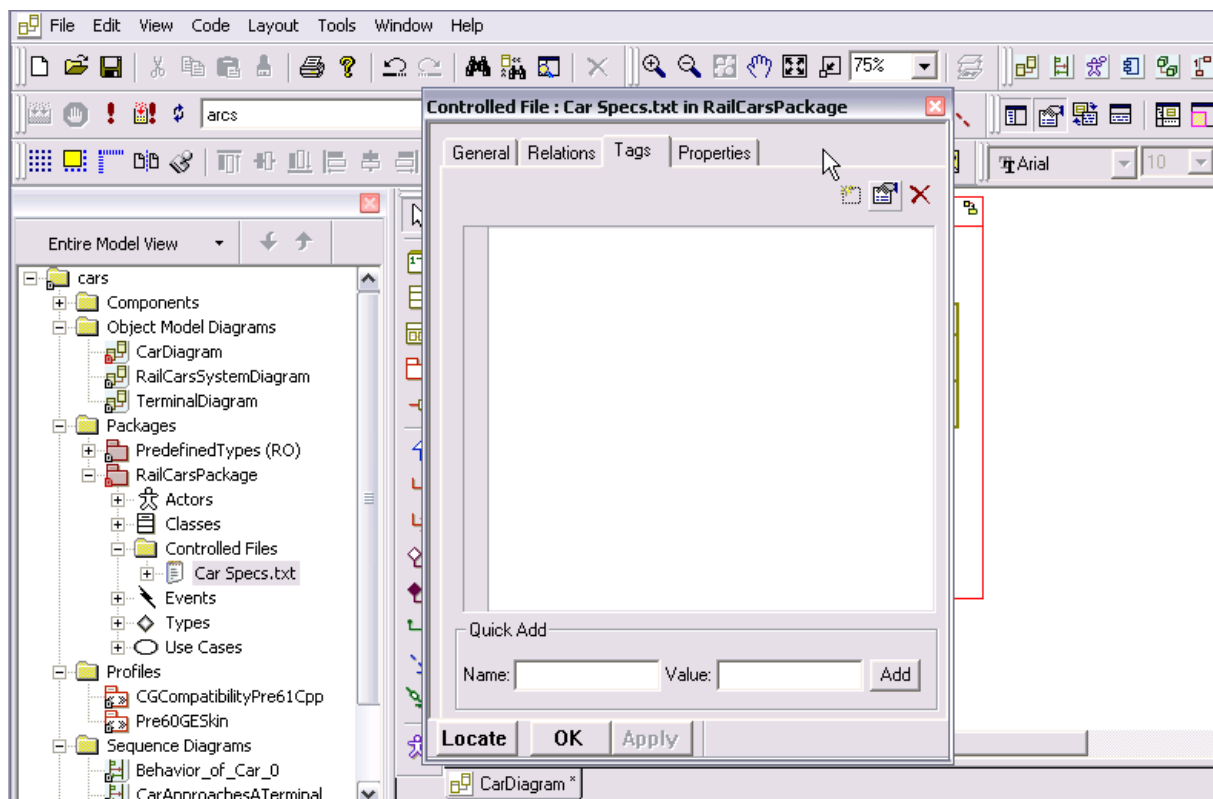
Tags can be added to controlled files for purposes such as identification. For instance, tagging all controlled files that are related to the specification for a product helps differentiate those controlled files from other controlled files that might be embedded in the project.

To add a tag to a controlled file (via the browser):

1. Right-click the controlled file and select **Add New >Tag**.
2. Type a name for the tag or leave the default name.
3. Double-click the tag in the browser to modify it.

To add a tag to a controlled file (via the Features window):

1. Right-click the controlled files in the browser and select **Features**.
2. The Features window displays. Click the **Tags** tab and enter the name of the tag in the “Quick Add” section at the bottom of the window. Click the **Add** button.



3. Click **OK**.

4. Expand the Controlled File by clicking on the “+” next to it in the browser.
5. The Tags listing displays. Click the “+” next to the Tags listing to see the new Tag you just created.

Configuration management

Configuration management options are as follows:

Configuration Management Feature	Feature Description
Add to Archive	This feature is used when a you want to add a new file into the configuration management server for the first time. After that, other users can check in/out, lock/unlock the file.
Check In	This feature checks the file into the configuration management server so that other users can modify the file. You still have read-only access to the file, but you cannot modify it.
Check Out	This feature checks the file out of the configuration management server so that other users cannot modify the file. Other users can open the file for read-only purposes.
Lock	This feature locks the file so that other users cannot modify it in the future. Only you can unlock the file.
Unlock	This feature unlocks the files so that other users can modify the file in the future. Only you can unlock the file.

Troubleshooting controlled files

Following is a list of the most commonly encountered problems with their solutions:

- ♦ **Question:** “I checked out a controlled file. But I am not able to change its description, properties, and so on.”

Answer: The metadata for the controlled file is stored in its parent element. So you must check out the parent element in order to change any associated information, such as the description, properties, and so on.

For instance, in the example provided in [Creating a controlled file](#), right-clicking on the parent element RailCarsPackage and choosing Configuration Management ▶ Check out, allows the controlled file attributes to be modified.

- ♦ **Question:** “I know controlled files are copied to a directory under _rpy during the creation of the controlled file, but where exactly is the controlled file copied?”

Answer: Controlled Files are copied to the same location where its parent is located on the disk. For instance, suppose you have a package called “package_1” and the repository file for it is stored like this: MyProject_rpy\package_1\package_1.sbs.

Now any controlled file you add to the package “package_1” is going to be stored in the folder: MyProject_rpy\package_1\

- ♦ **Question:** “When I open a Controlled File Features window and try to rename the Controlled File, I receive the ‘File not found’ error.”

Answer: You have the ability to select different files from an opened Features window. So when you type a different name, the system “thinks” that you are actually trying to point to a different file and returns an error message.

You can rename a controlled file from the browser by clicking two separate times on it (double-clicking on the file will open it).

- ♦ **Question:** “Is it possible to add a Controlled File to the model without copying the file to the _rpy folder?”

Answer: No, but you can add a hyperlink to the wanted file instead. However, this will result in the loss of all CM (Configuration Management) capability which includes the following operations:

- Add to archive
- Check in
- Check out
- Lock
- Unlock

- ♦ **Issue:** Configuration management (CM) operations on Controlled files fail on some occasions.
- ♦ **Details:** If you are using CM in command mode, make sure the file is located under the _rpy folder. In order to perform Configuration Management operations in command mode from Rational Rhapsody, the file (other than the project itself) has to be under the _rpy directory. There is no current fix for this issue.

Workaround: Create a new package under the project. By default, a package is stored in separate file: do not modify that setting. Now add the problematic controlled files to this package. Now you will be able to perform CM operations.

- ♦ **Issue:** Adding an existing element with a Controlled File to the Model and choosing a different name, instead of replacing the file.
- ♦ **Details:** If you perform an Add To Model of an already existing element, Rational Rhapsody gives you the option of replacing the element or adding it using a different name. If you choose to use a different name, Rational Rhapsody adds the element using the new name.

If both the element that already existed in the model and the element that was just added have controlled files by the same name, more than one controlled file of the same name exists in the model. But there might be only one instance of the file in the directory, which means the two controlled files in the model are actually pointing to the same file.

Controlled file limitations

General limitations

- ♦ **Connecting to Archive Error:**

If your configuration management tool is Rational ClearCase, this Connecting to Archive limitation applies to you.

If you are trying to “Connect to Archive” using the Configuration Items tool (File/Configuration Items), and your Rational Rhapsody project contains a controlled file, the following error results:

Cannot perform Connect to Archive.

Reason: Rational Rhapsody could not add “Project_rpy” to Source Control as the project contains at least one Controlled File. To proceed, add this directory (and its subdirectories) to Source Control using Rational ClearCase. Note that, you do not need to Connect to Archive again.

Use Rational ClearCase to add the directory to the appropriate source control. You now do not have to perform a “Connect to Archive.”

- ♦ **Deleting an Element with a Controlled Files Deletes the Controlled File From the Model Only.**

If you delete a controlled file from a Rational Rhapsody Model, you are given the option to either delete the file completely from the hard disk drive or leave the file intact and simply delete the controlled file from the model. If you delete any other element that includes a controlled file (as a child), the controlled file is left intact on the hard disk; the option to delete the controlled file completely from the hard disk drive is not offered.

- ♦ **Moving a Controlled File - Browser vs. Graphic Editor**

If you have two packages stored in different directories (P1 and P2 for this example), and you add a controlled file to the P1 directory, the physical controlled file on the hard disk is stored in P1's directory (the same directory where P1.sbs file is located).

From the Rational Rhapsody browser, drag the controlled file from P1 and drop it in the P2 package. Now examine the contents of the P1 and P2 directories on the hard disk. Notice that the file has moved from the P1 directory to the P2 directory.

If you repeat the above exercise in the Rational Rhapsody Graphic Editor, the physical file is *copied* from the old directory to the new directory, instead of being completely *moved* from the old directory to the new directory.

- ♦ **Configuration Management Synchronize window Does Not Show Controlled Files**

Configuration Management's Synchronize window shows items that are modified by other users but not yet loaded into your model. This window does not hold true for controlled files. The synchronize window does not display Controlled Files when there are newly-updated controlled files in the Configuration Management system.

- ♦ **Controlled Files Version Number Not Shown In the Configuration Management Window**

The Configuration Management window shows the version number for the items in its listing (only in command mode). But for a controlled file, it does not show the version number.

- ♦ **Foreign Language Controlled File Names are Not Supported**

If your file name has non-English characters, you cannot add it to your project as a controlled file.

- ♦ **CM Archive is Not Affected When Controlled Files are Moved/Renamed/Deleted**

When you move, rename or delete a Controlled File, its respective Configuration Management item is not updated accordingly.

DiffMerge limitations

- ◆ **Diffmerge Does Not Compare the Contents of Two Controlled Files**

Diffmerge does not identify/indicate any content differences between two controlled files. Diffmerge does not examine the contents of controlled files as they exist on the hard disk drive. However, Diffmerge can show differences between the metadata information of two controlled files in the model.

Ex. Given two packages (Pkg1 and Pkg2) that each contain the controlled file readme.txt, if the file contents of the Pkg1 readme.txt are different from the contents of the Pkg2 readme.txt then a Diffmerge performed between Pkg1 and Pkg2 will not identify that the readme.txt files are different. However, if the Pkg1 readme.txt and Pkg2 readme.txt have different metadata such as properties or description then those differences are shown in the Diffmerge.

- ◆ **'Save merge as...' or 'merge back to rhapsody' Does Not Save Any Controlled Files**

When using Diffmerge to compare two model units that contain controlled files, performing the “Save merge as...” or “merge back to rhapsody” functions does not save the controlled files on the hard disk drive.

Ex. Using the same example as above, if you compare Pkg1 with Pkg2 and save the merge as “Pkg_Merged,” the physical file readme.txt for Pkg_Merged will not be saved on the physical hard disk drive. If you were to navigate to the file under the appropriate directory (c:\diffmerge for example), you would notice the readme.txt file does not appear in that directory.

Search and replace limitations

Note

After performing a search using regular expressions, look inside the **Preview** tab in the Search and Replace window. The **Find Next**, **Replace**, and **Replace All** functionalities might not work for one or more selected search results.

This search and replace limitation is not limited to controlled files.

Print Rational Rhapsody diagrams

The File menu includes the following print options:

- ♦ **Print Diagrams** displays all the diagrams used in the model so you can easily select which ones to print (see [Selecting which diagrams to print](#))
- ♦ **Print** displays the standard Print window, which enables you to print the current diagram on the specified printer
- ♦ **Print Preview** displays a preview of how the diagram will look when it is printed
- ♦ **Printer Setup** displays the standard Print Setup window, which enables you to change the printer settings, such as landscape instead of portrait mode
- ♦ **Diagram Print Settings** enables you to specify diagram-specific print options (see [Diagram print settings](#))

Selecting which diagrams to print

Rational Rhapsody enables you to print multiple diagrams.

1. Choose **File > Print Diagrams**. The Print Diagrams window opens, listing the diagrams and statecharts in the current project. The window has the following features:
 - ♦ The list box on the left shows all of the diagrams in the project.
 - ♦ The list box on the right shows the names and types (for example, ObjectModelDiagram) of the diagrams selected for printing.
 - ♦ The buttons in the middle let you select and deselect diagrams for printing.
 - ♦ The Up/Down arrows on the far right control the order in which the diagrams are printed.
 - ♦ The buttons along the bottom edge of the window access the Print window, or cancel printing.
2. To expand the list of diagrams in the project, click the + sign to the left of one of the diagram type names. The tree expands to show all the diagrams of that type. Note that statecharts are listed first by package, then by class.
3. Do one of the following actions:
 - a. To add a diagram to the print list, select it in the list on the left and click **Add**.
 - b. To add all the diagrams in the project to the print list, click **Add All**.
 - c. To remove a diagram from the print list, select it in the list on the right and click **Remove**. Rational Rhapsody prints the diagrams listed in the right from top to bottom.

- d. To change the print order, select a diagram from the print list and use the up or down arrows.
 - 4. Click the appropriate button:
 - a. **Print** dismisses the Print Diagrams window and opens the standard Print window for your operating system. In this window, you set attributes such as page size (for example, 11" x 17"), the printer to use, whether to print to a file, whether to use grayscale, and initiate the printing operation.
 - b. **Cancel** cancels the print operation and dismisses the window.

Diagram print settings

File > Diagram Print Settings lets you specify diagram-specific print settings that can be retained between Rational Rhapsody sessions. This option is particularly useful for complex and multipage diagrams.

The window contains the following fields:

- ◆ **Orientation** specifies the page orientation
- ◆ **Scaling** specifies the scale percentage, or whether to shrink the diagram so it fits on one page.
- ◆ **Options** specifies whether to:
 - Print the background color.
 - Include a header or footer on the printout.
 - Retain your diagram settings across Rational Rhapsody sessions.

By default, the header includes the name of the diagram, and the footer contains the page number. To specify a new header or footer, uncheck the appropriate check box and type the new value in the text box.

For example, to include the name of your company in the diagram header:

1. Clear the **Add diagram name in header** check box.
2. In the text box, type the name of your company.
3. Click **OK**.
4. Use one of the Print options to print out the diagram.

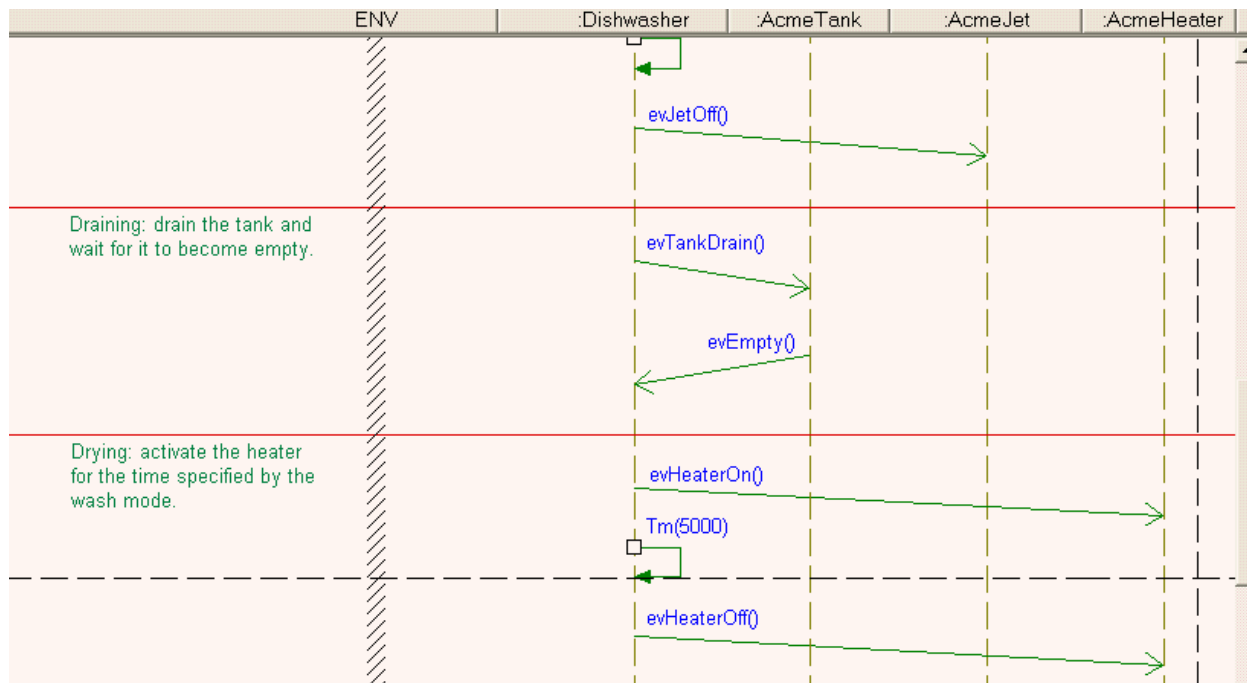
Using page breaks

Rational Rhapsody can break a large diagram across several pages or scale it down so it fits on a single page, depending on your preference.

To scale the diagram so it fits on a single page, right-click in the diagram and select **Printing > Fit on One Page**. To break the diagram across pages, simply disable this option.

To view the page breaks in a diagram, do one of the following actions:

- ◆ Click the **Toggle Page Breaks** tool to toggle the display of page breaks in your diagram.
- ◆ Select **Layout > View Page Breaks**.
- ◆ Right-click in the diagram and select **Printing > View Page Breaks**. The dashed lines that represent the page boundaries are displayed.



Exporting Rational Rhapsody diagrams

To export any Rational Rhapsody diagram:

1. Display the diagram.
2. Right-click inside the diagram, but not on a specific item in the diagram.
3. From the displayed menu, select **Export Diagram Image**.
4. In the Save As window, select a directory for the saved image in the **Save in** directory selection field.
5. In the **File name** field, type the name you want to use for the exported image file.
6. In the **Save as type** field, select one of these image types:
7. Click **Save** to complete the export process.

Annotations for diagrams

You can add different types of annotations to specify additional information about a model element. The annotation can add semantics (like a constraint or requirement) or can simply be informative, like a documentation note or comment.

Note

None of the annotation types generate code; they are used to improve the readability and comprehension of your model.

- ♦ **Constraint** for a condition or restriction expressed in text. It is generally a Boolean expression that must be true for an associated model element for the model to be considered well-formed. A constraint is an assertion rather than an executable mechanism. It indicates a restriction that must be enforced by the correct design of a system.

Constraints are part of the model and are, therefore, displayed in the browser.

- ♦ **Comment** for a textual annotation that does not add semantics, but contains information that might be useful to the reader and is displayed in the browser.
- ♦ **Note** for a textual annotation that does not add semantics, but contains information that might be useful to the reader. Notes are not displayed in the browser.
- ♦ **Requirement** for a textual annotation that describes the intent of the element. Note that a requirement modeled inside Rational Rhapsody does not replace the usage of a dedicated requirement traceability tool, such as DOORS. Instead, a modeled requirement complements the usage of such a tool because the hierarchical modeling of requirements enables you to easily correlate each requirement to the element that addresses it.

Requirements are part of the model and are therefore displayed in the browser.

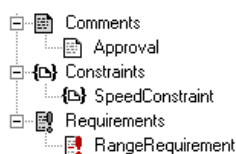
Creating annotations

To add an annotation to a diagram:

1. Click the appropriate icon in the Diagram Tools for the type of annotation you want to create.
2. Single-click or click-and-drag in the diagram to place the annotation at the intended location.
3. Type the note text or expression.
4. Press **Ctrl+Enter** to terminate editing.

The same annotation can apply to more than one model element and multiple annotations can apply to the same model element. You can use “ownership” instead of an anchor. For example, if a requirement is owned by a class, it is a requirement of the class.

The new annotation is displayed in the diagram, and in the browser if it is a modeled annotation. Note that documentation notes and text are *graphical annotations* and exist only in the diagram; all the other Rational Rhapsody annotations are *modeled annotations*, and are part of the model itself. Because modeled annotations are part of the model, they can be viewed in the browser. In addition, you can move modeled annotations to new owners using the browser, and can drag-and-drop them from the browser into diagrams. Modeled annotations are displayed under separate categories in the browser by type.



Alternatively, you can create modeled annotations in the browser, as follows:

1. Right-click the element that needs the annotation.
2. Select **Add New > Annotations > Constraint, Comment, or Controlled File**. The new annotation is added under the selected element.

An annotation created using the browser does not anchor the annotation to the specified model element. However, you can organize annotations so that anchoring is implied.

Creating dependencies between annotations

You can show dependencies between annotations using the **Dependency** icon in the Diagram Tools.

To create dependencies between annotations:

1. Click the **Dependency** icon in the **Diagram Tools**.
2. Click the edge of the dependent annotation.
3. Click the edge of the annotation on which the first annotation depends.

Creating hierarchical requirements

You can create hierarchical requirements by having one requirement own all the subrequirements; the owning requirement is called the “sum requirement.”

1. In the browser, right-click the component that will own the requirement and select **Add New > Requirement**.
2. Name the new requirement. This is the sum requirement.
3. Right-click the sum requirement and select **Add New > Requirement**.
4. Name the subrequirement.
5. Continue creating subrequirements as needed.
6. Create dependencies between the requirements as needed.

The following figure shows a hierarchical requirement.



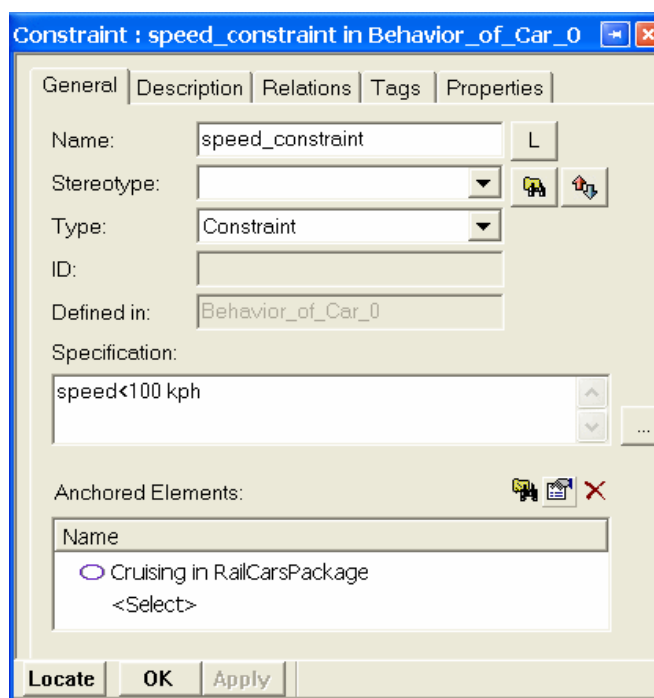
Editing annotation text

To edit the text of an existing annotation, do one of the following actions:

- ♦ Double-click the annotation.
- ♦ Right-click the annotation and then select **Features**.

Defining the features of an annotation

The Features window enables you to change the features of an annotation, including its name type. The following figure shows the Features window for an annotation (in this case, a constraint).



A constraint contains the following fields:

- ♦ **Name** specifies the name of the constraint. To enter a detailed description of the constraint, click the **Description** tab.
- ♦ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ **Stereotype** specifies the stereotype of the constraint, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).

- To select from a list of current stereotypes in the project, click the folder with binoculars  button.
- To sort the order of the stereotypes, click the up and down arrows  button.

Note: The COM stereotypes are constructive; that is, they affect code generation.

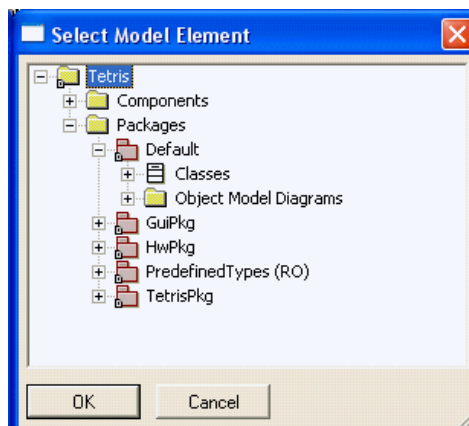
- ♦ **Type** specifies the annotation type. For example, you could use this field to easily change a constraint to a requirement or comment.
- ♦ **Defined in** (read-only field) specifies the owning class of the annotation.
- ♦ **Specification** specifies the note text or constraint expression.
- ♦ **Anchored Elements** lists all the elements that are anchored to this annotation.



To view the features for an anchored element, select the element in the list, then click the Invoke Feature Dialog button.



To anchor additional elements to this annotation, click the **Select** button or the <Select> line in the list to select the elements from the hierarchical display.



Converting notes to comments

Documentation notes “live” only within diagrams. They are not displayed in the browser.

To convert a note to a comment (to be displayed in the browser), in the diagram, right-click the note and the select **Convert to Comment**.

Anchoring annotations

You can anchor a constraint, comment, requirement, or note to one or more graphical objects that represent modeling concepts (classes, use cases, and so on). You can also anchor annotations to edge elements.

Although it is possible to anchor a requirement to an element, it is better to model it as a dependency. Anchors are shown as dashed lines.

To anchor an annotation to a model element:

1. Click the **Anchor** icon in the Diagram Tools.
2. Click an edge of the annotation.
3. Move the cursor to the edge of the model element to which you want to anchor the annotation, then click to confirm.

To change the line style of an anchor, right-click the anchor line and select the appropriate option from the **Line Shape** menu.

Note the following information:

- ♦ There should be only one anchor between a specific annotation and a given model element.
- ♦ An annotation can be anchored to more than one element.

Finding constraint references

To find which annotations are anchored to a particular class:

1. In the browser, select the class you want to query.
2. Select **References** from the menu. The References window lists all the references for the specified class. Anchored annotations are labeled “Anchor” in the Type column.

References of Class AirCondition				
Name	Type	Direction	From/To	Data
BatteryInfo	Flow	In	Battery	PowerStatus
BatteryInformation	Flow	In	Battery	PowerStatus
CommentInfo.anchor_0	Anchor	From	CommentInfo	
Flow Analysis	Object Model Dia...			
Overview	Object Model Dia...			
actualTemperatureInfo	Flow	In	TempSensor	Temperature
adjustFanSpeed	Flow	Out	FanCtrl	SetFanSpeed
adjustSteppersPosition	Flow	Out	SteppersCtrl	SetSteppers
buttonsNotifications	Flow	In	Keypad	PressedButtons
controlAC	Flow	Out	ACCtrl	ACOnOff,ACVol
keypadLights	Flow	Out	Keypad	SetLights
notifyCustomDisplay	Flow	Out	Display	DisplayInfo,evSt
queryACState	Flow	In	ACCtrl	ACStatus
queryFanSpeed	Flow	In	FanCtrl	FanSpeed
querySteppersPosition	Flow	In	SteppersCtrl	StepperState

Deleting an anchor

To delete an anchor:

1. Select the anchor.
2. Click the **Delete** tool in the main toolbar.

Changing the display options for annotations

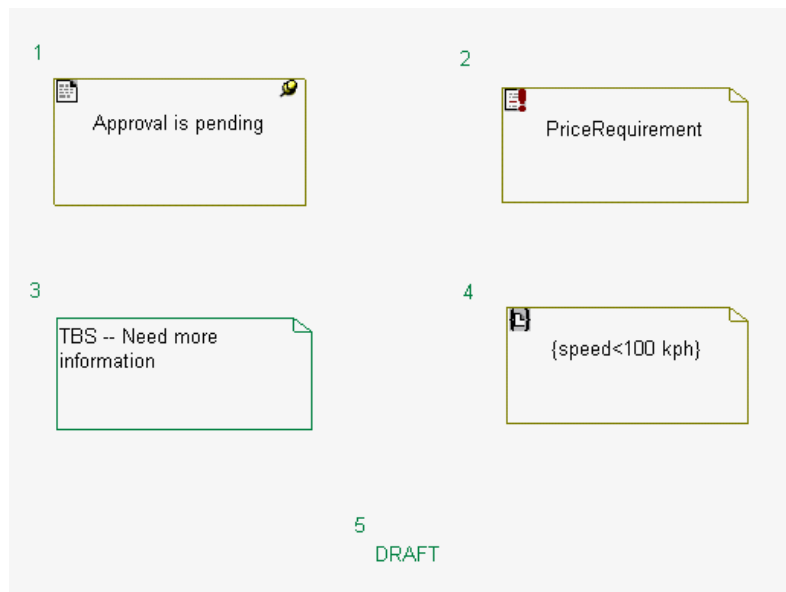
To change the display of annotations, right-click the note in the diagram and select **Display Options**.

The Display Options window for the selected annotation type opens, with only the relevant fields displayed. All of the fields are as follows:

- ♦ **Show** specifies which information to display for the annotation:
 - **Name** displays the name of the annotation
 - **Label** displays the label for the annotation
 - **Specification** displays the contents of the **Specification** field of the Features window for the annotation
 - **Description** displays the contents of the **Description** field of the Features window for the annotation
- ♦ **Form** specifies the graphical form of the annotation:
 - **Plain** shows the annotation without a border, as free-floating text
 - **Note Symbol** shows the annotation within a frame, with a symbol in the upper, left-hand corner to denote the annotation type
 - **Pushpin** displays the annotation with a rectangular border and a pushpin icon

The following figure shows some of the different display options. The annotation types and display options are as follows:

- ◆ Comment, displayed with its specification label using the pushpin style
- ◆ Requirement, displayed using its label
- ◆ Documentation note
- ◆ Constraint
- ◆ Text note



Deleting an annotation

To delete an annotation, on the browser, right-click the annotation to be deleted and select **Delete from Model**.

To delete an annotation from a diagram, right-click the note and do one of the following actions:

- ◆ Select the **Delete** icon from the toolbar.
- ◆ Select **Delete from Model** from the menu.

Using annotations with other tools

The following table lists the effect of annotations on both Rational Rhapsody and third-party tools.

Tool	Description
COM API	Annotations are supported by the COM API via the following interfaces: • IRPAnnotation • IRPComment • IRPConstraint • IRPRequirement
Complete Relation	When you select Layout > Complete Relations , anchors are part of the information added to the diagram.
DiffMerge	Annotations are included in difference and merge operations.
Populate Diagram	Annotations and anchors are not supported.
References	If you use the References functionality for a modeled annotation, the tool lists the diagrams in which the specified annotation displays. When you select a diagram from the returned list, the annotation is highlighted in the diagram. If you use the References functionality for an element, the tool includes any annotations that are anchored to the specified element For more information on this functionality, see Searching in the model .
Report on model	Modeled annotations are listed by type under the owning package. Graphical annotations are not included in the list.
Rational Rose	Notes in Rational Rose (which are not displayed in the browser) are imported as notes in Rational RhapsodyAnnotations.

Search in model	When you search in a model, the search includes modeled annotations, including the Body section. When selected, the annotation is highlighted in the browser. For more information on this functionality, see Searching in the model .

Annotation limitations

Note the following limitations for annotations:

- ◆ You cannot anchor an annotation to an element that is represented by a line (such as a message or activity flow).
- ◆ You cannot drag-and-drop an anchor from the browser to a diagram.

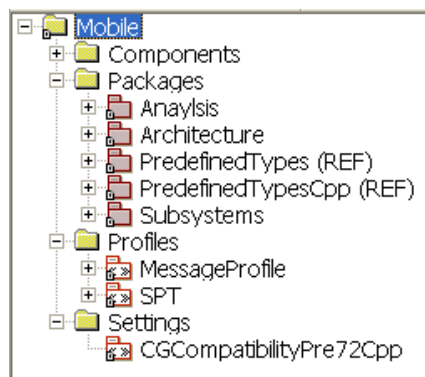
Profiles

A *profile* is a special kind of package that is distinguished from other packages in the browser. You specify a profile at the top level of the model. Therefore, a profile is owned by the project and affects the entire model. By default, all profiles apply to all packages available in the workspace, so their tags and stereotypes are available everywhere. A profile is very similar to any other package; however, profiles cannot be nested.

A profile “hosts” domain-specific tags and stereotypes. *Tags* enable you to add information to certain kinds of elements to reflect characteristics of the specific domain or platform for the modeled system. Tags are easily viewable in their own category in the browser and in the **Tags** tab of the Features window for an element.

You can apply tags to certain cases by associating the tag definition to stereotypes in a profile or a package. In this case, the tags are visible only for those model elements that have the specified stereotype. In addition, you can apply tags to individual elements.

The browser displays all of the profiles used in a project together, as shown in this example:



This example lists two profiles. **MessageProfile** is a user-defined (customized) profile. For more information about these profiles, see [Creating a customized profile](#). **SPT** was automatically added to the project because the developer specifies the Scheduling, Performance, and Time method to add timing analysis data. For more information, see [Types of profiles](#).

Creating a project without a profile

You are not required to use profiles in your Rational Rhapsody project. When creating a new project, you can create a *Default* project, containing all of the basic UML features:

1. Create the new project by either selecting **File > New**, or click the **New project** icon in the main toolbar.
2. Replace the default project name (Project) with <your project name> in the **Project name** field. Enter a new directory name in the **In folder** field or Browse to find an existing directory.
3. Select Default for the **Project Type** and **Project Settings**.
4. Click **OK**.

The basic structure for your new project is displayed in the browser with no Profiles folder.

If you have a project that was created in an older version of Rational Rhapsody and you begin to work on it in a newer version, Rational Rhapsody automatically inserts the required *settings* to manage any compatibility issues.

Backward compatibility profiles

You might note any of the following backward compatibility profiles automatically loaded into your project:

- ♦ **Pre60GESkin** uses the colors and fonts from versions before Rational Rhapsody 6.0.
- ♦ **CGCompatibilityPre61C** makes the code generation backwards compatible with pre-6.1 Rational Rhapsody in C models.
- ♦ **CGCompatibilityPre61Cpp** makes the code generation backwards compatible with pre-6.1 Rational Rhapsody in C++ models.
- ♦ **CGCompatibilityPre61Java** makes the code generation backwards compatible with pre-6.1 Rational Rhapsody in Java models.
- ♦ **CGCompatibilityPre61M1C** makes the code generation backwards compatible with pre-6.1 maintenance release 1 Rational Rhapsody in C models.
- ♦ **CGCompatibilityPre61M1Cpp** makes the code generation backwards compatible with pre-6.1 maintenance release 1 Rational Rhapsody in C++ models.
- ♦ **CGCompatibilityPre70C** makes the code generation backwards compatible with pre-7.0 Rational Rhapsody in C models.
- ♦ **CGCompatibilityPre70Cpp** makes the code generation backwards compatible with pre-7.0 Rational Rhapsody in C++ models.
- ♦ **CGCompatibilityPre70Java** makes the code generation backwards compatible with pre-7.0 Rational Rhapsody in Java models.
- ♦ **CGCompatibilityPre71C** makes the code generation backwards compatible with pre-7.1 Rational Rhapsody in C models.
- ♦ **CGCompatibilityPre71Cpp** makes the code generation backwards compatible with pre-7.1 Rational Rhapsody in C++ models.
- ♦ **CGCompatibilityPre71Java** makes the code generation backwards compatible with pre-7.1 Rational Rhapsody in Java models.
- ♦ **CGCompatibilityPre72C** makes the code generation backwards compatible with pre-7.2 Rational Rhapsody in C models.
- ♦ **CGCompatibilityPre72Cpp** makes the code generation backwards compatible with pre-7.2 Rational Rhapsody in C++ models.
- ♦ **CGCompatibilityPre72Java** makes the code generation backwards compatible with pre-7.2 Rational Rhapsody in Java models.

- ◆ **CGCompatibilityPre73Ada** makes the code generation backwards compatible with pre-7.3 Rational Rhapsody in Ada models.
- ◆ **CGCompatibilityPre73C** makes the code generation backwards compatible with pre-7.3 Rational Rhapsody in C models.
- ◆ **CGCompatibilityPre73Cpp** makes the code generation backwards compatible with pre-7.3 Rational Rhapsody in C++ models.
- ◆ **CGCompatibilityPre73Java** makes the code generation backwards compatible with pre-7.3 Rational Rhapsody in Java models.
- ◆ **CGCompatibilityPre75C** makes the code generation backwards compatible with pre-7.5 Rational Rhapsody in C models.
- ◆ **CGCompatibilityPre75Cpp** makes the code generation backwards compatible with pre-7.5 Rational Rhapsody in C++ models.

Types of profiles

Rational Rhapsody provides the following types of profiles:

- ◆ Rational Rhapsody predefined [Profiles](#), such as AutomotiveC and DoDAF, can be selected from the New Project window when creating a new project or added to a project later, as described in [Adding a Rational Rhapsody profile manually](#).
- ◆ User-defined profiles, as described in [Creating a customized profile](#).
- ◆ Add On product profiles, which require an additional license, are automatically added when the engineer uses the associated add-on product, or it can be added using **File > Add Profile to Model**.

Converting packages and profiles

You can convert any existing package into a profile and vice versa. To convert a package, right-click the package in the browser, then select **Change to > Profile**. Similarly, to convert a profile to a package, right-click the profile in the browser, then select **Change to > Package**.

Profile properties

By default, all profiles apply to all packages available in the workspace. To associate existing profiles with new Rational Rhapsody annotation models, use the following properties (under `General::Profile`):

- ◆ **AutoCopied** specifies a comma-separated list of physical paths to profiles that will automatically be copied into new projects when they are created. By default, this property is an empty string.

- ◆ `AutoReferences` specifies a comma-separated list of physical paths to profiles that will automatically be referenced by new projects when they are created. By default, this property is an empty string.

Note that you can specify packages in the property values. The paths can be disks, networks, or Universal Naming Conventions (UNC), and the extension `.sbs` is optional. If Rational Rhapsody cannot find at least one of the specified profiles, it generates an error message.

You can use profiles like any other package, including exchanging them between users and using configuration management tools with them.

Use a profile to enable access to your custom help file

This section shows you how you can enable access to your custom help file from within Rational Rhapsody with the use of the **F1** key. This feature is applicable for a Rational Rhapsody project that has a Rational Rhapsody profile with a New Term stereotype defined for the profile. For information about profiles, see [Profiles](#). For information about stereotypes, see [Stereotypes](#).

As referred to by Rational Rhapsody, a custom help file consists of a help file and a map file, both of which you must create. There might be times when you have a project for which you would like to access your own help file. This might be particularly useful when there is a team working on the same project and you want them to share the same specific information. For example, your company might create its own help file to document its project terminology and project/corporate standards.

To enable the ability to access your custom help file from within Rational Rhapsody, you have to set properties to identify and locate your custom help file and map file. Then for an element associated with the New Term stereotype in the Rational Rhapsody profile for your Rational Rhapsody project, when a user presses **F1** to call a help topic, your custom help file would open instead of the main Rational Rhapsody help file.

Keep in mind that the Rational Rhapsody product provides you with an extensive help file that always displays when there is no custom help file available.

Note

IBM is not responsible for the content of any custom help file and map file.

The creation, functioning, testing, and maintenance of a custom help file and map file are the responsibilities of the creators of these files.

About creating your custom Help file and map file

A custom help file consists of a help file and a map file, both of which you create. See [About creating your custom help file](#) and [Creating your map file](#).

About creating your custom help file

Your custom help file must be in HTML format. (Example help file name: myhelp.htm.)

If you plan to share the custom help file with team members, place it in a shared location on a network. You identify this file and its location in the `Model::Stereotype::CustomHelpURL` property, as detailed in [Enabling access to your custom help file](#).

Creating your map file

The map file for your custom help file must contain one or more Rational Rhapsody resource IDs for which the custom help is provided. A resource ID corresponds to a particular GUID element in the Rational Rhapsody product. For example, for the **Attributes** tab, the resource ID is 161328.

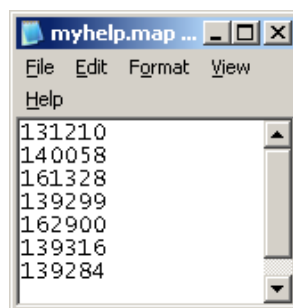
To find a Rational Rhapsody resource ID:

1. Add the following line to the [General] section of the `rhapsody.ini` file and save your changes:

`ShowActiveWindowHelpID=TRUE`

This means that now when you press **F1** to open the Rational Rhapsody help file, you will see a window with a resource ID instead, as shown in the next step.
2. In Rational Rhapsody, press **F1** for a GUID element that you plan to associate with a New Term stereotype (for example, if your New Term stereotype is applicable to a class, the **Attributes** tab of the Features window).
3. Add the resource ID number to your help map file. Each number must be on its own line. The following figure shows the contents of a map file called `myhelp.map`. (Another example of a help map file name: `myhelp.txt`.)

Note: For testing purposes as mentioned in [Enabling access to your custom help file](#), include the resource IDs shown in the following figure in your map file. You can delete these IDs later.



Note: You might find it useful to add the name of the element next to the resource ID number (for example, 161328 `Attributes` tab).

4. If you plan to share the custom help file with team members, place the help map file in a shared location on a network. You identify this file and its location in the `Model::Stereotype::CustomHelpMapFile` property, as detailed in [Enabling access to your custom help file](#).

Note: When your map file is complete, to ensure that help topics appear instead of resource ID numbers when you press **F1**, you can comment out the `ShowActiveWindowHelpID=TRUE` line in the `rhapsody.ini` file (for example, insert a semicolon or pound sign in front of the line) or you can delete the line.

Bookmarking your custom help file for a specific resource ID

When you press **F1** to open your custom help file (when possible), it opens at the beginning of the help file.

If for a particular resource ID (for example, the **Tags** tab of the Features window) you want your custom help file to open to a specific spot in the custom help file, add a bookmark to that spot in the help file. Use the standard HTML `<a name>` and `</a>` tags.

For example, the `myhelp.htm` help file has three sections labeled: Attributes, Ports, and Tags. When you open the help file, you see the Attributes section first. If for the **Tags** tab, which is resource ID 139316, you want to open the custom help file at the Tags section of the help file, you would code it as follows in the `myhelp.htm` help file:

```
<a name="139316">Tags</a>
```

Then when you press **F1** on the **Tags** tab, the custom help file opens at the Tags section of the help file.

Enabling access to your custom help file

To make these instructions easier to follow, they assume that there is a fictional project called **AutomobileProject** that has a profile called **Auto2009** with a New Term stereotype called **Stereotype_Auto2009**. These profile and stereotype names are for illustrative purposes only; they are not provided in Rational Rhapsody. (For information about creating a profile, see [Creating a customized profile](#). For information about stereotypes, see [Stereotypes](#).)

Note

These instructions assume you have created your custom help file and your map file. See [About creating your custom help file](#) and [Creating your map file](#).

To enable access to your custom help file:

1. For the profile or its stereotype for the Rational Rhapsody project on which you want to provide custom help, set the `Model::Stereotype::CustomHelpURL` and `CustomHelpMapFile` properties as follows:
 - a. Open the Features window for the profile or its New Term stereotype. On the Rational Rhapsody browser, double-click the profile name **or** stereotype name (for example, profile **Auto2009** or stereotype **Stereotype_Auto2009**).
 - b. On the **Properties** tab, select **All** from the **View** drop-down arrow (the label changes to **View All**).
 - c. Locate `Model::Stereotype::CustomHelpMapFile`. Type the path to your help map file.

Note: You can specify an environment variable as part of the URL (for example, `$DODAF_HLP_ROOT\dodaf_help.map`).

- d. Locate `Model::Stereotype::CustomHelpURL`. Type the path to your help file. You can specify an environment variable as part of the URL (for example, `$DODAF_HLP_ROOT\dodaf_help.htm`).
 - e. Click **OK**.
 2. Create an element using the New Term stereotype. Right-click a package in your project and select **Add New > Auto2009 > Stereotype_Auto2009**. Notice that these choices are located at the bottom of the pop-up menu.
 3. Double-click the stereotype created in the previous step (for example, `stereotype_auto2009_0`) and press **F1**. If you followed these steps and used the resource IDs show in [Creating your map file](#), your custom help file should open.

Testing the custom help file

Test your custom help file before releasing it to the users of the help file. You should be sure the help file works as expected before releasing it. For example:

- ◆ If the help file is to be accessed over a network, make sure it is accessible. You might want to have someone else (besides the person who set it up) try to access the custom help file from within Rational Rhapsody over the network before rolling out the feature to the rest of your project team.
- ◆ If you have set it so that your custom help opens at specific spots in the help file for specific resource IDs, be sure to check these particular links. See [Bookmarking your custom help file for a specific resource ID](#).

Using the custom help file

Once set up, anyone who uses the profile and elements with the New Term stereotype you set up can use the **F1** key to open the custom help file when and where it is possible. When you press **F1**, Rational Rhapsody checks if there is a custom help file for the New Term stereotype associate with the element.

- ♦ If yes, Rational Rhapsody searches for the resource ID in your help map file and it opens the custom help file.
- ♦ If no, Rational Rhapsody opens its main help file.

Note

The GUI element must be applicable to the New Term stereotype for the Rational Rhapsody profile for your custom help file to work.

For example, if the **Auto2009** profile has a New Term stereotype called **Stereotype_Auto2009** and you add an element with this stereotype to a package called **Default**, when you open the Features window for one of these stereotypes (for example, **stereotype_auto2009_0**) and then you press **F1**, your custom help should open (assuming you added the resource IDs for the tabs on this window to your help map file).

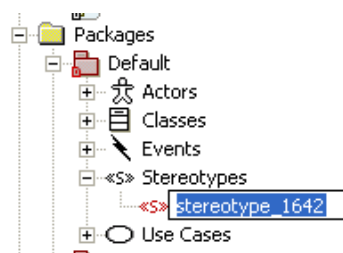
However, when you open the Features window for the package and you press **F1**, you open the main Rational Rhapsody help file because this package is not associated with the New Term stereotype call **Stereotype_Auto2009**.

Stereotypes

Defined stereotypes are displayed in the browser. Stereotypes can be owned by both packages and profiles.

To define a stereotype:

1. In the Rational Rhapsody browser, right-click the profile or package that owns the stereotype and then select **Add New > Stereotype**. Rational Rhapsody creates a new stereotype called `stereotype_n` under the **Stereotypes** category.



2. Enter a name for the new stereotype.
3. Double-click the new stereotype to open its **Features window**:
4. Select one or more metaclasses to which the stereotype is applicable to.
5. For profiles, if you are working in a domain with specialized terminology and notation, select the **New Term** check box to create a new metaclass. Since a term is based on an out-of-box metaclass, it functions in the same manner as its base metaclass. For more information about new terms, see [Special stereotypes](#).
6. Click **OK**.
7. Optionally, to set the formatting for the stereotype, right-click the new stereotype name in the browser and select **Format**. Use the Format window to define the visual characteristics for the stereotype.

Associating stereotypes with an element

You associate stereotypes with a model element using the **Stereotype** field of the Features window for that element. The **Stereotype** field includes a list of all the stereotypes defined in the profiles and packages that can extend the metaclass of that element. For example, if the element is a class, the **Stereotype** list includes all the stereotypes in all the profiles and packages that extend that class.

To associate stereotypes with an element:

1. Open the Features window for the element.
2. Open the **Stereotype** list.
3. Use the check boxes to select the stereotypes you would like to apply to the element.
4. Click the arrow of the list to close the list.
5. Click **Apply** or **OK**.

Alternatively, you can select the stereotypes from a tree display, as follows:

1. Open the Features window for the element.
2. Click the Browse button next to the list.
3. Find the stereotypes you would like to apply. Use **Ctrl** and **Shift** to select more than one stereotype.
4. Click **OK** to close the tree display.
5. Click **Apply** or **OK**.

Associate a stereotype with a new term element

Stereotypes can also be applied to elements that are based on “new term” stereotypes. In such cases, the **Stereotype** list will not contain the stereotype on which the element is based, nor any other “new term” stereotypes.

Re-ordering stereotypes in a list

Stereotypes can be selected for display at the top of the list.

To change the order in which the selected stereotypes are displayed:

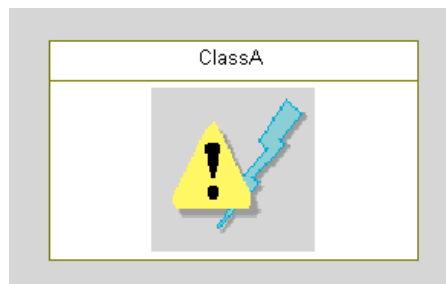
1. Click the Change Stereotype Order button , or right-click the stereotype list when it is closed and select **Edit Order**.
2. When the list of selected stereotypes is displayed, use the up and down arrows to reorder the list.
3. Click **OK**.

Associating a stereotype with a bitmap

Rational Rhapsody provides a set of predefined bitmaps in the directory <Rational Rhapsody_installation>\Share\PredefinedPictures. You can associate these icons with Rational Rhapsody stereotypes and classes.

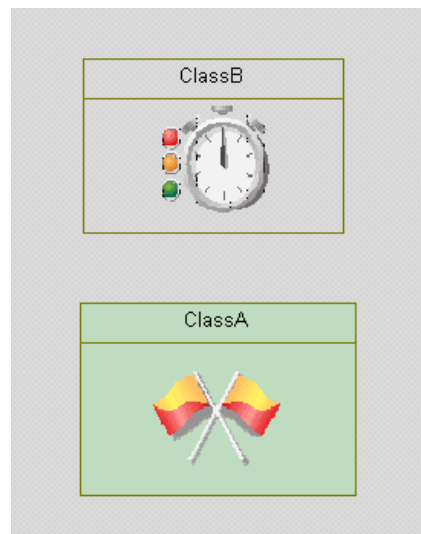
1. Define a stereotype of your own or select one of the existing stereotypes.
2. Select this stereotype for a class.
3. Rename an existing .bmp file (or add a new file) in the PredefinedPictures directory so it has the same name as the stereotype.
4. Drag-and-drop the class into an OMD.
5. Right-click the class and then select **Display Options**.
6. Under **Show stereotype**, select **Icon**; change **Display name** to **Name only**.

The bitmap is included in the class box.



To change the bitmap background to transparent (so only the graphic itself is visible), set the property `General::Graphics::StereotypeBitmapTransparentColor` to the RGB value of the bitmap background.

The following figure shows bitmaps with transparent backgrounds.



Deleting stereotypes

1. Select the stereotype to delete.
2. Click the **Delete** button.
3. Click **OK**.

Establishing stereotype inheritance

Stereotype inheritance allows you to extend existing stereotypes. Stereotypes can inherit from predefined stereotypes or from stereotypes that you have created.

The derived stereotype inherits the following characteristics from its base stereotype:

- ◆ Applicability (which elements it can be applied to)
- ◆ Properties (this includes locally-overridden properties)
- ◆ Tags

While the initial values of properties for the derived stereotype are those that were inherited from the base stereotype, the values can be overridden for the derived stereotype.

You can add tags to the derived stereotype, and add elements to the list of elements to which it can be applied.

To establish stereotype inheritance, in the browser, right-click the stereotype that will be the derived stereotype and select **Add New > Super Stereotype**

These steps can be repeated in order to create a stereotype that inherits from a number of other stereotypes.

Special stereotypes

If a stereotype inherits from one of the “special” stereotypes, for example, usage or singleton, it inherits the special meaning of the base stereotype.

If a stereotype inherits from a “new term” stereotype, then it is also a “new term” stereotype. However, the “new term” status of the derived stereotype is removed if you do something that contradicts this status, for example, using multiple inheritance such that the derived stereotype ends up being applicable to more than one type of element.

Use tags to add element information

Tags are used to add information to the model base specific to the domain or platform. You can access them using the **Tags** tab of the Features window for Rational Rhapsody model elements. The **Tags** tab shows the tag definitions and values associated with the given element. You can create tags for a stereotype, metaclass, or individual element.

Defining a stereotype tag

When the **Tags** tab applies to a stereotype, it specifies the tag definition for all elements that use the given stereotype. To define a new tag:

1. Create the profile to hold the tag if it does not already exist (see [Creating a customized profile](#)).
2. If it does not already exist, define a stereotype for the profile and select <<New>> (see [Stereotypes](#)). The Features window opens.
3. Select the **Tags** tab to display the window.
4. The **Quick Add** fields let you define the name for the tag and its default value quickly and click **Add**.
5. You might want to enter a more detailed description of the tag in the area above the Quick Add.
6. Click **OK**.

The new tag is added to the **Tags** tab. In addition, it is added to the browser.



This sample tag is listed as `Profile::Component::<TagName>` (in this example, `Avionics::Component::RiskFactor`) because it was defined under the stereotype of the profile. You would use this tag for components with the corresponding stereotype. For example, if you have a component named `System` with the `SafetyCritical` stereotype, its **Tag** tab would include the tag `Avionics::SafetyCritical::RiskFactor`.

Note

To create a new tag using the browser, right-click the stereotype and select **Add New > Tag**. You can rename the tag if you want.

Defining a global tag

When the **Tags** tab applies to a metaclass, it hosts all the tag definitions that are available to all instances of a certain type (anywhere within the model), without the need to set a stereotype. When you define a tag at the metaclass level, the **Applicable to** field is read-write so you can select the appropriate element type from the list.

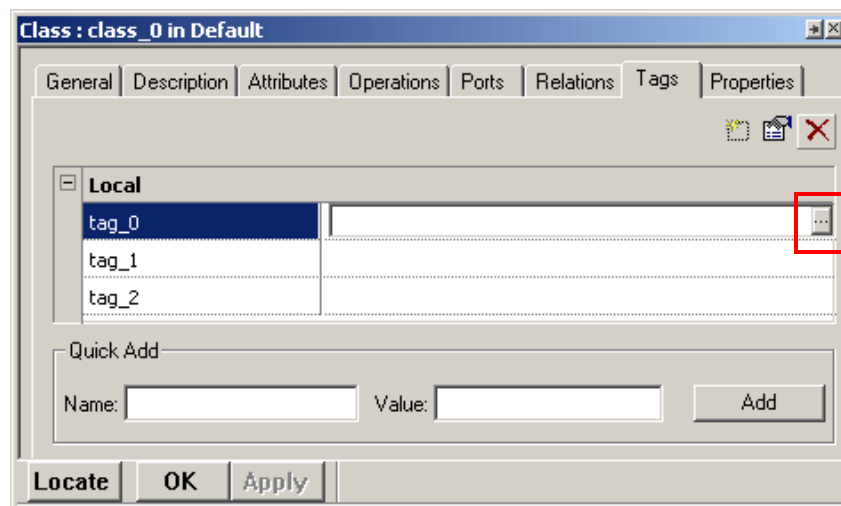
For example, you could create a new tag to specify prerequisite attributes for all primitive operations in the project by selecting the **Primitive Operation** element from the **Applicable to** list. This tag will be included automatically in the **Tags** tab of any primitive operation in the project as `<ProfileName>::<ElementType>::<TagName>`.

Defining a tag for an individual element

You can set a tag on an individual element to flag it in some way. When you create a tag for an individual element, it is listed in the tab as `LocalTags::Class::<TagName>`. For example, `Local::Class::Review`.

Adding a value to a tag

To add a value to a tag, click the Ellipsis button  at the right end of the box next to the name of the tag to open the internal text editor.



Using the internal text editor

The Rational Rhapsody internal text editor is a simple text editor on which you can enter text or code (depending on the functionality of the element you are working with). When using the internal text editor is possible, Rational Rhapsody provides you with access to it. For example, to open the internal text editor, you can click an Ellipsis button that displays on the **Tags** tab and the Properties of the Features window.

Deleting a tag

You can use the Rational Rhapsody browser or the Features window to delete a tag. Select.

To delete a tag using the browser:

1. Right-click the tag on the browser and select **Delete from Model**.
2. Click **Yes** to confirm your requested action.

To delete a tag using the Features window:

1. Open the Features window for the element to which the tag belongs and select the **Tags** tab.
2. Select the tag you want to delete and click the **Delete** button  in the upper-right corner of the tab.

Note

If you delete a tag definition in a stereotype, it is removed from the list of tags. However, if the value for the tag has been overridden, that tag will not be removed.

The Internal code editor

When you choose to edit code, Rational Rhapsody launches the internal code editor. This editor has a wide range of features to assist with editing.

Unlike external editors, the Rational Rhapsody internal code editor provides dynamic model-code associativity (DMCA). The DMCA feature of Rational Rhapsody enables you to edit code and automatically roundtrip your changes back into the model. It also generates code if the model has changed. If you use an external editor, DMCA will no longer be available.

Window properties

The Window Properties window enables you to customize the Rational Rhapsody internal code editor window. Note that these window properties are completely separate from Rational Rhapsody project properties. To open the window, right-click anywhere within the editor window and select **Properties**.

Alternatively, press **Alt+Enter** on the keyboard. You can customize this shortcut using the **Keyboard** tab on the Window Properties window. The window contains the following tabs:

- ♦ **Color/Font**
- ♦ **Language/Tabs**
- ♦ **Keyboard**
- ♦ **Misc**

The following sections describe how to use these tabs in detail.

The Color/Font tab

The internal code editor highlights syntax elements for easy editing. The default color settings follow standard code editing conventions. Using the **Color/Font** tab, shown in the figure, you can customize the colors and highlighting settings.

In addition to the default keywords (such as `class` and `public`), you can specify the additional language-specific keywords to be color-coded by the Rational Rhapsody internal code editor by

setting the value of the property `General::Model::AdditionalLanguageKeywords` to the comma-separated list of additional keywords you want to have color-coded.

Changing the default colors

The following table lists the default color and highlighting settings.

Item	Foreground	Background
Bookmarks	Default	Default
Comments	Green	Default
Keywords	Blue	Default
Left Margin	White	N/A
Numbers	Teal	Default
Operators	Red	Default
Scope Keywords	Blue	Default
Strings	Purple	Default
Text	Black	Default
Window	Default	N/A

To change the colors or highlighting:

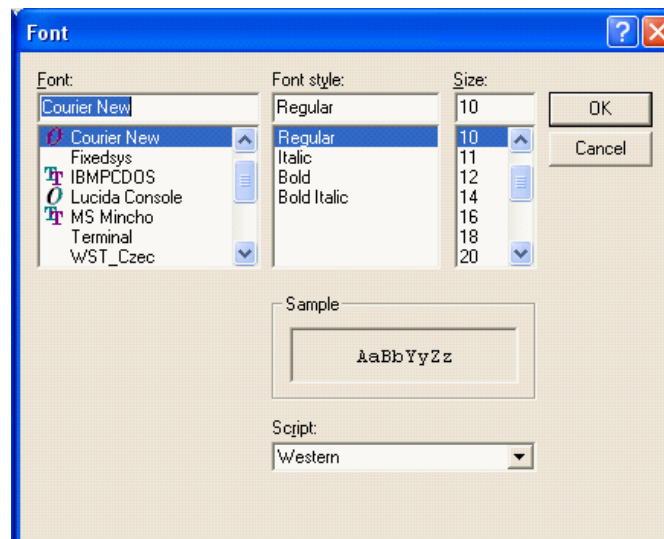
1. Select the **Color/Font** tab on the Windows Properties window.
2. Select a code element from the **Item** list.
3. Choose a text color by selecting it in the **Foreground** list.
4. Choose a highlighting color by selecting it in the **Background** list. Note that background highlighting is unavailable for all elements.
5. Click **Apply** to apply your changes and close the window.

Changing the default font

By default, the internal code editor uses Courier New, regular, 10-point font to display text. It supports any fixed-pitch font. When you change the font, it affects the appearance of all text in the editor.

To change the font:

1. In the Font area of the **Color/Font** tab, click **Change**. The Font window opens, as shown in the following figure.



2. As wanted, select new values for the **Font**, **Font style**, and **Size** fields. You can view the effects of your changes in the Sample field.
3. Click **OK** twice.

The Language/Tabs tab

The **Language/Tabs** tab, shown in the following figure, controls the indentation and tab size used by the editor.



The **Language/Tabs** tab contains the following fields:

- ♦ **Auto indentation style** specifies whether lines are automatically indented according to either the language scope or to the previous line in the file.

The possible values are as follows:

- **Off** turns off automatic indentation.
- **Follow language scoping** indents the code according to the language specifications. This is the default setting.
- **Copy from previous line** uses the indentation established in the previous line of code.

- ♦ **Tab** specifies how many spaces make up a tab space.

If you want the tab character converted to spaces after insertion, check the **Convert tabs to spaces while typing** box.

Note: Changes to tab size do not affect existing tab spacing. The new size applies to tabs entered after the change is made.

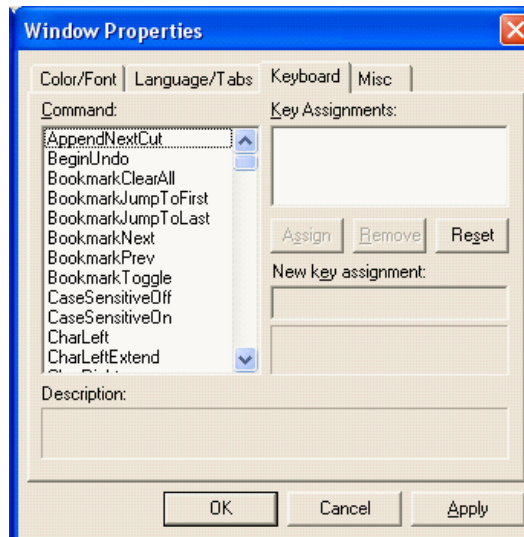
- ♦ **Language** specifies your programming language.

If you want any case errors corrected automatically for you, select the **Fixup text case while typing language keyword** check box. For example, if this option is selected and

you typed “WHile” as the keyword, the internal code editor automatically corrects the case of the keyword to read “while.”

The Keyboard tab

The **Keyboard** tab, shown in the following figure, allows you to create shortcuts.



The edit commands are started using keyboard shortcut keys. Mapping edit commands to easy-to-use and easy-to-remember keyboard shortcuts reduces the time and difficulty of editing text. Most commands have been mapped to a default shortcut. You can modify all shortcuts from the **Keyboard** tab of the Window Properties window.

Assigning custom keyboard mappings

The internal code editor provides customizable keyboard mappings so you can create your own shortcuts for edit commands, or assign shortcuts to commands that do not have a default assigned.

To assign keyboard shortcuts:

1. On the **Keyboard** tab, select an edit command from the **Command** list.

If a shortcut has been assigned to this command, it is displayed in the **Key Assignments** field.

A short description of the command is displayed in the **Description** field.

2. Click in the **New Key Assignment** box to activate it for editing.
3. Use the keyboard to type the shortcut key sequence. If you make a mistake, click **Reset** and type the sequence again.
4. Click **OK**.

Default keyboard mappings

The following table lists the default edit commands that have been assigned shortcut keys.

Command	Keystroke
BookmarkNext	F2
BookmarkPrev	Shift + F2
BookmarkToggle	Ctrl + F2
CharLeft	Left
CharLeftExtend	Shift + Left
CharRight	Right
CharRightExtend	Shift + Right
Copy	Ctrl + C
Copy	Ctrl + Insert
Cut	Shift + Delete
Cut	Ctrl + X
CutSelection	Ctrl + Alt + W
Delete	Delete
DeleteBack	Backspace

Command	Keystroke
DocumentEnd	Ctrl + End
DocumentEndExtend	Ctrl + Shift + End
DocumentStart	Ctrl + Home
DocumentStartExtend	Ctrl + Shift + Home
Find	Alt + F3
Find	Ctrl + F
FindNext	F3
FindNextWord	Ctrl + F3
FindPrev	Shift + F3
FindPrevWord	Ctrl + Shift + F3
FindReplace	Ctrl + Alt + F3
GoToLine	Ctrl + G
GoToMatchBrace	Ctrl +]
Home	Home
HomeExtend	Shift + Home
IndentSelection	Tab
LineCut	Ctrl + Y
LineDown	Down
LineDownExtend	Shift + Down
LineEnd	End
LineEndExtend	Shift + End
LineOpenAbove	Ctrl + Shift + N
LineUp	Up
LineUpExtend	Shift + Up
LowercaseSelection	Ctrl + U
PageDown	Next
PageDownExtend	Shift + Next
PageUp	PRIOR
PageUpExtend	Shift + Prior

Command	Keystroke
Paste	Ctrl + V
Paste	Shift + Insert
Properties	Alt + Enter
RecordMacro	Ctrl + Shift + R
Redo	Ctrl + A
SelectLine	Ctrl + Alt + F8
SelectSwapAnchor	Ctrl + Shift + X
SentenceCut	Ctrl + Alt + K
SentenceLeft	Ctrl + Alt + Left
SentenceRight	Ctrl + Alt + Right
SetRepeatCount	Ctrl + R
TabifySelection	Ctrl + Shift + T
ToggleOvertyping	Insert
ToggleWhitespaceDisplay	Ctrl + Alt + T
Undo	Ctrl + Z
Undo	Alt + Backspace
UnindentSelection	Shift + Tab
UntabifySelection	Ctrl + Shift + Space
UppercaseSelection	Ctrl + Shift + U
WindowScrollDown	Ctrl + Up
WindowScrollLeft	Ctrl + Page Up
WindowScrollRight	Ctrl + Page Down
WindowScrollUp	Ctrl + Down
WordDeleteToEnd	Ctrl + Delete
WordDeleteToStart	Ctrl + Backspace
WordLeft	Ctrl + Left
WordLeftExtend	Ctrl + Shift + Left
WordRight	Ctrl + Right
WordRightExtend	Ctrl + Shift + Right

The Misc tab

The **Misc** tab, shown in the following figure, controls numerous miscellaneous attributes for the editor.



The following sections describe how to use some of the more interesting features available on the **Misc** tab.

Using split views

You can split the editor window into up to four simultaneous views of one file, where each view scrolls independently from the others. You can make changes in any view and all other views are automatically updated.

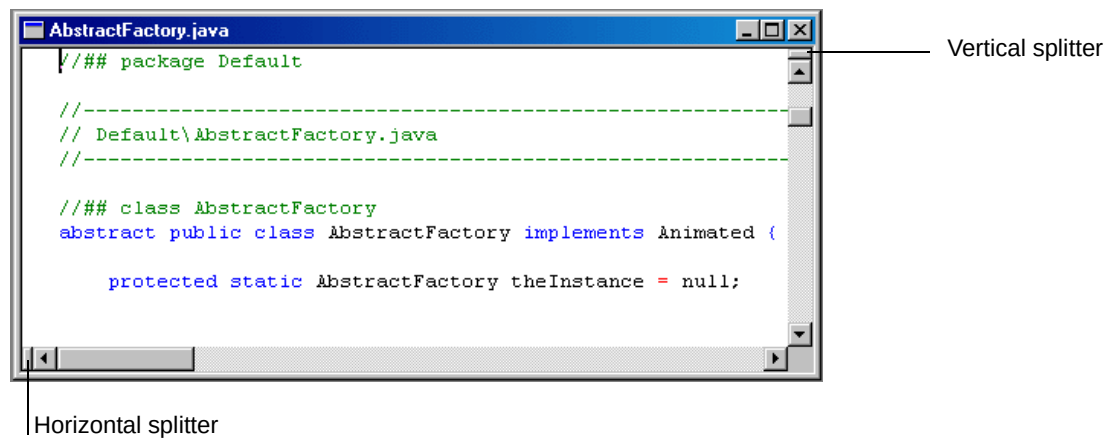
To allow splitting of the screen into two horizontal panes, check the **Allow Horizontal Splitting** box on the **Misc** tab. To allow splitting of the screen into two vertical panes, check the **Allow Vertical Splitting** box.

To split the screen horizontally:

1. Click the horizontal splitter at the left end of the horizontal scroll bar (see the figure).
2. While holding down the mouse button, drag the splitter to the right until the new pane displays.

To split the screen vertically:

1. Click the vertical splitter at the top of the vertical scroll bar.
2. While holding down the mouse button, drag the splitter downward until the new pane displays.



Mouse actions

Use the mouse to select and edit text in the editor window. The following table lists the available mouse actions.

Operation	Mouse Action
Display menu	Right-click.
Select entire line	Click in the left margin next to the target line of text.
Select multiple lines	Click in the left margin next to a line of text and drag the mouse up or down
Select entire word	Double-click anywhere in the word.
Move text (drag-and-drop)	Select text; hold down the left mouse button while dragging the selection to the new location.
Copy text (drag-and-drop)	Select text; press Ctrl and hold down the left mouse button while dragging the selection to the new location.

Using Undo and Redo

The internal code editor allows you to undo any number of operations, and redo the previous operation.

The edit commands for undo and redo are as follows:

- ◆ **Undo** use **Alt+Backspace**, or **Ctrl+Z**
- ◆ **Redo** use **Ctrl+A**

For information on editing keyboard shortcuts for these commands, see [Assigning custom keyboard mappings](#).

You can set the maximum number of undo operations from the Window Properties window; the default is to allow an unlimited number of undo operations.

To set the maximum number of undo actions:

1. On the **Misc** tab, in the **Maximum Undoable Actions** section, select **Limited to**.
2. In the adjacent box, type the maximum number of undo actions you want to allow.
3. Click **OK**.

Using the search feature of the internal code editor

The internal code editor contains a useful search feature that finds keywords within the current file.

To search for a keyword:

1. Right-click and select **Find**.
2. In the **What** box, type the search string.
3. Select one of the following options:
 - ♦ **Match whole word only** finds instances where the search term is the whole word.
 - ♦ **Match case** finds matching instances that have the same case as the search term.
 - ♦ **Direction** choose **Up** to search text above the cursor position, or **Down** to search text below the cursor position.
4. Click **Find** to find the next instance of the search term or click **Mark All** to place a bookmark in the left margin next to all instances matching the search term.

Bookmarks

The bookmark feature places a blue triangular marker in the left margin to identify a line of text.

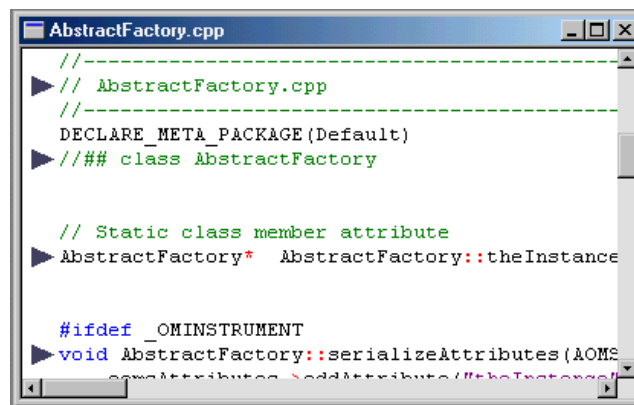
To place a bookmark in the margin, use the BookmarkToggle edit command. The default shortcut for BookmarkToggle is **Ctrl+F2**.

Repeat the BookmarkToggle command to remove the bookmark.

The bookmark commands are as follows:

- ◆ **BookmarkToggle** use **Ctrl+F2**
- ◆ **BookmarkNext** use **F2**
- ◆ **BookmarkPrev** use **Shift+F2**
- ◆ **BookmarkJumpToFirst** is unassigned
- ◆ **BookmarkJumpToLast** is unassigned
- ◆ **BookmarkClearAll** is unassigned

For instructions on editing the keyboard shortcuts for these commands, see [Assigning custom keyboard mappings](#).



Printing from the internal code editor

To print a file from the internal code editor:

1. Make sure that the editor window containing the file you want to print is the active window in Rational Rhapsody.
2. From the Rational Rhapsody **File** menu, select **Print**.
3. In the Print window, select the printing options, then click **Print**.

Graphic editors

The Rational Rhapsody graphic editors for the different UML diagrams provide the tools needed to create different views of a software model. Each graphic editor is described in detail in subsequent sections of this guide. This section describes the menus and features that are common to all the graphic editor.

Create new diagrams

You can use the **Diagrams** toolbar, Edit menu, Tools menu, or right-click menu commands in the browser to create a new UML diagram.

Creating new statecharts

Statecharts describe the behavior of a particular class. They can be added only at the class level. A class can have either a statechart or an activity diagram, but not both.

To create a new statechart, right-click a class in the Rational Rhapsody browser and select **Add New > Diagrams > Statechart**. A new (blank) diagram is displayed in the drawing area.

Note

Add New > Diagrams is the default menu command structure in Rational Rhapsody. It can be changed by users. This topic assumes that all defaults are in place.

Creating new activity diagrams

Activity diagrams describe the behavior of a particular class. They can be added only at the class level. A class can have either a statechart or an activity diagram, but not both.

To create a new activity diagram, right-click a class in the Rational Rhapsody browser and select **Add New > Diagrams > Activity**. A new (blank) diagram is displayed in the drawing area.

Note

Add New > Diagrams is the default menu command structure in Rational Rhapsody. It can be changed by users. This topic assumes that all defaults are in place.

Creating all other diagram types

The other diagrams can be grouped under the project or a package in the project hierarchy.

To create a new diagram other than a statechart or activity diagram:

1. Select the appropriate diagram type from **Tools > Diagrams** or click the appropriate button on the **Diagrams** toolbar:



Object Model Diagram



Component Diagram



Structure Diagram



Deployment Diagram



Use Case Diagram



Collaboration Diagram



Sequence Diagram



Panel Diagram

2. From the Open Diagram window, select the project or a package where you would like to add the diagram.
3. Click the **New** button. The New Diagram window opens.
4. Type a name for the new diagram in the **Name** box.
5. Depending on the diagram type, the New Diagram window can contain the following options:
 - a. **Populate Diagram**, which automatically populates the diagram with existing model elements. This option applies to object model, use case, and structure diagrams. For more information, see [Automatically populating a diagram](#).
 - b. **Operation Mode** specifies whether to create an *analysis* SD, which enables you to draw message sequences without adding classes and operations to the model; or a *design* SD, in which every message you create or rename is realized to an operation in the static model. For more information, see [Sequence diagrams](#).
6. Click **OK**. A new (blank) diagram is displayed in the drawing area.

Opening existing diagrams

To open an existing UML diagram, double-click the diagram in the Rational Rhapsody browser. The diagram opens in the drawing area.

Alternatively, you can:

1. Click the appropriate button in the **Diagrams** toolbar to open the Open <Diagram Name> window.
2. Depending on the type of diagram you want to open:
 - ♦ For statecharts and activity diagrams, select the diagram you want to open and click **OK**.
 - ♦ For all other diagrams, select the diagram (if there are any) you want to open and click **Open**.

As with other Rational Rhapsody elements, the Features window for the diagram enables you to edit its features, including the name, stereotype, and description. For more information, see [The Features window](#).

Navigating forward from opened diagram to opened diagram

To go forward from open diagram to open diagram, choose **Window > Forward**.

In addition, you can go to a currently opened diagram by selecting it from the last section of the Windows menu.

Navigating backwards from opened diagram to opened diagram

To go backwards from open diagram to open diagram, choose **Window > Back**.

In addition, you can go to a currently opened diagram by selecting it from the last section of the Windows menu.

Deleting diagrams

In most cases, you can delete existing UML diagrams only from the Rational Rhapsody browser. However, you can delete statecharts and activity diagrams from both the browser and from the Tools menu.

To delete an existing diagram:

1. Select the diagram from the browser.
2. Right-click and select **Delete from Model**, or press the **Delete** key.
3. Click **Yes** to confirm your action.

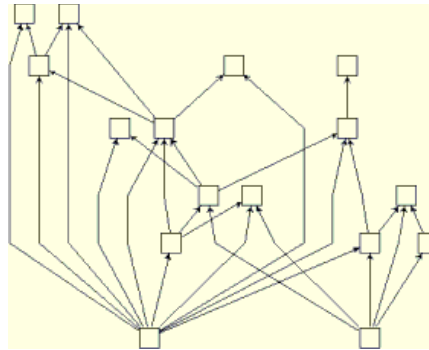
Automatically populating a diagram

When you create a new use case, object model, or structure diagram, you can use the **Populate Diagram** feature to populate the diagram automatically with existing model elements. You can select which model elements to add to the diagram. Rational Rhapsody automatically lays out the elements in an orderly and easily comprehensible manner.

Relation type styles

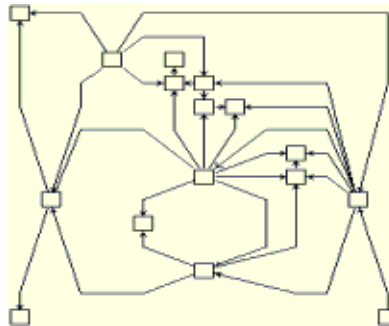
Once the diagram has been created, you can edit it by adding or deleting elements to tailor it to your needs. This feature is particularly useful for quickly creating diagrams after reverse engineering or importing a model. When you automatically populate a new diagram, Rational Rhapsody offers a choice of layout style. You can select either of these styles for any relation type:

- ♦ **Hierarchical**
The elements are organized according to levels of inheritance, with lower levels inheriting from upper levels. Each layer is organized to minimize the crossing and bending of inheritance arrows and is individually centered in the diagram. You can choose this style for any type of relation. If there are no classes or inheritance, the elements might appear organized as layers, depending on the direction of the populated diagrams.



♦ **Orthogonal**

The entire drawing is made as compact as possible. Classes are placed to minimize the intersection, length, and bending of relation arrows. You can use this style for any relation, including inheritance relations.



Creating and populating a new diagram

To create a new, automatically populated diagram:

1. In the browser, right-click an existing use case, object model, or structure diagram category where you want to create another of the same diagram and select **Add New <element type>** (for example, **Add New Object Model Diagram**) to open the New Diagram window.
2. Type the **Name** of the new element.
3. Select the **Populate Diagram** check box and click **OK** to open the Populate Diagram window.
4. In the **Create Contents Of Diagram Using** group, indicate how you would like Rational Rhapsody to create the contents of the diagram:
 - ♦ **Relations Among Selected** populates the diagram only with the selected elements and the relations between them

- ♦ **Relations From/To Selected** populates the diagram with the selected elements, their incoming and outgoing relations, and the model elements that complete these relations
 - ♦ **Relations From Selected** populates the diagram with the selected elements, their outgoing relations, and the model elements that complete the relations
 - ♦ **Relations To Selected** populates the diagram with the selected elements, their incoming relations, and the model elements that complete the relations
5. In the **Types Of Relations To Be Used** group, select which types of relations you would like Rational Rhapsody to use when creating the contents of the diagram:
 - ♦ OMDs and structure diagrams: **Instance**, **Association/Aggregation**, **Inheritance**, **Dependency**, **Link**, and **Anchor/Annotations**
 - ♦ UCDs: **Association**, **Generalization**, and **Dependency**, and **Anchor/Annotations**
 6. In the **Selection** box, place a check mark next to each element you want to include in the new diagram. To select a package without selecting the elements it contains, right-click the package.
 7. In the **Preferred Layout Style** group, select the type of layout you would like Rational Rhapsody to use when creating the diagram. If you select **None**, Rational Rhapsody automatically chooses the best layout style according to the type of relations you have chosen to display.
 8. Click **OK**. The new diagram displays in the drawing area with all of the selected elements added. You can then begin to add information to the new diagram.

Note

When auto-populating a diagram, if you want Rational Rhapsody to populate each class so it shows its attributes and operations, set the `ObjectModelGE::Class::ShowAttributes` and `ObjectModelGE::Class::ShowOperations` properties to `All` in the scope of the package or project.

Automatically populating existing diagrams

Use the Populate Diagram window to automatically add elements and their relations to an existing and already populated object model, use case, or structure diagram. To open this window, do any of the following actions:

- ♦ Right-click a blank spot on the diagram in its drawing area and select **Populate** or choose **Layout > Populate**.
- ♦ Right-click the diagram on the Rational Rhapsody browser and select **Populate**.

See the details for the Populate Diagram window in [Creating and populating a new diagram](#).

Note

Rational Rhapsody does not change the location of already existing elements. It just adds the new elements.

About reverse engineering object model diagrams

During reverse engineering with the **Populate Diagrams** check box selected, Rational Rhapsody creates object model diagrams visualizing the elements added during reverse engineering. If you run reverse engineering subsequently with the **Merge existing package** option, Rational Rhapsody updates the visualized diagrams to show the new elements added and the dependencies between the diagram elements.

Limitations of populating existing diagrams automatically

The Populate feature does not support the adding of ports, though existing ports on the diagram will be preserved.

Property settings for the diagram editor

The properties under `General::Graphics` control how features of the diagram editors operate. The following table lists the available properties.

Property	Description
<code>AutoScrollMargin</code>	Controls how responsive the autoscrolling functionality is
<code>ClassBoxFont</code>	Specifies the default font for new class names
<code>CRTerminator</code>	Specifies how multiline fields in notes and statechart names should interpret a carriage return (CR)
<code>DeleteConfirmation</code>	Specifies whether confirmation is required before deleting a graphical element from the model
<code>ExportedDiagramScale</code>	Specifies how an exported diagram is scaled and whether it can be split into separate pages for better readability
<code>FixBoxToItsTextuals</code>	Specifies whether to resize boxes automatically to fit their text content (such as names, attributes, or operations)
<code>grid_color</code>	Specifies the default color used for the grid lines
<code>grid_snap</code>	Specifies whether the Snap to Grid feature is available for new diagrams, regardless of whether the grid is actually displayed
<code>grid_spacing_horizontal</code>	Specifies the spacing, in world coordinates, between grid lines along the X-axis when the grid is available for diagrams
<code>grid_spacing_vertical</code>	Specifies the spacing, in world coordinates, between grid lines along the Y-axis when the grid is available for diagrams
<code>HighlightSelection</code>	Specifies whether items should be highlighted when you move the cursor over them in a diagram
<code>LandScapeRotateOnExport</code>	Rotates an exported metafile so it can fit on a portrait page
<code>MaintainWindowContent</code>	Specifies whether the viewport (the part of a diagram displayed in the window) is kept for window resizing operations when you change the zoom level, providing additional space in the diagram in a smooth manner
<code>MarkMisplacedElements</code>	Specifies whether misplaced elements are marked in a special way. Previously, misplaced elements were shown with a small X in the upper corner
<code>MultiScaleByOne</code>	Specifies whether objects in the diagram keep the same amount of space between them when you scale them (Cleared)
<code>PrintLayoutExportScale</code>	Specifies the factor by which the Windows metafile format (WMF) files are scaled down in order to fit on one page
<code>RepeatedDrawing</code>	Specifies whether repetitive drawing mode (stamp mode) is available

Property	Description
ShowEdgeTracking	Specifies whether to show the “ghost” edges of a linked element when you move it
ShowLabels	Specifies whether to display labels instead of names in diagrams
StereotypeBitmap TransparentColor	Creates a “transparent” background for bitmaps associated with stereotypes (so only the graphics are displayed in the class box)
Tool_tips	Enables the display of tooltips

You can set these properties by selecting **File >Project Properties**.

For detailed information on how the General::Graphics properties affect the drawing of your model, see the definitions displayed in the **Properties** tab of the Features window.

Setting diagram fill color

To set the color of the diagram background:

1. Right-click in the diagram window and then select **Diagram Fill Color**.
2. Select a color.

Create elements

The diagram editors enable you to create a number of elements to enhance your model. To create any kind of element, you must first select one of the drawing tools on the diagram editor toolbar.

When Rational Rhapsody is in drawing mode, the cursor includes a tooltip showing an icon of that element. For example, the following cursor is displayed when you are drawing a class.



The items you can draw in the diagram editors fall into two main categories: boxes and lines. In addition, Rational Rhapsody enables you to create freestyle shapes. The following sections describe how to create these elements.

Repetitive drawing mode

By default, each time you want to add an element to a diagram, you must first click the appropriate icon in the **Diagram Tools**.

In some cases, however, you might want to add a number of elements of the same type. To facilitate this, Rational Rhapsody includes a *repetitive drawing mode*.

To enter repetitive drawing mode, click the “stamp” icon in the **Layout** toolbar. Now, after choosing a tool in the **Diagram Tools**, you will be able to continue drawing elements of that type without selecting the tool again each time. If you choose a different tool from the toolbar, then Rational Rhapsody will allow you to draw multiple elements of the newly selected type.

After you click the icon, Rational Rhapsody remains in repetitive drawing mode until you turn it off. To turn off the repetitive mode, just click the "stamp" icon a second time.

Drawing boxes

Each editor contains tools to draw boxes. The following table lists the box elements available with each editor.

Diagram Editor	Box Elements
Object model	Classes, packages, composite classes, objects, files, actors, and annotations
Use case	Use cases, actors, systems boundary boxes, packages, and annotations
Component	Components, files, folders, and annotations
Deployment	Nodes, components, and annotations
Collaboration	Objects, multi-objects, actors, and annotations
Statechart	States and annotations
Activity	Actions, subactivities, action blocks, object nodes, swimlane frames, connectors, and annotations
Structure	Composite classes, objects, and annotations

To draw a box:

1. Select a box tool.
2. Move the cursor to the drawing area, and do one of the following actions:
 - a. **Quick-Draw** where you can click once to draw a box with the default size, shape, and name.

- b. Drag** where to draw classes, simple classes, and packages, you click-and-drag to the opposite corner and release. If you hold down the Shift key, you create a square box.

Note that the cursor changes to the icon of whatever you are drawing (class, object, package, and so on).

3. When you complete the box, the element is given a default name. To rename it, click the text to enable editing. Type the new name, then press **Enter** or click outside the box.
4. Click anywhere outside the box to start a new box, or click the **Select** tool to stop creating boxes.

To add an existing box element to a diagram, you can simply drag-and-drop the element from the browser to the drawing area.

Drawing arrows

Arrows connect boxes, representing the connections between different boxes in the diagram. For example, associations and dependencies are two types of arrows that can be drawn in use case diagrams.

You can draw arrows in the following ways:

- ♦ **Drag** where you click inside the source box, drag, and release the mouse inside the destination box.
- ♦ **Simple arrow clicks** where you click inside the source box and click the border of the destination box.
- ♦ **Multiple clicks** where you click inside the source box, click any number of times to mark the control points on the path of the arrow, and double-click inside the destination box. You cannot add control points to a message arrow because it allows only two clicks. Another point displays immediately to show where the next arrow will start.

Note that the cursor changes when you create arrows:

- ♦ If you click a valid element for the destination of the arrow (for example, you are drawing a activity flow and you click a class), the cursor changes to crosshairs in a small circle.
- ♦ If you try to connect an arrow to an invalid element, Rational Rhapsody displays a “no entry” symbol to show that you cannot connect the arrow to that element, as shown in the following figure.



Changing the line shape

Rational Rhapsody has three line shapes that you can use when you are drawing arrows in the graphic editors. You can set a unique line shape for each line; the default line shape varies by element.

The Edit menu contains the **Line Shape** option, which specifies your preferred line shape for the specified arrow. You can also access the **Line Shape** option by right-clicking an arrow or line in a diagram.

The possible values for the line shape are as follows:

- ◆ **Straight** changes the line to a straight line.
- ◆ **Spline** changes the line to a curved line.
- ◆ **Rectilinear** changes the line to a group of line segments connected at right angles.
- ◆ **Re-Route** removes excess control points to make the line more fluid.

Note

You can use the `line_style` property to change the line shape (straight, spline, rectilinear) for a line type (for example, Link, Dependency, Activity Flow, Generalization) for a diagram type (for example, Object Model, Activity, Statechart) by project. For example, in your Handset project, you can use the `ObjectModelGe: :Depends: :line_style` property to set Dependency lines for Object Model diagrams to be a spline shape. You should set this property when you begin a project, as it takes effect going forward.

In addition, you can customize a line by adding/removing user-defined points to an arrow element.

To add points to an arrow element:

1. In the diagram, select the line you want to change.
2. Right-click and select **User Points**.
3. Depending on what you want to do:
 - ◆ To add a user-defined point, click **Add**. Note that the new point is added at the location where you accessed the menu.
 - ◆ To remove a point, click **Delete**. Rational Rhapsody removes the point closest to the location where you accessed the menu.

Note that you can reshape the line when the cursor changes to the icon shown in the following figure:



Simply drag the line to reshape it.

Naming boxes and arrows

Use either of the following methods to name boxes and arrows:

- ♦ When you draw a box or a line, the cursor displays in the name field so you can edit it immediately. Type a name and press **Ctrl+Enter**. Note that names, except for activity flow labels, are limited to one line. To add an extra line to a activity flow label, press **Enter**.
- ♦ For activity flows in statecharts and activity diagrams, you can select the **Name** tool. The cursor changes from an arrow to a pen. Click to select the name position, type a name, and click outside the element to finish.

To edit an existing name, do one of the following actions:

- ♦ Double-click the name to enable editing, and type a new name.
- ♦ Open the Features window and edit the **Name** field on the **General** tab.

Note

If you rename a class box, you are renaming the class in the model.

Draw freestyle shapes

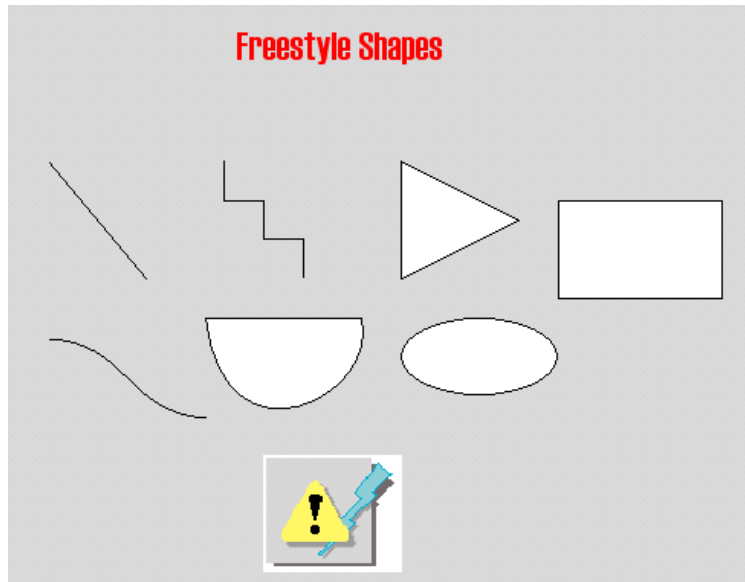
Use the **Free Shapes** tool, which are displayed in its own section on the **Diagram Tools** panel, to draw elements freehand in a diagram. To display or hide the **Diagram Tools** panel, choose **View > Toolbars > Drawing**.

The **Free Shapes** toolbar provides tools that enable you to customize your diagrams using freestyle shapes and graphic images.

The **Free Shapes** toolbar includes the following tools:

Tool	Name	Description
	Line	Draws a straight line between the selected endpoints.
	Polyline	Draws a polyline using multiple points.
	Polygon	Draws a polygon.
	Rectangle	Draws a rectangle.
	Polycurve	Draws a curve.
	Closed Curve	Draws a closed, curved shape within the bounds of the specified shape.
	Ellipse	Draws a circle or ellipse.
	Text	Draws free text.
	Image	Enables you to import an image into the diagram.

The following figure shows examples of each freestyle shape.



The following sections describe how to draw these freestyle shapes.

Drawing lines and polylines

To draw a simple line:

1. Click the **Line** tool.
2. Click in the diagram and drag the cursor away from the endpoint to create the line. The line is shown as a dashed line until you click to end the line.

If you click too early so only the square endpoint is displayed, click the endpoint and redraw the line.

To draw a polyline using several points:

1. Click the **Polyline** tool.
2. Click to place the first endpoint.
3. Drag the cursor away from the endpoint, clicking once to place each subsequent point in the polyline.
4. and once to place each point along the line.
5. Double-click to end the line.

Drawing polygons

To draw a polygon:

1. Click the **Polygon** tool.
2. Click twice in the diagram to define two endpoints of a side, then drag and click to define the polygon.

For example, to draw a triangle, simply define one side and drag the cursor to form the triangle.

3. Click twice to complete the polygon.

Drawing rectangles

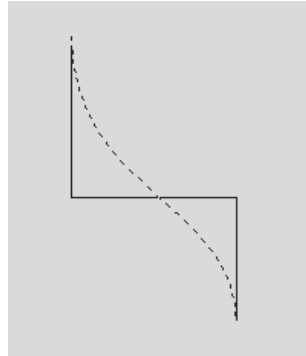
To draw a rectangle:

1. Click the **Rectangle** tool.
2. Click once in the diagram. By default, Rational Rhapsody draws a square with the selection handles active.
3. Use the selection handles to create a rectangle.

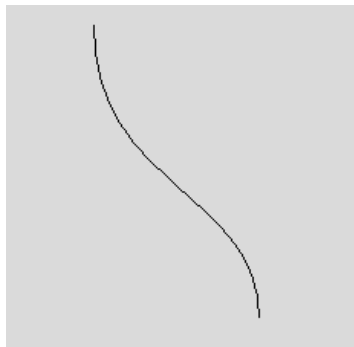
Drawing polycurves and closed polycurves

To create a curve:

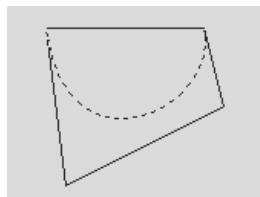
1. Click the **Polycurve** tool.
2. In the diagram, click to define several points that define the curve. As you define points (and, therefore, line segments), the resulting curve is drawn as a dashed line.



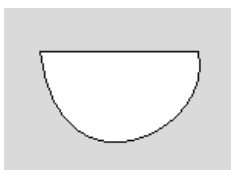
3. Double-click to create the curve.



Similarly, the **Closed Polycurve** tool creates a closed polycurve. As you define line segments, a dashed line shows the shape of the resulting closed curve.



Double-click to create the closed curve.



Drawing ellipses and circles

To draw an ellipse:

1. Click the **Ellipse** tool.
2. Click once in the diagram to place the left-most point of the ellipse.
3. Holding down the mouse button, drag the cursor to create the ellipse or circle.
4. Release the mouse button to complete the shape.

Drawing text

To create floating text:

1. Click the **Text** tool.
2. Click once in the diagram to place the text box.
3. Type the wanted text.
4. Press **Enter** for a new line; press **Ctrl-Enter** to place the text and dismiss the text box.

To change the text color, font, size, and so on, right-click the text and select **Format**. For more information on changing text attributes, see [Change the format of a single element](#).

Adding images

To add an image to your diagram:

1. Click the **Image** tool. The Open window displays.
2. Navigate to the directory that contains the image you want to add to the diagram.

Note

The Rational Rhapsody distribution includes numerous bitmaps for common design elements, such as CDs, timeouts, displays, and so on. These images are available in `Share\PredefinedPictures` under the root installation directory.

3. Select the image to add.
4. Move the cursor to where you want to add the image, then click once to place it.

Deleting freestyle shapes

To delete a freestyle shape or graphic image:

1. In the diagram, select the shape to delete.
2. Click the **Delete** tool.

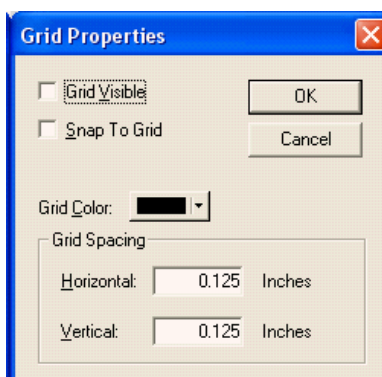
Placing elements using the grid

You can display a grid and rulers to assist with positioning elements in all the graphic editors except the sequence diagram editor.

To display the grid, click the **Grid** tool or select **Layout > Grid > Show Grid**.

Setting the grid properties

To set the attributes of the grid, select **Layout > Grid > Grid Properties**. The following figure shows the resultant window.



You can set the following properties for the grid:

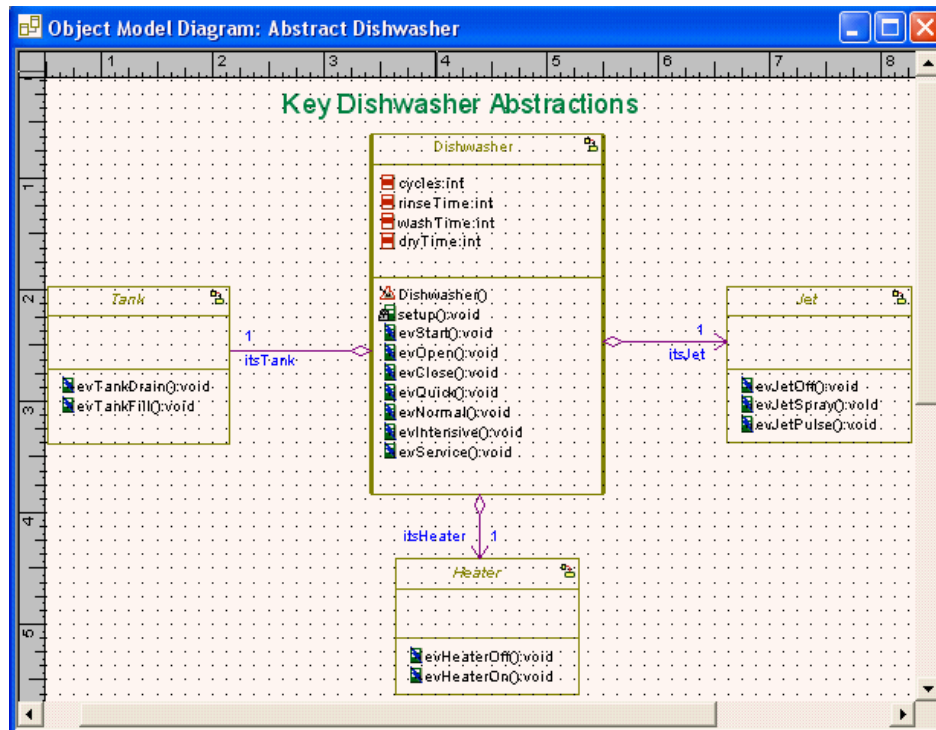
- ♦ **Grid Visible** specifies whether the grid is displayed.
- ♦ **Grid Color** specifies the color used for the grid dots. The default color is black.
- ♦ **Grid Spacing** specifies the horizontal and vertical spacing of the grid points, in inches.

Snapping to the grid

To move an element you are drawing automatically to the closest grid points, select **Layout > Grid > Snap to Grid**.

Displaying the rulers

To display rulers in the drawing area, click the **Rulers** tool or select **Layout > Show Rulers**. The following figure shows an OMD with both the grid and rulers available.



Autoscroll

By default, Rational Rhapsody automatically scrolls the diagram view while you are busy doing another operation (such as moving an existing box element or drawing new edges by dragging) that prevents you from doing the scrolling yourself. The autoscroll begins scrolling when the mouse pointer enters the autoscroll margins, which are virtual margins that define a virtual region around the drawing area (starting from the window frame and going *X* number of points into the drawing area).

You can change the size of the autoscroll margins by setting the property `General::Graphics::AutoScrollMargin`. This property defines the *X* number of points the margins enter into the drawing area. If you specify a large number for this property, the margin becomes bigger, thereby making the autoscroll more sensitive.

Set this property to 0 (no scroll region) to disable autoscrolling.

Select elements

There are many ways Rational Rhapsody enables you to select elements in diagrams. You can select an element using the mouse or using a variety of menu commands. Once you have selected an element, you can edit it depending on what kind of element it is.

Selecting elements using the mouse

To select an element using the mouse:

1. Click the **Select** tool. When you move the mouse over the diagram, a standard mouse pointer is displayed.
2. Click an element. When you select an element, all other elements are deselected.
 - a. To select a line or an arrow, click anywhere on it.

You can select the control point of an arrow only by clicking and dragging. For more information, see [Clicking-and-dragging](#).

- b. To select a box, click anywhere inside it or on its border.

Selecting elements using the edit menu

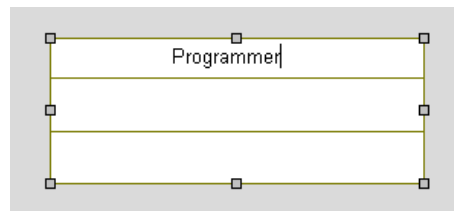
Use **Edit > Select** to access the following commands for making selections:

- ♦ **Edit > Select > Select All** selects all elements in the diagram.
- ♦ **Edit > Select > Select Next** selects the element next to the current one, when two elements are close together. This lets you easily navigate to each element in a diagram, one at a time.
- ♦ **Edit > Select > Select by Area** enables you to draw a selection box around a group of elements within a container element (for example, classes in a package).
- ♦ **Edit > Select > Select Same Type** selects all of the elements in the diagram that are of the same type as the element currently selected. If more than one type of element is selected, then all the elements of the different selected types will be selected.

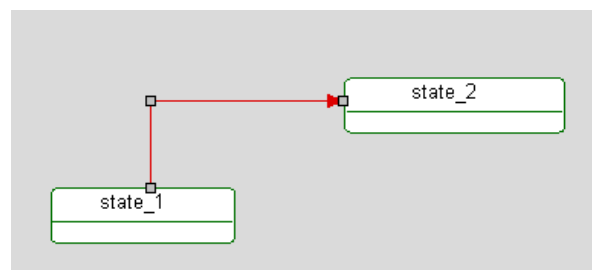
Selection handles

When an element is selected, selection handles are displayed around its edges. The following diagrams selection handles on different elements.

Boxes have markers on each corner and usually on each side.



Arrows and lines have a marker on each end and on each control point on the line.



Non-rectangular elements, such as use cases or actors, and groups of elements have an invisible bounding box with eight visible handles.

Note

The cursor changes depending on how the selected element will be changed. If the cursor is displayed as a four-pointed arrow, the selected element can be moved to a new location. If the cursor changes to a two-headed arrow, you can resize the element.

Selecting multiple elements

There are two ways to select more than one element:

- ◆ **Shift+Click**
- ◆ Clicking-and-dragging

Note that the last element selected in a multiple selection is distinguished by gray selection handles. Gray handles indicate that the element is an anchor component. Rational Rhapsody uses anchor components as a reference for alignment operations. If you want to use a different

component as the anchor, hold the **Ctrl** key down and click another element within the selection. Rational Rhapsody transfers the gray handles from the previous element to the selected element

For more information, see [Arranging elements](#).

Shift+Click

1. Click the first element you want to select.
2. Press and hold down the **Shift** key, then click the rest of the elements you want to select.

Note

To remove an element from the selection group, press and hold down the **Shift** key and click the element you want to remove from the selection.

Clicking-and-dragging

To make a multiple selection using the click-and-drag method:

1. Move the mouse pointer to a blank area of the diagram.
2. Press and hold down the left mouse button.
3. Drag to surround the area that contains the elements you want to select.
4. To add more elements to the selection, hold down the Shift key while you click-and-drag again.

Edit elements

You can edit elements using the following methods:

- ♦ **Features window.** For detailed information, see [The Features window](#).
- ♦ **Right-click menu.** When you select an element in a diagram and right-click, a menu lists common operations you can perform on that element. Many of these options are element-specific, but some of the common operations are as follows:
 - **Features** or **Features in New Window** displays the Features window for the specified element.
 - **Display Options** enables you to specify how elements are displayed in the diagram.
 - **Cut** removes the element from the view and saves it to the clipboard.
 - **Copy** saves a copy of the element to the clipboard and keeps it in the view.
 - **Copy with Model** copies an element such that when it is pasted into a diagram, a new element will be created in the model with the exact same characteristics as the original element.
 - **Remove from View** removes the specified element from the diagram but not from the model.
 - **Delete from Model** deletes the element from the model.
 - **Format** changes the format used to draw the element (color, line style, and so on). For more information, see [Change the format of a single element](#).
 - **Line Shape** enables you to change the shape of the line. For more information, see [Changing the line shape](#).
 - **User Points** enables you to add additional points to, or delete points from, a line element. This functionality enables you to customize the shape of the line. For more information, see [Changing the line shape](#). Element-specific options are described with the individual elements.
- ♦ **Manipulating the element in the diagram.** Use any of these methods to edit an element in a diagram:
 - Resize it.
 - Move its control points.
 - Move it to a new location.
 - Copy it.
 - Arrange it relative to one or more elements.
 - Remove it from the view.
 - Delete it from the model.
 - Edit any text associated with it.

Resizing elements

You can resize an element by stretching its sides. You can resize only one element at a time.

To resize an element:

1. Select the element.
2. Move the mouse pointer over one of its selection handles. When the mouse pointer is over a selection handle, it changes to a double-pointed arrow. If you are editing a line, you can also move the mouse pointer over a line segment.
3. Click-and-drag the mouse until the element is the wanted size and shape. If you want the element to maintain its proportions as you resize it, hold down the Shift key before you begin to click-and-drag.

If the box you are editing contains text, the text wraps to fit inside the boundaries of the new box. If you are editing a box that is connected to other lines or contains other elements, the lines and elements move and resize according to the box as you stretch it. If you hold down the **Shift** key, you can stretch diagonally while maintaining the scale for the element.

To prevent elements from being resized when you resize their parent, press and hold the **Alt** key while you click-and-drag with the mouse. Alternatively, you can select the menu item **Edit > Move/Stretch Without Contained** before resizing the element.

To make Rational Rhapsody automatically enlarge the text box of a box element to fit the size of an element name, select the **Expand to fit text** in the menu. Alternatively, select **Layout > Expand to fit text**. Note that once you apply this feature to an element, it remains available until you resize the element.

To use this functionality as the default behavior, set the property
`General::Graphics::FitBoxToItsTextuals` property to Checked.

Moving control points

If you have placed control points on arrows, you can select a control point and move it individually to change the curve of the line. This is particularly effective within statecharts where activity flows are rendered as spline curves.

See the description of the **Reroute** command in [Changing the line shape](#) for information on removing extra control points.

Moving elements

You can move an element by clicking on it and dragging it to a new location. You can move several elements simultaneously.

To move an element:

1. Select the element you want to move.
2. Move the mouse pointer over the element. The cursor changes to a four-pointed arrow, which denotes a move operation.
3. Click-and-drag the element to a new location.

Note the following information:

- ♦ To prevent elements from being moved when you move their parent boxes, hold down the Alt key while you click-and-drag with the mouse. Alternatively, you can select the menu item **Edit > Move/Stretch Without Contained** before moving the element.
- ♦ If you hold down the Ctrl key when moving an element, a copy of the selected element is created.
- ♦ If you hold the Shift key down when moving an element, you can move it only horizontally or vertically.

Maintain line shape when moving or stretching elements

When elements are stretched or moved, Rational Rhapsody maintains the relationship between lines and boxes as much as possible to preserve diagram layout.

When moving or stretching more than one element at the same time, lines maintain their ratio to the other elements. That is, the line stretches and moves along with the other elements. This can be problematic for straight lines, which do not remain straight.

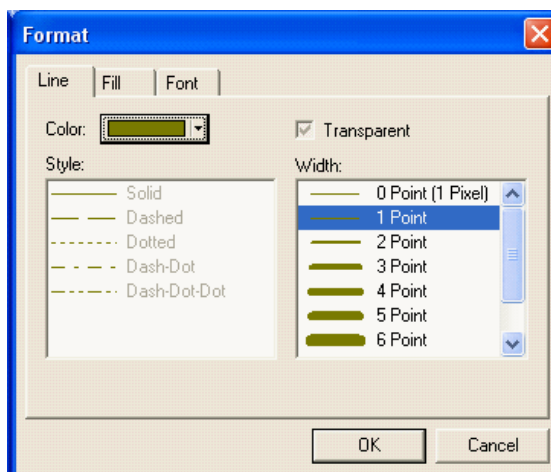
To maintain straight lines when moving or stretching boxes, move the boxes one at a time.

When moving or stretching only one box, lines that connect with vertical boundaries of the box maintain their Y-coordinate, and lines that connect with horizontal boundaries of the box maintain their X-coordinate. The coordinates change only when the box is moved to the extent that maintaining those coordinates is impossible. This way, lines that are straight remain straight whenever possible.

Change the format of a single element

To edit the format of a single element in a diagram, right-click the element and select **Format** or select **Edit > Format**. Alternatively, you can use the tools in the **Format** toolbar. For more information, see [Format text on diagrams](#).

The Format window lists the current line, fill, and font information for the selected element. The following figure shows the default line attributes for a class.

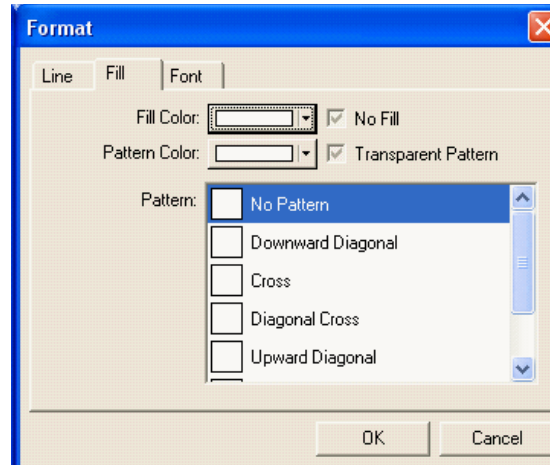


Use the **Line** tab of the window to change the line color, style, or width or the lines used to draw the element.

Note

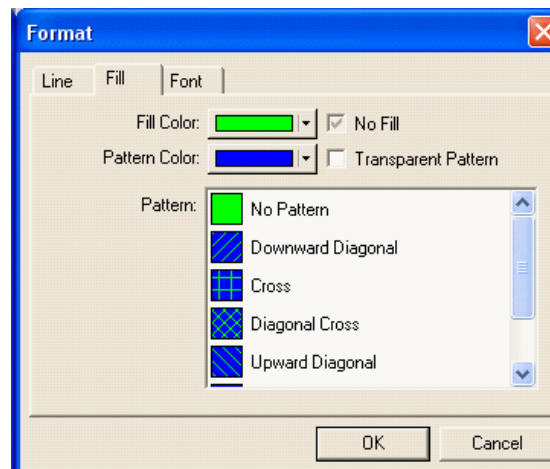
Line widths 0 and 1 are the same; however, you can specify a new line type (dashed, dotted, and so on) only for line width 0. Otherwise, the style options remain unavailable.

The **Fill** tab enables you to specify new fill attributes for the specified element, including the fill and pattern colors, and the pattern style. The following figure shows the **Fill** tab for a class.

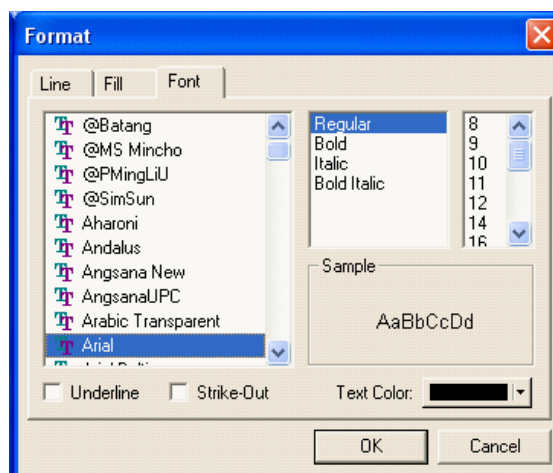


A preview of the pattern is displayed in the small box beside each pattern name.

For example, the following figure shows the preview of a fill color of green, a pattern color of blue, with the Transparent pattern check box cleared.



The **Font** tab enables you to specify new font attributes for the specified element, including the font size and type, color, and whether the text should be in italics or bold, or underlined. The following figure shows the **Font** tab for a class.



Note

These settings apply only to the specified element. Therefore, if you change the attributes for an individual class, any new classes will use the default attributes. For information on changing the attributes for a specific type of element (for example, all classes), see [Change the format of a single element](#).

Format text on diagrams

The **Format** toolbar provides tools that affect the display of text in your diagrams, such as font, size, color, and so on. In addition, you can access these options by selecting **Edit > Format > Change**. To display or hide this toolbar, choose **View > Toolbars > Format**.

The **Format** toolbar includes the following tools:

Tool Button	Name	Description
	Font Type	Specifies the font style ("type face") used for text. Use the drop-down list to select a different font.
	Text Size	Specifies the size used for text. Use the drop-down list to select a different size.
	Italic	Changes the selected text to italic font.
	Bold	Changes the selected text to boldface font.
	Left	Left-justifies the selected text. This tool is available only in fields that support RTF, including the Description for elements and annotation elements (comment, requirement, and constraint)
	Center	Centers the selected text. This tool is available only in fields that support RTF, including the Description for elements and annotation elements (comment, requirement, and constraint). This tool is available only in fields that support RTF, including the Description for elements and annotation elements (comment, requirement, and constraint)
	Right	Right-justifies the selected text. This tool is available only in fields that support RTF, including the Description for elements and annotation elements (comment, requirement, and constraint).
	Bullet	Creates a bulleted list. This tool is available when you are editing the Description for an element.
	Font Color	Specifies the color to use for the text or label of the selected element. For example, if you select a state and use this tool to change the color to red, the name of the selected state is displayed in red. This tool performs the same action as right-clicking an element and selecting Format .
	Line Color	Specifies the color to use for the selected line element. For example, if you select a state and use this tool to change the line color to blue, the text box for the state will be displayed in blue. This tool performs the same action as right-clicking an element and selecting Format .

Tool Button	Name	Description
	Fill Color	Specifies the color to use as fill color for the selected element. For example, if you select a state and use this tool to change the color to yellow, the selected state will be filled with yellow. This tool performs the same action as right-clicking an element and selecting Format .

Copying formatting from one element to another

Rational Rhapsody provides a Format Painter button  to copy formatting from one element to another element in the same diagram.

To copy formatting by using the Format Painter button:

1. In the diagram, click the element whose formatting you want to copy.
2. Click the Format Painter button in the **Standard** toolbar.
3. Click the element to which you would like to apply the copied formatting.

To copy formatting without having to click the Format Painter button each time:

1. In the diagram, click the element whose formatting you want to copy.
2. Click the Stamp Mode button in the **Layout** toolbar.
3. Click the Format Painter button
4. Click the elements to which you would like to apply the copied formatting.

Note

The Stamp Mode button is a toggle button. Rational Rhapsody will remain in “stamp mode” until you click the button a second time.

Changing the format of a metaclass

In addition to changing the format of an individual element, you can change the format of an entire metaclass. For example, you can specify styles for all classes, all actors, all associations, and so on.

To change the default settings for an entire metaclass:

1. Right-click the diagram and select **Format** to open the Format window.

Note: For diagrams only, the Format window has an **Apply to modified elements** check box. For projects, packages, and stereotypes, there is an **Apply to sub elements** check box instead.

2. In the **Select Meta-class to format** box, select the metaclass whose attributes you want to change, then click the **Format selected meta-class** button.

Note: The **Select Meta-class to format** box displays only the metaclasses that are relevant for the type of diagram.

3. Use the Format Properties window to select the new line, fill, and font attributes for the metaclass and click **OK**. (For more information about using the Format Properties window, see [Change the format of a single element](#)). The Preview box on the Format window displays how the element will look in the diagram.

4. Click one of the available buttons on the Format window:

- a. **Cancel** to discard all of your changes.
- b. **OK** to save your changes to the specified metaclasses.

Note: If **Apply to modified elements** is available when you click **OK**, all the existing elements in the selected scope (of the given metaclass) are changed to the specified format (in addition to any elements that are created later). If **Apply to modified elements** is unavailable, existing elements of the specified metaclass that have individual overrides are not changed, but new elements will use the new style by default.

For example, consider the case where all actors have white fill by default, but actor A has blue fill with yellow stripes. If you change the default fill color for all actors to be green, and **Apply to modified elements** is available, all the actors will have white fill. However, if you clear this check box, all the actors will have white fill, except for actor A, which will keep the blue fill and yellow stripes. All subsequently created actors will use white fill.

- c. **Reset** to remove all user-specified formats for the specified element and “rolls back” to the default values.

Note: If **Apply to modified elements** is available when you click **Reset**, Rational Rhapsody displays a confirmation window that asks whether you want to reset the default values for all of the modified subelements in the specified diagram or project. Click **Yes** to remove all overrides; otherwise, click **No** and clear this check box to reset specific metaclass styles.

- d. **Enforce** to force a subelement to use the locally modified style. Note that this affects only the styles that were explicitly specified; other formats remain unchanged. It is similar to the behavior specified in [Change the style scope](#).

Note: If **Apply to modified elements** is available when you click **Enforce**, Rational Rhapsody displays a confirmation window that asks whether you want to force the local style on all the elements in the specified diagram. Click **Yes** to remove all overrides; otherwise, click **No** and disable this check box to reset specific metaclass styles. Note that if you apply **Enforce** without enabling the **Apply** check box, nothing is changed.

Use **Edit > Format > Un-override** to remove overrides on boxes and line elements in diagrams.

Change the style scope

Where you open the Format window affects the scope of the formatting style:

- ◆ If you open the Format window from within a diagram, the format change applies only to the metaclass in the current diagram.

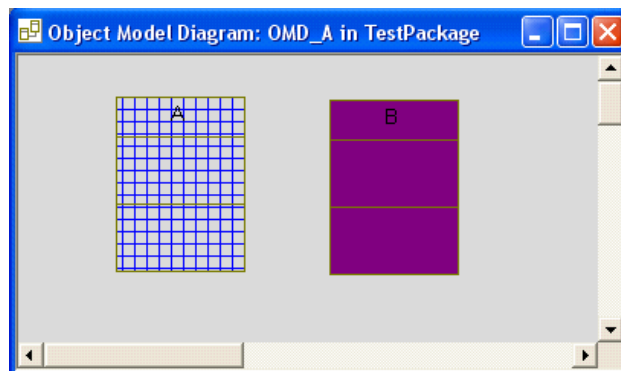
For example, if you change the format of states in the Tester statechart so they are filled with yellow, states in other statecharts will not be filled with yellow automatically.

- ◆ If you open the Format window at the project level (select the project node in the browser, right-click and select **Format**), the specified style is applied to that metaclass throughout the entire model.
- ◆ If you open the Format window by selecting an individual element in a diagram, the specified style will be applied to that element only.

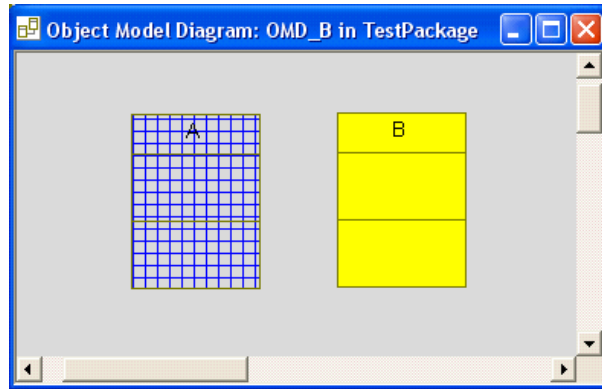
If you apply a style to an individual element and then copy it to a new diagram, it brings its style with it. Consider the following scenario:

- ◆ OMD_A uses purple fill for classes.
- ◆ Class A in OMD_A has the individual fill style of blue cross-hatching.
- ◆ Class B in OMD_A uses the default style for classes (purple fill).
- ◆ OMD_B uses yellow fill for classes.

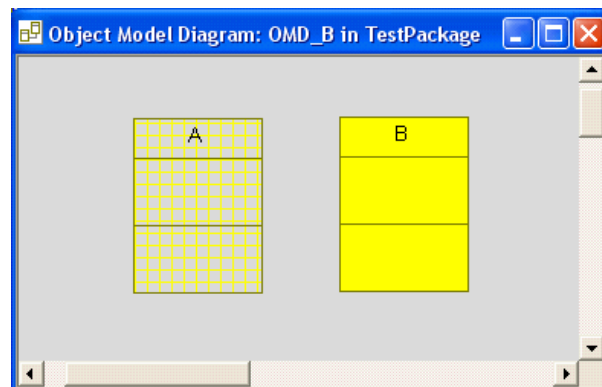
The following figure shows OMD_A.



If you copy classes A and B into OMD_B, class A keeps its individual style, but class B now uses the default local style (yellow fill). The following figure shows OMD_B.



To force the class to use the local style, click **Enforce**. In this example, class A is now forced to use yellow as its fill color. Note that the cross-hatching is still used (because there is no setting for cross-hatching in this OMD).



Making the format for an element the default formatting

After you have applied formatting to a diagram element, you can make the formatting for the element the default formatting for new elements of this type.

To make the formatting of an element the default formatting for elements of that type:

1. Right-click the element in the diagram and select **Make Default** (or select the element and choose **Edit > Make Default**). The Make Default window opens.
2. Select the characteristics to set as default. The available options are format, display options, and size.
3. Select the level at which you would like to set the defaults, for example, diagram level or package level.

Note

This option sets the default formatting for all new elements of the same type. For elements that already exist in the diagram, the default formatting will be applied unless the elements have been overridden. (This applies only to the formatting; the size of existing elements will not be changed, nor will the display options.)

Copy an element

There are two different ways in which elements can be copied and pasted in a diagram:

- ♦ **Simple Copy** where another representation of the element is created on the diagram canvas.
- ♦ **Copy with Model** where a new element is created in the model and pasted into the diagram. The new model has the exact same characteristics as the original element.

Simple copy

You can copy an element in one of three ways:

- ♦ Use the **Copy** and **Paste** commands in the Edit menu.
- ♦ Use the **Ctrl+Drag** method.
- ♦ Use the **Layout > Replicate**. Note that this applies to statecharts only.

To copy an element using the **Ctrl+Drag** method:

1. Select the element you want to copy.
2. Move the mouse pointer over the element.

3. Press **Ctrl**. A small box with a plus sign in it is displayed below and to the right of the pointer.
4. Click-and-drag the element. When you release the mouse button, a copy of the element displays in the new location.

Note

In statecharts, copying creates new elements. In OMDs and UCDs, these methods copy graphic elements but do not create new elements in the model.

Replicating

To copy using the **Replicate** command:

1. Select the element you want to copy.
2. Choose **Layout > Replicate** to open the Replicate window.
3. Enter the number of rows and columns you want the copied elements to use and the spacing between them.
4. Click **OK**. Rational Rhapsody displays the replicated elements in the diagram.

Copying with model

This refers to the ability to copy a diagram element such that when it is pasted into a diagram, an entirely new element is created, with all of the characteristics of the original element. For example, if you use this option to copy and paste a class, a new class will be created in the model and it will contain the same attributes and operations as the original class, the same associated diagrams (such as a statechart), and so on.

To create a new model element based on an existing diagram element:

1. Select the element to be copied.
2. From the main menu, select **Edit > Copy with Model**.
3. Navigate to the diagram where you would like to paste the new object.
4. From the main menu, select **Paste**. The new element will appear in the diagram as well as in the browser.
5. Rename the new item if wanted. The default name will be the name of the original element with the string “_copy” appended to it.

Arranging elements

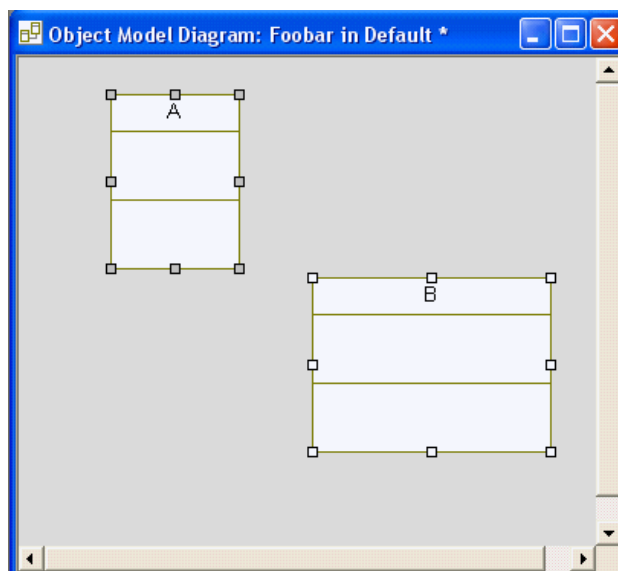
In addition to the grid and ruler tools (described in [Placing elements using the grid](#)), the **Layout** toolbar includes several tools that enable you to align elements in the drawing area.

To arrange elements:

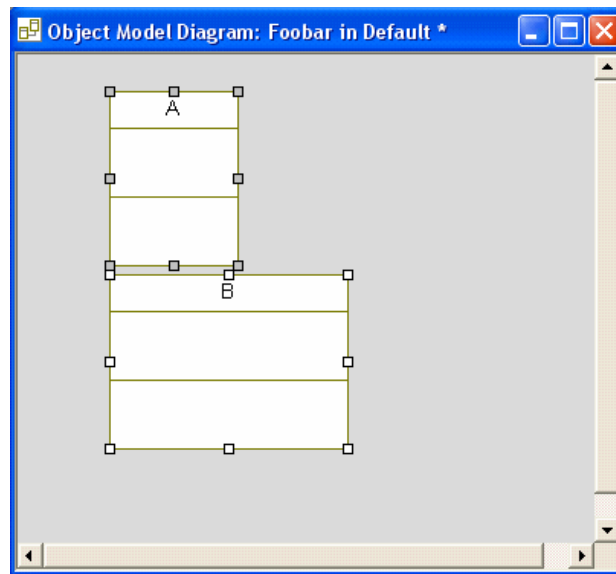
1. Select **View > Toolbars > Layout**. The **Layout** toolbar is displayed.
Note: You can dock the toolbar by clicking-and-dragging it to a window edge.
2. The tools that enable you to arrange elements are unavailable until you select an elements in the drawing area. In the diagram, select two or elements to arrange.
3. Select one of the layout tools. The elements are arranged according to the layout selected.

Note that the selection handles use different colors to show which element is used for the default alignment and sizing (the last element selected).

Consider the two classes in the following figure.



The selection handles on class A are gray, which denotes that Rational Rhapsody will use the values of class A for any alignments and sizing. Therefore, if you align the left sides of class A and B, class A stays where it is and class B moves under it, as shown in the following figure.



Layout toolbar

The **Layout** toolbar provides quick access to tools that help you with the layout of elements in your diagram, including a grid, page breaks, rulers, and so on. To display or hide this toolbar, choose **View > Toolbars > Layout**.

The **Layout** toolbar includes the following tools:

Tool Button	Name	Description
	Toggle Grid	Displays or hides the grid in the drawing area. This is equivalent to selecting Layout > Grid > Show Grid .
	Snap to Grid	Automatically aligns new elements to the closest grid points. This is equivalent to selecting Layout > Grid > Snap to Grid .
	Toggle Rulers	Displays or hides the rulers in the drawing area. This is equivalent to selecting Layout > Show Rulers .
	Toggle Page Breaks	Displays or hides the page breaks in your diagram. This is equivalent to selecting Layout > View Page Breaks . Page boundaries are denoted by dashed lines.
	Stamp Mode	Turns repetitive drawing mode on or off.
	Align Top	Aligns the selected elements to the top of the element with the gray selection handles. This is equivalent to selecting Layout > Align > Top .
	Align Middle	Aligns the selected elements to the middle (along the horizontal) of the element with the gray selection handles. This is equivalent to selecting Layout > Align > Middle .
	Align Bottom	Aligns the selected elements to the bottom of the element with the gray selection handles. This is equivalent to selecting Layout > Align > Bottom .
	Align Left	Aligns the selected elements to the left side of the element with the gray selection handles. This is equivalent to selecting Layout > Align > Left .
	Align Center	Aligns the elements so they are aligned to the center, vertical line of the element with the gray selection handles. This is equivalent to selecting Layout > Align > Center .
	Align Right	Aligns the selected elements to the right side of the element with the gray selection handles. This is equivalent to selecting Layout > Align > Right .
	Same width	Resizes all the selected elements so they are the same width as the element with the gray selection handles. This is equivalent to selecting Layout > Make Same Size > Same Width .

Tool Button	Name	Description
	Same height	Resizes all the selected elements so they are the same height as the element with the gray selection boxes. This is equivalent to selecting Layout > Make Same Size > Same Height .
	Same size	Resizes all the selected elements so they are the same size as the element with the gray selection boxes. This is equivalent to selecting Layout > Make Same Size > Same Size .
	Space across	Spaces the selected elements so they are equidistant (across) from the element with the gray selection handles. This is equivalent to selecting Layout > Space Evenly > Space Across .
	Space down	Spaces the selected elements so they are equidistant (down) from the element with the gray selection handles. This is equivalent to selecting Layout > Space Evenly > Space Down .

Removing an element from the view

To remove an element from the view but not from the model:

1. Select the element to be removed.
2. Press the **Delete** key. The element is removed from the view.

Alternatively, right-click the element in the diagram and select **Remove from View**.

Deleting an element from the model

To delete an element from both the view and the model:

1. Select the element to be deleted.
2. Click the **Delete** tool or press **Ctrl+Delete**. The element is deleted from both the model and the view.

Alternatively, right-click the element in the diagram and select **Delete from Model**.

Editing text

To edit text:

1. Double-click any text to highlight it.
2. Edit the selection.
3. To add another line of text, press **Enter**; press **Ctrl+Enter** to end the edit.

Note that both the right mouse button and the Esc key cancel the edit.

Alternatively, for some features you can right-click the element in the drawing area and select **Edit Text**.

Display compartments

One of the display options available for diagram elements is the display of contained items in visual compartments. This option is available for the following items:

- ◆ Classes
- ◆ Objects
- ◆ Files

For the above elements, the following items can be displayed in compartments where applicable:

- ◆ Constraints
- ◆ Tags (both local and on the element's stereotype)
- ◆ Ports
- ◆ Parts

Selecting items to display

To select the items that should be displayed:

1. In the Display Options window, click **Compartments**.
2. In the window that is displayed, use the arrows to select the items that should be displayed, and the order in which they should be displayed.
3. Click **OK**.

In each compartment, individual items are displayed with an icon indicating the type of sub-element. If the text is too long to display, an ellipsis is displayed. When this ellipsis is clicked, you can view/edit the full text.

Note

For constraints, the content of the specification field is displayed.

The name of items can be edited in the compartments but items cannot be added or deleted.

It is not possible to modify the order in which individual items are displayed within each compartment.

User-defined terms are displayed in the same compartment as the item on which they are based, however, the icon indicates that this is a user-defined term.

Note

While this feature allows you to display/hide attributes and operations, it does not replace the attribute and operation tabs, which allow more precise display options, such as the display of only a subset of defined attributes and operations.

Display stereotype of items in list

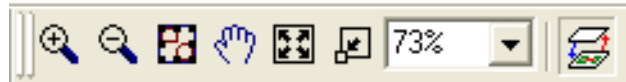
For elements listed in compartments, Rational Rhapsody provides the option of displaying the name of stereotypes applied to the element alongside the name of the element.

To display elements' stereotypes in the compartment list, use these guidelines:

- ♦ At the diagram level or higher, modify the property `General::Graphics::ShowStereotypes`. The possible values for this property are `No`, `Prefix`, `Suffix`.
- ♦ The default setting for this property is `Prefix`.
- ♦ This property applies to all of the compartments that can be displayed.

Zoom

There are several zoom tools available on the **Zoom** toolbar, shown in the following figure.



Alternatively, you can choose **View > Zoom/Pan > 50%**, **View > Zoom/Pan > 75%**, **View > Zoom/Pan > 100%**, or **View > Zoom/Pan > 200%**; or use the zoom options available in the menu in the drawing area.

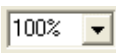
To prevent elements from being resized when you resize their parent (for example, classes contained in a composite class), press and hold the **Alt** key while you click-and-drag with the mouse.

Zoom toolbar

The **Zoom** toolbar contains the zoom tools you can use with all the different diagram types. These tools are also available in **View > Zoom** or **View > Zoom/Pan**. To display or hide this toolbar, choose **View > Toolbars > Zoom**.

The **Zoom** toolbar includes the following tools:

Tool Button	Name	Description
	Zoom In	Zooms in on a diagram. Click this button and click in a graphic editor window to increase the view by 25%.
	Zoom Out	Zooms out on a diagram. Click this button and click in a graphic editor window to decrease the view by 25%.
	Zoom to Selection	Select an element in a diagram and click this button to zoom into the selected section of the diagram. Alternately, you can click Zoom In and hold down the left mouse button to draw a selection box around the part of the diagram you want to zoom in on.
	Pan	Moves the diagram in the drawing area so you can see portions of the diagram that do not fit in the current viewing area.
	Zoom to Fit	Resizes the active diagram to fit within the graphic editor window.

Tool Button	Name	Description
	Undo Zoom	Reverses the last zoom action.
	Scale Percentage	The options on this drop-down list resize the active diagram scale by the selected percentage.
	Specification/Structured View	Displays either the specification or structured view of the active diagram.

Zooming in and zooming out

To zoom in or out on a diagram:

1. Click the **Zoom In** or **Zoom Out** button (or choose **View > Zoom > Zoom In** or **View > Zoom > Zoom Out**).
2. Move the cursor over the diagram. The cursor displays as a magnifying glass with either a plus or minus sign in it.
3. Click the diagram to enlarge or shrink it by 25%, depending on which tool you selected.

There are two ways to zoom in on a portion of a diagram:

- ◆ Click the **Zoom In** button, then hold down the left mouse button to draw a selection box around the part of the diagram you want to zoom in on.
- ◆ Select an element in the diagram, then click **Zoom to Selection** to enlarge the selected element so it takes up the entire drawing area.

Note

You remain in zoom mode until you select another tool from the toolbar.

Refreshing the display

If at any time the screen becomes difficult to read, you can refresh it by pressing F5 or selecting **View > Refresh**.

Scaling a diagram

To scale a diagram to a certain percentage, use the drop-down scale box. You can set the diagram scaling to a value between 10% and 500%.

Alternatively, select **View > Zoom/Pan**, then select the percentage used to scale the diagram.

To scale the diagram so the entire diagram is visible in the current window, click the **Zoom to Fit** button (or **View > Zoom/Pan > Zoom to Fit**). When you click the button, the diagram is resized to fit in the current window. This button performs the same command as **View > Zoom to Fit**.

Panning a diagram

Click the **Pan** button to move the diagram in the drawing area so you can see portions of the diagram that do not fit in the current viewing area.

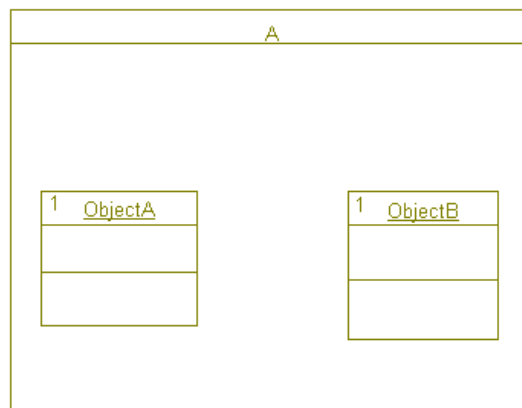
Undoing a zoom

To undo the last zoom, click the **Undo Zoom** button, or select **View > Zoom/Pan > Undo Zoom/Pan**.

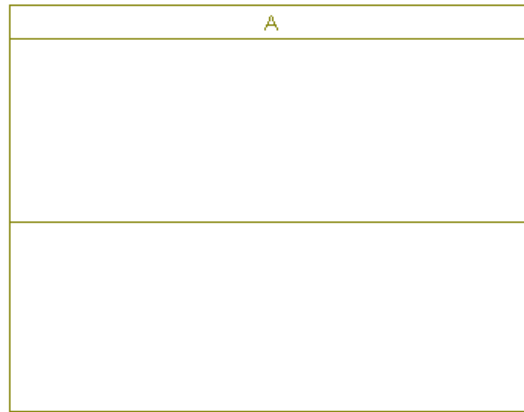
Specifying the specification or structured view

To show the specification or structured view of a diagram, click the **Specification/Structured View** button.

For example, suppose you have a structured (composite) class A that contains the parts `ObjectA` and `ObjectB`. The structured view looks like the following figure.



The specification view looks like the following figure.



In addition to toggling between the two views, any new classes or objects created with the selected mode will use that mode.

Note that if there is a mix of structured and specification elements, the button is disabled.

The Bird's Eye (diagram navigator)

The Bird's Eye (diagram navigator) provides a high-level view of the diagram that is currently displayed. This can be very useful when dealing with very large diagrams, allowing you to view specific areas of the diagram in the drawing area, while, at the same time, maintaining a view of the diagram in its entirety.

The Bird's Eye contains a depiction of the entire diagram being viewed, and a rectangle viewport that indicates which portion of the diagram is currently visible in the drawing area.

Showing and hiding the Bird's Eye window

To show/hide the Bird's Eye window, do one of the following actions:

- ♦ Choose **View > Bird's Eye**.
- ♦ Click the Bird's Eye button  on the **Windows** toolbar.
- ♦ Use the keyboard shortcut, **Alt+5**.
- ♦ Right-click the diagram in the drawing area, and then select **Bird's Eye**.

Navigating to a specific area of a diagram

To use the Bird's Eye to move to a specific area of a diagram, do one of the following actions:

- ♦ Drag the viewport to the area you would like to view.
- ♦ Click and draw a “new” viewport in the Bird's Eye window over the area you would like to view.

Using the Bird's Eye to enlarge and shrink the visible area

To use the Bird's Eye to enlarge/shrink the visible area of the diagram, drag an edge or corner of the viewport to enlarge/shrink the viewport.

Enlarging the viewport has the same effect as zooming out in the drawing area. Shrinking the viewport has the same effect as zooming in the drawing area.

Note

When you drag an edge of the viewport, the viewport size will always change in both dimensions, maintaining the height/width ratio.

By default, the viewport will grow in the direction of the edge selected. If you hold down

Ctrl while dragging, however, the viewport will grow in the direction of the opposite edge as well in order to maintain the current center point of the diagram.

Scrolling and zooming in drawing area

If the scroll bars are used to change the visible area of the diagram in the drawing area, the viewport will move accordingly in the Bird's Eye window.

If the zoom level is changed in the drawing area, the size of the viewport will change accordingly in the Bird's Eye window.

Changing the appearance of the viewport

You can modify the appearance of the viewport rectangle in the Bird's Eye window. Display characteristics that can be modified include line color, line style, line width, fill color, and fill pattern.

To display the viewport appearance window: Right-click anywhere in the Bird's Eye window.

General characteristics of the Bird's Eye window

- ♦ The Bird's Eye window is used only for changing the viewable area of a diagram. You cannot modify any diagram elements in the Bird's Eye window.
- ♦ The size and position of the Bird's Eye window is saved in the Rational Rhapsody workspace.
- ♦ The Bird's Eye window can be resized, and it can float or be docked. The docking-related options are accessed by right-clicking the borders of the window (but not the title bar).
- ♦ The black dotted line in the Bird's Eye window represents the diagram's drawing canvas. When the canvas size is changed in the drawing area, this dotted line changes accordingly. Depending on the size of the Bird's Eye window, there might be some whitespace to the right of and below the dotted line.

Complete relations

The **Layout > Complete Relations** menu completes relation lines in diagrams. For example, you can define relations in the browser, draw the participating classes in an OMD, then select **Complete Relations** to speed up the drawing of the relation lines.

- ♦ **Layout > Complete Relations > All** completes all the relation lines
- ♦ **Layout > Complete Relations > Selected to All** completes relation lines only for relations originating in the selected classes
- ♦ **Layout > Complete Relations > Among Selected** completes relation lines only for relations existing between the selected classes

Use IntelliVisor

The IntelliVisor feature offers intelligent suggestions based on what you are doing to reduce:

- ♦ The number of steps required to complete a task
- ♦ The amount of time spent changing between different views (browser, graphics editor, code editor, and so on), giving more time to actually complete the task
- ♦ The time spent in the “build and run” cycle because of fewer compilation problems resulting from wrong type usage, misspellings, and so on

The suggestions offered by IntelliVisor depend on the current context. The context is a model element (usually a class or a package) in which the IntelliVisor is activated. The IntelliVisor contents is defined by the scope of the context. For example, the context can be class `MyClass`; the list contents will be all the methods, attributes, and relations, including superclasses and interfaces implemented by `MyClass`, and all the global methods defined in the package containing the class.

The property `IntelliVisor::General::ActivateOnGe` specifies whether to enable IntelliVisor in the Rational Rhapsody graphics editors. By default, IntelliVisor is available.

Activating IntelliVisor

When you press **Ctrl+Space** in the graphic editor, Rational Rhapsody displays a list box with information from which to choose. You can navigate in this list box using either the arrow keys or the mouse. When you select an item from the list of suggestions and press **Enter**, that text is placed in the text box.

To dismiss the IntelliVisor list box, do one of the following actions:

- ♦ Press one of the following keys:
 - **Esc**
 - **Enter**
- ♦ Double-click the mouse button.
- ♦ Change the window focus.
- ♦ Press **space/Alt**.
- ♦ Click outside of the text box.

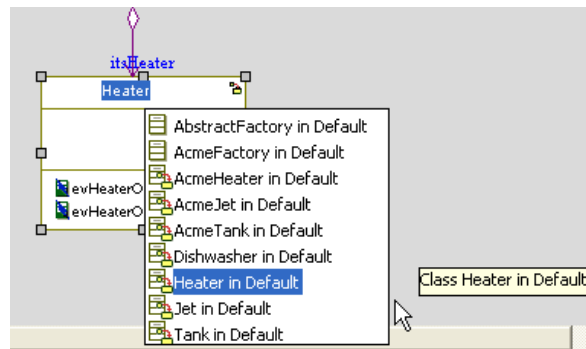
IntelliVisor information

When you are using a graphics editor, IntelliVisor can simplify your tasks by offering shortcuts to similar model elements.

For example, if you are drawing a class in an OMD or structure diagram and start IntelliVisor, the list box contains the default name of the new class and all the classes that already exist in the model. If you highlight one of the classes in the list, IntelliVisor displays summary information available for that element, including:

- ◆ Its type, name, and parent information
- ◆ Its stereotype, if one exists
- ◆ The first few lines of the description for the element, if one exists

The following figure shows information displayed by IntelliVisor.

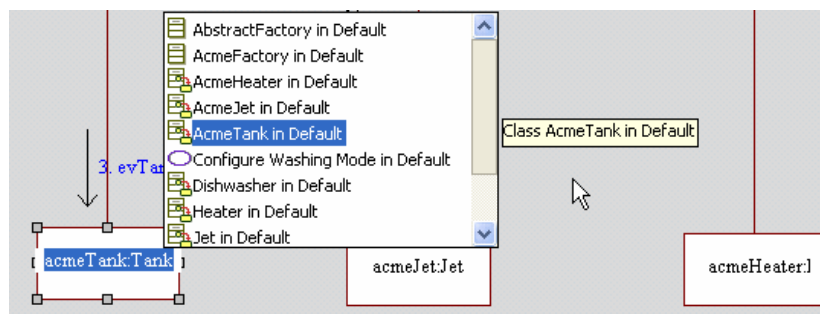


To replace the new class with an existing class, simply highlight the class in the list box and double left-click. IntelliVisor replaces the new class with the specified class.

In addition to classes, IntelliVisor can be opened in OMDs when you are drawing actors and packages.

Collaboration diagrams

If you open IntelliVisor on an object or multi-object, the list box contains all the classes in the current project. For example:

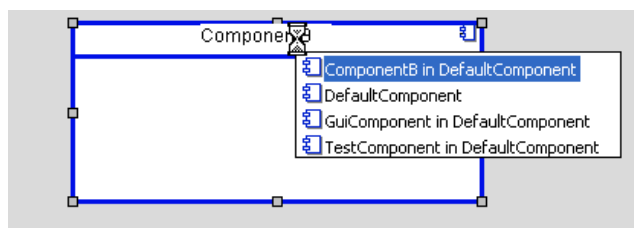


If you apply a selection from the list, IntelliVisor replaces the part after the role name. For example, `AcmeTank : Tank` will become `AcmeTank : Jet` if you select the `Jet` class in the list box.

In addition to classes, IntelliVisor can be started in collaboration diagrams when you are drawing actors.

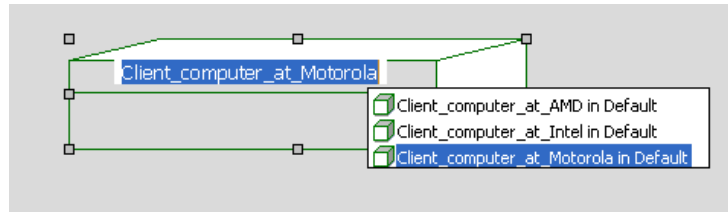
Component diagrams

If you start IntelliVisor for a component in a component diagram, the list box contains all the components in the model. For example:



Deployment diagrams

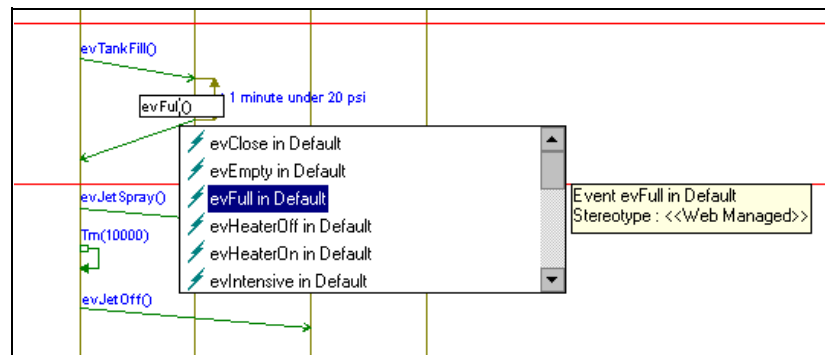
If you start IntelliVisor on a node in a deployment diagram, the list box contains all the nodes defined in the model. For example:



Note that you cannot activate component instances in IntelliVisor.

Sequence diagrams

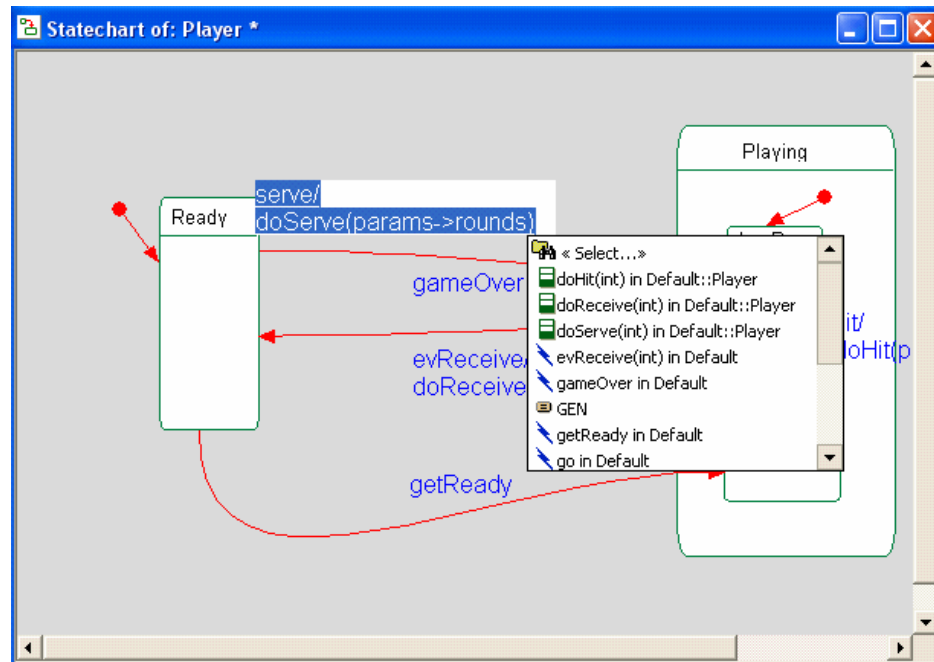
If you start IntelliVisor within a sequence diagram, the list box contains the events, operations, and triggered operations consumed by the target class. If there are base classes that consume events, operations, and triggered operations, they are included in the list.



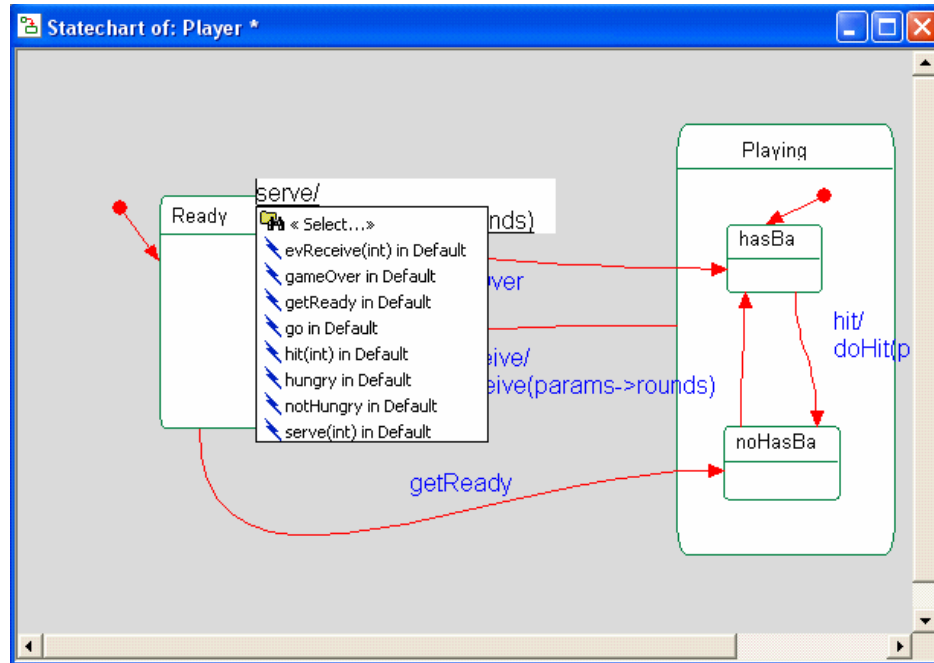
Note the following information:

- ◆ If you select a constructor line, IntelliVisor displays the list of constructors.
- ◆ The instance line should be associated with a part class, or the IntelliVisor list box will be empty.

If you start IntelliVisor inside a activity flow trigger in a statechart or activity diagram before the “/” or “[” symbol, the list box contains all the events that can be consumed by the class, as in this example.



If you start IntelliVisor after the “/” or “[” symbol, the list box contains the default class content. For example:

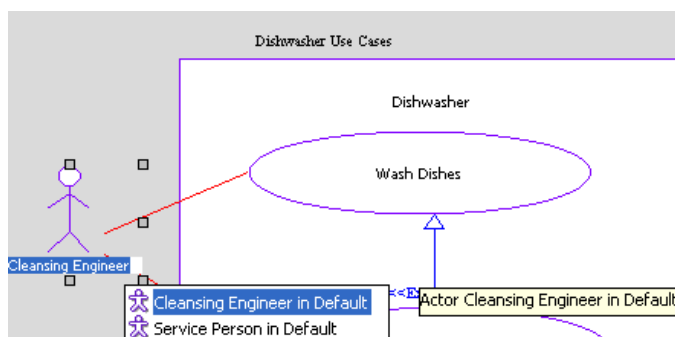


In addition, you can start IntelliVisor in activity diagrams to help you perform the following tasks:

- ◆ Write initialization code for actions.
- ◆ Edit the code for an activity.

Use case diagrams

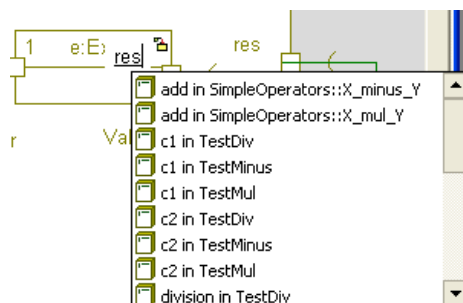
If you apply IntelliVisor on an actor, the list box contains all the actors defined in the project.



Similarly, if you start IntelliVisor on a use case, the list box contains all the use cases defined in the project.

Structure diagrams

If you start IntelliVisor for an object in a structure diagram, the list box contains all the objects defined in the project. For example:



Similarly, if you start IntelliVisor on a composite class, the list box contains all the composite classes defined in the project.

Customizations for Rational Rhapsody

You can customize Rational Rhapsody in the following ways:

- ◆ Add helper applications (also known as helpers). Helpers are custom programs that you attach to Rational Rhapsody to extend its functionality. They can be either external programs (executables) or Visual Basic for Applications (VBA) macros that typically use the Rational Rhapsody COM API. They connect to a Rational Rhapsody object via the `GetObject()` COM service. See [Helpers](#).
- ◆ Use Visual Basic for Applications (VBA), an OEM version of Microsoft Visual Basic to develop automation and extensibility scripts that interact with the tool repository that provides a full complement of user interface components (“forms”). See [Visual Basic for applications](#).
- ◆ Create a customized profile to use in the models for your company. A customized profile has the following advantages:
 - Contains terminology specific to your company
 - Forces adherence to special requirements or industry standards
 - Can be reused in other models to simplify and standardize development effortsSee [Creating a customized profile](#).
- ◆ Add new element types to your models. See [Adding new element types](#).
- ◆ Create a customized diagram. See [Creating a customized diagram](#).
- ◆ Customize the Add New menu. See [Customize the Add New menu](#).
- ◆ Create a Rational Rhapsody plug-in. See [Creating a Rational Rhapsody plug-in](#).

Helpers

Helpers are custom programs that you attach to Rational Rhapsody to extend its functionality. They can be either external programs (executables) or Visual Basic for Applications (VBA) macros that typically use the Rational Rhapsody COM API. They connect to a Rational Rhapsody object via the `GetObject()` COM service.

You can add your helper to the Rational Rhapsody **Tools** menu. To open the Helpers window, open a Rational Rhapsody project and choose **Tools > Customize**.

The following tools are available on the Helpers window:



Click the New icon to create a new helper menu item.



Click the Delete icon to delete a helper menu item.



Click the Move Up icon to move up the helper item on the Rational Rhapsody **Tools** menu.



Click the Move Down icon to move down the helper item on the Rational Rhapsody **Tools** menu.

Use the following boxes on the Helpers window to identify and apply your helper application:.

Command	Browse to the path to your helper application.
Arguments	Optionally, add a binding for a parameter that resolves to a run-time instance.
Initial directory	For an external program helper only, browse to the path of the default directory for the helper application.
Applicable to	From the drop-down list, select the model elements to associate with the helper.
Project type	From the drop-down list, select one or more profiles (for example, FunctionalC, DoDAF, SysML) to which the helper applies.
Helper trigger	From the drop-down list, select the action that will trigger this helper.

At the bottom of the window, identify if your helper application is an external program helper or VBA macro helper.

List of helper triggers

The following helper triggers are available:

- ◆ After Project New
- ◆ After Project Open
- ◆ Before Project Save
- ◆ After Project Save
- ◆ Before Check Model
- ◆ Before Code Generation
- ◆ After Roundtrip
- ◆ After Change To
- ◆ After Add Element

Creating a link to a helper application

To create a link to a helper:

1. Open a Rational Rhapsody project and choose **Tools > Customize** to open the Helpers window.
2. Click the New icon  to add a blank line for a new menu item in the **Menu content** box.
3. In the blank field, type the name of the new menu item (for example, My New Command).

- ♦ To specify a submenu structure, enter the menu text with a backward slash (\), for example, External\My New Command1.

Note that you can have more than one item in your submenu structure. You can create another link to a helper and specify it as, for example, External\My New Command2.

- ♦ To make a shortcut key, add an ampersand character before a letter in the name. For example, &My makes the letter M a menu shortcut. You can press **Alt+M** to open this particular helper application once it has been created. Be sure to not use a letter that is already used as a shortcut key on the **Tools** menu or the pop-up menu for the associated model element.
4. Specify the applicable helper parameters:
 - ♦ In the **Command** box, enter the command that the menu item should start, such as E:\mks\mksnt\cp.exe or click its Ellipsis button  to browse to the location of the application.
 - ♦ Optionally, in the **Arguments** box, enter any arguments for the command.
 - ♦ Optionally, in the **Initial Directory** box, enter an initial default directory for the program. This applies only to external programs.
 - ♦ In the **Applicable To** list, specify which model elements to associate with the new command.

If you do not specify a value for this field, the menu command for this helper application can be added to the **Tools** menu depending on what you do in Step 6.
 - ♦ In the **Project Type** list, select a **project profile**, as defined in [Creating a project](#).

If leave this box blank, it uses as the default the profile of the current project you have opened.
 - ♦ In the **Helper Trigger** list, select the actions that triggers the new command.

5. Specify the helper type:

- ♦ Select the **External program** radio button if the new command is an external program, such as Microsoft Notepad.

Select the **Wait for completion** check box if you want the external program to complete running before you can continue to work in Rational Rhapsody.

- ♦ Select the **VBA macro** radio button if the new command is a VBA macro and is defined in the <Project>.vba file. See [Adding a VBA macro](#).

6. Depending on what you decided for the **Applicable To** list:

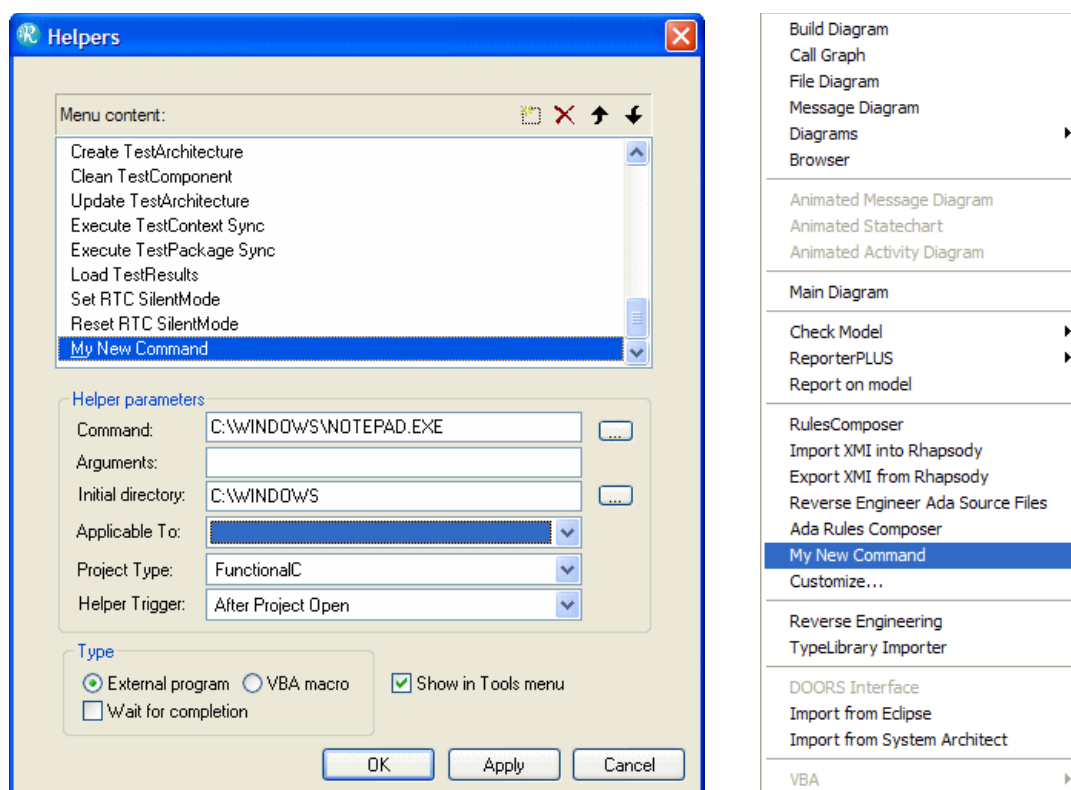
- ♦ If you did not specify an applicable element for the command, verify that the **Show in Tools menu** check box is selected. This means the new menu command for your link to a helper application displays on the **Tools** menu. If you clear this check box, there is no menu command for it on the **Tools** menu, though the link to the helper application still works once the trigger for this command is started.
- ♦ If you specified an applicable element for the command, verify that the **Show in Pop-up menu** check box is selected. This means the new command displays in the menu for the specified model element. If you clear this check box, there is no menu command for it on the pop-up menu for the specified model element, though the link to the helper application still works once this command is started.

7. Click **OK** to apply your changes and close the window. (You can click the **Apply** button if you want to save your changes but keep the window open to continue working with it.)

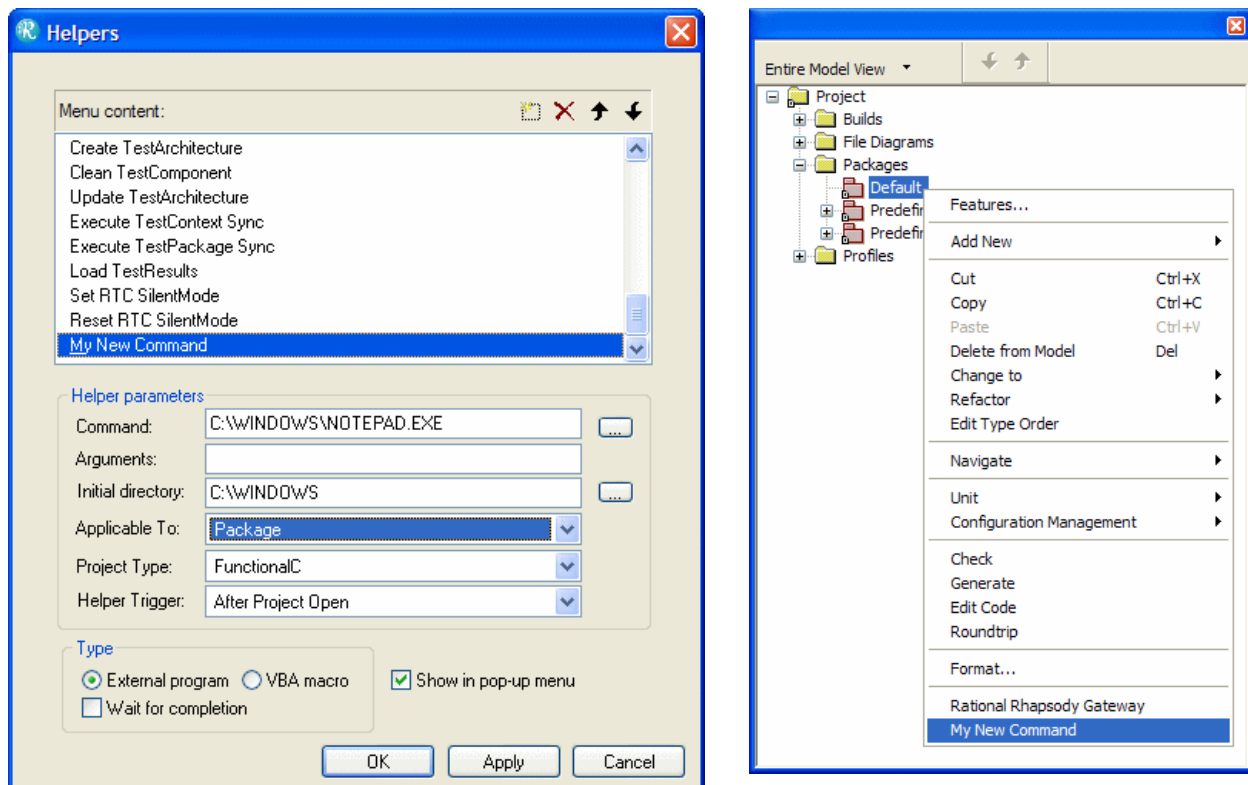
Once you save and close the Helpers window, the link to the helper application you just created is immediately available if the current project is within the parameters that you set for the link. For example, if the Rational Rhapsody project you currently have open uses the FunctionalC profile and you created the My New Command helper application for this profile, then this link to the helper application is immediately available. However, if you specified the DoDAF profile (as selected in the **Project Type** drop-down list) for the My New Command link, then it will not work in your current project.

Examples of helper application menu commands

If you did not specify an applicable model element and you selected the **Show in Tools menu** check box on the Helpers window, as shown in the following figure on the left, your helper application menu command is added to the **Tools** menu, as shown on the right.



If you specified an applicable model element for your command and you selected the **Show in pop-up menu** check box on the Helpers window, as shown in the following figure on the left, you can right-click either the model element on the Rational Rhapsody browser to access the menu command for the helper application, as shown on the right, or the applicable element in a graphical editor. Note that the command now does not show on the **Tools** menu.



Using a .hep file to link to helper applications

You can use a .hep file to group links to helper applications that help achieve the purpose of a Rational Rhapsody profile. Helper applications are custom programs created by you or a third-party that you can link to within the Rational Rhapsody product. Helper applications add functionality that a profile might not have, such as the ability to query a model or write to a project. For more information about profiles, see [Profiles](#).

Using a .hep file is ideal for teams where all members should use the same helper applications. When your project profile and its corresponding .hep file are loaded by reference, when your team members update, they get the latest versions of these files. A .hep file is considered part of a profile and is treated as such.

To see sample .hep files, you can look at the ones provided with the Rational Rhapsody product for certain profiles, such as AUTOSAR, DoDAF, MODAF, NetCentric, and Harmony. For example, <Rational Rhapsody installation path>\Share\Profiles\DoDAF contains **DoDAF.sbs** (the profile file) and **DoDAF.hep**. Java users might want to look at the sample .hep files for the AUTOSAR and NetCentric profiles.

If sharing links to helper applications is not an issue, or perhaps it is company policy that everyone have access to the same helper applications for all Rational Rhapsody projects, then adding links to helper application in the rhapsody.ini file might suffice.

However, you might find using a .hep file more convenient:

- ◆ Easier maintenance. The rhapsody.ini file is typically overwritten when you get a new version of Rational Rhapsody. Since the name of a .hep file must correspond with the name of a profile, there is less likelihood of the .hep file being overwritten.
- ◆ Less clutter on your list of menu commands. The links to help applications can appear as menu commands on the **Tools** menu. Typically, many people work on various projects that might use different models (and profiles) and different helper applications. Using the rhapsody.ini file to store all your links to helper applications might cause clutter on your list of menu commands. You can use a different .hep file for each profile so that you only see the helpers needed for your project.

To use a .hep file to link to helper applications:

Note

The method described here is the typical way to link to a helper application. If you prefer to not use **Tools > Customize** (which opens the Helpers window), you should review the options on the Helpers window to familiarize yourself with the available options and their syntax. For example, look at the **Helper Trigger** box for the list of available triggers and notice how they are spelled out and capitalized. For more information about the Helpers window, see [Helpers](#).

1. Open Rational Rhapsody and choose **Tools > Customize** to create one or more links to helper applications. See [Creating a link to a helper application](#).

The code for your links is added to the rhapsody.ini file.

2. Close Rational Rhapsody.
3. Open the rhapsody.ini file and from the [Helpers] section of the file, copy the code for your help application. The following example show helper application code that was added to the rhapsody.ini file:

```
[Helpers]
...
name30=AutoCommand45
command30=C:\WINDOWS\notepad.exe
arguments30=
initialDir30=C:\WINDOWS
JavaMainClass30=
JavaClassPath30=
JavaLibPath30=
isVisible30=1
isMacro30=0
isPlugin30=0
isSynced30=0
UsingJava30=0
applicableTo30=
applicableToProfile30=Auto2009
helperTriggers30=After Project Open
isPluginCommand30=0
```

Note: Each section of link code starts with name##.

4. Open your .hep file and paste the code for the link to a helper application (what you copied in the previous step).

Your .hep file must have the same name as the name of the profile for your Rational Rhapsody project. For example, if the profile for your Rational Rhapsody project is called Auto2009, your .hep file must be called Auto2009.hep. In addition, both the profile and the .hep file must reside in the same folder.

5. In the rhapsody.ini file, delete the code that you copied in Step 3. The code to link to a helper application should only reside in the .hep file when you are using a .hep file.

6. Open Rational Rhapsody and open your model.
7. Load the applicable profile by reference. This is the profile that has the corresponding .hep file; see Step 4.
8. Test to make sure your link to a helper application works as expected.
For example, if a link is suppose to open a helper application after you open a model (helperTriggers30=After Project Open, as shown in the sample code in Step 3), make sure that happens.

Note

You can use the `General::Model::HelpersFile` property to associate a .hep file with a model.

Note that if you specify a .hep file using this property, Rational Rhapsody will not recognize the helper applications defined in the profile-specific .hep file if one is provided for the profile you are using.

Modifying a link to a helper application

To modify a link to a helper application:

1. With a project open in Rational Rhapsody, choose **Tools > Customize** to open the Helpers window.
2. If you want to edit the name of the helper application, double-click it in the **Menu content** box and make your changes.
3. Make other changes on the Helpers window as you want. For an explanation of the controls on the Helpers window, see [Creating a link to a helper application](#).

Modifying a .hep file

To modify a link to a .hep file by modifying the applicable .hep file:

1. Close the Rational Rhapsody model.
2. Open the .hep file in a text editor (such as Microsoft Notepad) and make your changes.
3. Save your changes to the .hep file.
4. Open the Rational Rhapsody model.
5. Test to make sure your changes work as expected.

Adding a VBA macro

The Helpers window can also be used to add a VBA macro. To add a VBA macro:

1. With a project open in Rational Rhapsody, choose **Tools > Customize** to open the Helpers window.
2. Click the New icon  to add a blank line for a new VBA macro menu command in the **Menu content** box.
3. In the blank field, type the name of the new menu item (for example, My VBA Command).
4. Select the **VBA macro** radio button as the helper type. The Helpers window lists VBA-specific options.
5. Specify the applicable helper parameters:

- ◆ In the **Module** box, enter the name of the VBA module.
- ◆ In the **Macro name** box, enter the name of the VBA macro.
- ◆ In the **Applicable To** list, specify which model elements to associate with the new command.

If you do not specify a value for this field, the menu command for this link to a helper application might be added to the **Tools** menu depending on what you do in Step 6.

- ◆ In the **Project Type** list, select a project profile, as defined in [Creating a project](#).

If leave this box blank, it uses as the default the profile of the current project you have opened.

- ◆ In the **Helper Trigger** list, select the actions that triggers the new command.

6. Depending on what you decided for the **Applicable To** list:

- ◆ If you did not specify an applicable model element for the command, verify that the **Show in Tools menu** check box is selected. This means the new menu command for your link to a helper application displays on the **Tools** menu. If you clear this check box, there is no menu command for it on the **Tools** menu, though the link to the helper application still works once the command is started.
- ◆ If you specified an applicable model element for the command, verify that the **Show in Tools menu** check box is selected. This means the new command displays in the menu for the specified model element. If you clear this check box, there is no menu command for it on the pop-up menu for the specified model element, though the link to the helper application still works once the command is started.

For examples, see [Examples of helper application menu commands](#).

7. Click **OK**.

The helper application you just created is immediately available if the current project is within the parameters that you set for the helper application. For example, if the Rational Rhapsody project you currently have open uses the FunctionalC profile and you created the My New Command helper application for this profile, then this helper application is immediately available. However, if you specified the DoDAF profile (as selected in the **Project Type** list) for the My New Command helper application, then it will not work in your current project.

Note

It is your responsibility to add code to your VBA macro to verify that the selected object is actually the core object for your command. The COM command to get the selected element is `getSelectedElement()`.

Visual Basic for applications

Visual Basic for Applications (VBA), an OEM version of Microsoft Visual Basic, is integrated as an automation engine into the Microsoft Office family and for use in all Microsoft tools. It provides a complete application development environment based on Visual Basic.

With VBA, you can develop automation and extensibility scripts that interact with the tool repository that provides a full complement of user interface components (“forms”). There is virtually no limit to application extensibility that can be achieved using VBA. Conceptually, it would be possible to completely transform the hosting application into another application using VBA extensibility.

VBA and Rational Rhapsody

The basic interaction between VBA and Rational Rhapsody is facilitated through the Rational Rhapsody COM API, similar to the way Visual Basic interacts with Rhapsody using API external programs. Rhapsody exports a set of COM interfaces that represent its metamodel objects, as well as its application operational functions. Through the COM interfaces, a VBA macro can easily access all the Rhapsody objects and manipulate them.

The VBA project file

A VBA project is a file container for other files and components that you use in Visual Basic to build an application. After all the components have been assembled in a project and code written for it, you can compile the project into an executable file.

Each Rational Rhapsody project is associated with a single VBA project that contains all VBA artifacts (scripts, forms, and so on) that you created within the Rational Rhapsody project. This project file has the name `<project name>.vba` and is located in the same directory as the Rational Rhapsody project file (`<project>.rpy`). This binary file will be loaded (if present) with the Rational Rhapsody project and saved when you select **Save** from Rational Rhapsody or the VBA IDE.

VBA versus VB programs

The major difference between writing API external programs with VB and writing VBA scripts inside Rational Rhapsody is the availability of the Rational Rhapsody root object, known as the Rational Rhapsody application. External VB programs need to create a Rhapsody application object; Rhapsody VBA scripts have direct access to the already existing application object.

Whether accessed by VB or VBA programs, operations of the `Application` object are identical in function. To the VBA user, however, it looks like all the methods of the root object are local methods in the VBA context. For example, traversing the Rhapsody model always starts with

accessing the project object. The following example shows a VBA script that displays the name of the project:

```
Dim a as Object
Set a = getProject
MsgBox a.name
```

Note: The method `getProject` is a function of the root object.

Writing VBA macros

Rhapsody allows you to program a script (or “macro”) in the Microsoft Visual Basic programming language to perform automated activity.

To write a Visual Basic macro for Rational Rhapsody:

1. Launch the VBA IDE in one of the following ways:
 - a. Select **View > Toolbars > VBA** and then select the first icon from the left to launch the VBA IDE.
 - b. Select **Tools > VBA** and then select **Visual Basic Editor** from the popup menu.
2. Edit the Visual Basic project file, `<project>.vba`, to implement different macros. Once you are finished editing, exit the VBA IDE and save the Rhapsody project. The VBA project is automatically saved whenever the Rhapsody project is saved.

Running and sharing macros

Later, you can run a Rhapsody VBA macro from the Macros window or as a helper in the Tools menu. In addition, macros can be shared with other users through the macro exporting and importing process.

API methods for a VBA macro

Note that helper applications might not close the current document. This means that you should not use the following API methods in a VBA macro that you specify as a helper:

Method	Interface Object
quit	IRPApplication
openProject	IRPApplication
close	IRPPProject

Creating and editing macros

You can create a new macro or edit an existing macro in two ways:

- ♦ Using the VBA Macros window in Rational Rhapsody
 - To create a macro, type in a new name in the **Macro Name** field, then select **Create**.
- ♦ Launch the VBA IDE and create and edit new macros there. There, you can do one of the following actions:
 - Select **Tools > VBA > Macros** to open the Macros window.

Start typing the new macro with the line `Sub xxxx( )`, where `xxxx` is the name of the new macro. The last line of the macro must be “End Sub.”

- Find an existing macro by expanding the Modules folder of the Project window and double-clicking the appropriate module. You can scroll the code window to the existing macro or select it in the right pull-down above the code window.

This is a simple VBA macro:

```
Sub GetNameOfProject()  
Dim a as Object  
Set a = getProject  
MsgBox a.name  
End Sub
```

Once you have finished typing this macro, return to the Rhapsody window and run the new macro through the Macros window. You see a small message box with the name of the currently loaded project.

VBA Macros window

The VBA Macros window enables you to run, edit, or delete a macro.

To open the Macros window, use the VBA Toolbar shortcut or select **Tools > VBA > Macros**. The following figure shows the Macros window. The window contains the following fields:

- ♦ **Macro Name** contains the name of the highlighted macro in the **Macro Box** field. This field is blank if there are no macros in the **Macro Box**.
- ♦ **Macro Box** lists the available macros in the VBA project selected in the **Macros In** box.
- ♦ **Macros In** lists the available VBA projects that contain macros.

The window contains the following buttons:

- ♦ **Run** runs the selected macro.

To run a macro, highlight a macro in the **Macro** box, then click **Run**.

- ♦ **Step Into** highlights the first line of the macro and places the Current Execution Line Indicator.
- ♦ **Edit** opens the Code window with the selected macro visible so you can modify your macro.

To edit a macro, highlight the macro in the **Macro** box, then click **Edit**.

- ♦ **Create** opens a module in the Code window so you can create a new macro.

To create a macro, type in a new name in the **Macro Name** field, then click **Create**.

Note: Since VBA macros are contained in modules, you must first create a module before creating your first macro. If you have not yet created a module, the **Create** button is disabled. Modules cannot be created from the Macros window. You must open the VBA IDE to do so.

- ♦ **Delete** removes the selected macro from your project.

To delete a macro, highlight a macro in the **Macro** box, then click **Delete**.

Saving your macros

Rational Rhapsody VBA macros are saved automatically with your Rhapsody project. When you load the project again, the macros you have created for it will be available.

Exporting and importing VBA macros

To export the VBA macros for a module from the VBA IDE:

1. Select a module from the modules tree.
2. From the VBA IDE, select **File > Export File**.
3. In the Export Files window, browse to the correct location and enter the name of the receiving file.
4. Select **OK** to dismiss the Export Files window.

Rational Rhapsody also enables you to import an existing module or form to the project.

To import VBA macros:

1. From the VBA IDE, select **File > Import File**. The Import Files window is displayed.
2. Browse to the correct location and select the file to import.

A copy of the file is added to the project and the original file is left intact. If you import a form or module with the same name as an existing form or module, the new form or module file is added with a number appended to its name.

Creating a customized profile

To create a customized profile:

1. Create a project that you want to use as the basis for your customized profile. If you select a Rational Rhapsody profile for this project, the characteristics of that profile are going to be used as the default values.
2. Right-click the top-level project name (for example, **Dishwasher**) and select **Add New > Profile**, and then enter a name for your profile. Notice that Rational Rhapsody creates a **Profiles** category and places your profile within it.

Alternatively, if you have a package that you want to change to be a profile, right-click the package and select **Change to > Profile**. For more information, see [Converting packages and profiles](#).

3. Enter any information about the profile that you want your team members to know about on the **Description** tab.
4. Optionally, you can do the following actions:
 - a. Define global tags for your profile: Open the Features window for the profile (for example, double-click the profile name) and define tags on the **Tags** tab.
 - b. Add a stereotype to your profile: On the **General** tab of the Features window for your profile, select <<New>> from the **Stereotype** box. A Features window opens for the stereotype on which you can name the stereotype. Notice that by default Rational Rhapsody sets that this stereotype is applicable to a profile. Notice also that you can make it a New Term stereotype. For more information about creating stereotypes, see [Stereotypes](#).

After you close the Features window for the stereotype and return to the Features window for the profile, notice that the stereotype you just created is showing in the **Stereotype** box of the **General** tab for the profile.

Creating a new stereotype for the new profile

To create a stereotype, but not immediately apply it to a profile:

1. Right-click your profile and select **Add New > Stereotype**. Remember to select the metaclass to which the stereotype applies.

Later you can apply the stereotype to a profile by opening the Features window for the profile and selecting the particular stereotype on the **General** box.

2. Once Rational Rhapsody creates the **Stereotype** category on the browser, you can right-click the name and select **Add New Stereotype** to create more stereotypes.

Re-using your customized profile

Since a profile is a package, it can be used in other projects to save time and support corporate standards easily.

To re-use your customized profile:

1. Make a corresponding text file for your custom profile and give it the same name but with the .txt extension. Add a description for the profile. The content of the text files displays on the description area of the New Project window.
For example, if your customized profile is `MyProfile.sbs`, create a `MyProfile.txt` file.

2. Copy the .sbs file for the profile and the corresponding .txt file to <Rational Rhapsody installation path>\Share\Profiles\<customized profiles foldername>. For example, you would place a copy of the `MyProfile.sbs` and `MyProfile.txt` files in, for example, `C:\Rhapsody\Share\Profiles\CustomProfiles`.

If you have packages under your profile, choose to not make each package its own separate unit so that you can keep the entire profile in a single .sbs file (clear the **Store in separate file** for those package).

3. When you want to use a custom profile with a new project, you can select the profile from the **Project Type** drop-down list of the New Project window.

You can also share your custom profiles through the following methods:

- ◆ Save the customized profile (package) and make it available through a CM system or in a shared area where other developers can access it.
- ◆ Add a custom profile to an existing project with **File > Add Profile to Model**. Select the customized profile (an .sbs file) from its stored location.
Note that with this method certain elements might not be brought over to your new project is they are not associated specifically with the profile/project.
- ◆ When multiple projects are displayed in a browser, a developer can drag and drop the customized profile from one project to a different open project to re-use it. For more information about this process, see the instructions in [Copy and reference elements among projects](#).

Note

You can set your customized profile to be automatically added to a new project either as a copy or a reference using the AutoCopied or AutoReferences properties. For more information, see [Profile properties](#).

Adding new element types

The stereotype mechanism is used for introducing such new elements. In general, stereotypes are used to add new information to a model element. However, if you define a new stereotype and specify that it should be a **New Term**, it becomes a new element that can be used in models.

To add a new type of element:

1. Create a new stereotype. See [Stereotypes](#).
2. Open the Features window for the new stereotype and select one item from the **Applicable To** list. This item is the element on which the new element is based.
3. Select **New Term**.
4. Click **OK**.

Once the new term is created, it is possible to add elements of this type via the context menu in the browser.

New terms and their properties

For each new term introduced, properties can be used to specify characteristics such as the type of icon to use for the term in the browser or the icon to be used in the Diagram Tools. For each of the available properties, if no value is provided, Rational Rhapsody uses the value provided for the element on which the new term is based.

Note

Since stereotypes can be added to profiles and packages, the new terms created can be shared across models.

Availability of out-of-the-box model elements

In addition to allowing the introduction of new element types, Rational Rhapsody allows you to hide any out-of-the-box element types that your users do not need.

The availability of *metaclasses* is determined by the `General::Model::AvailableMetaclasses` property. This property takes a comma-separated list of strings.

Note

To keep all of the out-of-the box metaclasses, leave this property blank.

To limit the availability of certain metaclasses, use this property to indicate the metaclasses that you would like to have available. The strings to use to represent the different metaclasses are as follows

- ◆ ActivityDiagram
- ◆ ActivityFlow
- ◆ Actor
- ◆ Argument
- ◆ Association
- ◆ AssociationEnd
- ◆ Attribute
- ◆ Block
- ◆ Class
- ◆ ClassifierRole
- ◆ CollaborationDiagram
- ◆ CombinedFragment
- ◆ Comment
- ◆ Component
- ◆ ComponentDiagram
- ◆ ComponentInstance
- ◆ Configuration
- ◆ Connector
- ◆ Constraint
- ◆ Constructor

- ◆ ControlledFile
- ◆ Dependency
- ◆ DeploymentDiagram
- ◆ Destructor
- ◆ EnumerationLiteral
- ◆ Event
- ◆ ExecutionOccurrence
- ◆ File
- ◆ Flow
- ◆ FlowItem
- ◆ Folder
- ◆ Generalization
- ◆ HyperLink
- ◆ InteractionOccurrence
- ◆ InteractionOperand
- ◆ Link
- ◆ Message
- ◆ Module
- ◆ Node
- ◆ Object
- ◆ ObjectModelDiagram
- ◆ Package
- ◆ Pin
- ◆ Port
- ◆ PrimitiveOperation
- ◆ Profile
- ◆ Project
- ◆ Reception
- ◆ ReferenceActivity
- ◆ Requirement
- ◆ SequenceDiagram
- ◆ State

- ◆ Statechart
- ◆ Stereotype
- ◆ StructureDiagram
- ◆ Swimlane
- ◆ SysMLPort
- ◆ Tag
- ◆ Transition
- ◆ TriggeredOperation
- ◆ Type
- ◆ UseCase
- ◆ UseCaseDiagram

Creating a customized diagram

In addition to allowing you to filter out certain out-of-the-box diagrams that you do not want to see, Rational Rhapsody allows you to add customized diagrams. This is done by creating a new diagram type on the basis of one of the Rational Rhapsody basic diagrams and adding customized diagram elements, if needed, to the list of elements available for the new type of diagram.

Note

The procedure for adding customized diagrams with custom elements can only be used for adding new types of diagrams. It is not possible to add new diagram element types to the standard Rational Rhapsody diagrams.

Customized diagrams can be added at the individual model level, or they can be added to profiles so that they can be used with other models as well.

To create your customized diagram:

1. In the browser window, add your customized profile. See [Creating a customized profile](#). (While the customized diagram can be added for the current model only, usually developers and designers want to add it to a profile so that it can be reused.)
2. Select the name of the new profile in the browser, and use the context menu to create a new *stereotype*. See [Stereotypes](#).
3. Open the Features window for the new stereotype you created, and set the following values:
 - a. On the **General** tab, from the **Applicable to** list select the type of diagram that should serve as the base diagram for the new diagram type you are creating. In addition, select the **New Term** check box.
 - b. On the **Properties** tab, enter the required values for the following properties:
 - `Model::Stereotype::DrawingToolIcon` supplies the name of the .ico file that should be used as the icon for the new diagram type in the **Diagrams** toolbar.
 - `Model::Stereotype::BrowserIcon` supplies the name of the .ico file that should be used as the icon to represent the new diagram type in the browser.

If no value is provided for `DrawingToolIcon`, the file name entered for `BrowserIcon` are used in the **Diagrams** toolbar as well. If values are not provided for either of these properties, then the icon for the base diagram is displayed both in the browser and in the **Diagrams** toolbar.

 - `Model::Stereotype::DrawingToolbar` is a comma-separated list representing the elements that should be included in the **Diagram Tools** for this type of diagram, for example, `RpyDefault`, `RpySeparator`, `Actor`, `Block`.

`RpyDefault` represents all the elements included in the **Diagram Tools** of the base diagram. If this property is left empty, only the tools from the base diagram is displayed. The toolbar can contain any drawable elements supported by the base diagram, and any new elements based on these elements.

Adding customized diagrams to the diagrams toolbar

Once you have created a customized diagram type, it is included automatically in the **Tools** menu. You also have the option of including an icon for the new diagram type in the **Diagrams** toolbar. To add the new type of diagram to the **Diagrams** toolbar:

1. Open the Features window for the profile to which you added the new type of diagram.
2. On the **Properties** tab, modify the value of the `General::Model::DiagramsToolbar` property to include the name of the new diagram type in the comma-separated list, for example, `OV-1, RpySeparator, RpyDefault`. (If this property is left empty, the toolbar includes only the default icons.)

The strings to use in this list are given in [Diagram types](#).

Creating a customized diagram element

After a new diagram type have been defined, you can define new drawing elements that can be included in the **Diagram Tools** for the new type of diagram. This is done by basing the new element on one of the elements that is available by default in the diagram type that served as the base for the new customized diagram, as follows:

1. Select the name of the relevant profile in the browser, and use the context menu to create a new stereotype.
2. Open the Features window for the new stereotype you created, and set the following values:
 - a. On the **General** tab, from the **Applicable to** list select the type of drawing element that should serve as the base element for the new diagram element type you are creating. Also, select the **New Term** check box.
 - b. On the **Properties** tab, provide values for the following properties:
 - `Model::Stereotype::DrawingToolIcon` provides the name of the .ico file that should be used as the icon for the new drawing element when it is included in a **Diagram Tools**.
 - `Model::Stereotype::DrawingToolTip` provides the text that should be displayed as a tool tip for the icon in the **Diagram Tools**.

- `Model::Stereotype::DrawingShape` where if you would like to customize, to a certain degree, the appearance of the new element that you created, you can select one of the options provided for this property, for example, you can create a new element based on **Class**, but specify that the object have "rounded corners" when displayed on a diagram.
- `Model::Stereotype::AlternativeDrawingTool` where in certain cases, a number of different out-of-the-box drawing elements are based on the same metaclass, for example, both Class and Composite Class are based on a metaclass called Class. In these cases, in addition to specifying the base metaclass in the **Applicable to** box, you must provide the name of the wanted base element in the property. This property does not have to be provided for the "default" element for the metaclass. Using our example above, if you were basing the new element on the Class element, there would be no need to provide a value for this property.

The complete list of diagram elements and corresponding metaclasses is provided in [Diagram elements](#). This table also indicates the “default” diagram element for each metaclass.

Adding customized diagram elements

After customized diagram elements have been created, they can be added to one or more of the customized diagram types you have created:

1. In the browser, under **Stereotypes**, select the customized diagram to which you would like to add the custom element, and open its Features window.
2. On the **Properties** tab, for the `Model::Stereotype::DrawingToolbar` property, add the name of the new element type you created to the comma-separated list representing the elements that should be included in the **Diagram Tools** for this type of diagram.

Then names of the elements that can be used for this list are given in [Diagram elements](#).

Note

After defining new diagrams or diagram elements, you need to reload the model to have access to the new items or, alternatively, choose **View > Refresh New Terms**.

Diagram types

The following list contains the strings to use for the `General::Model::DiagramsToolbar` property:

- ◆ ActivityDiagram
- ◆ CollaborationDiagram
- ◆ ComponentDiagram
- ◆ DeploymentDiagram
- ◆ ObjectModelDiagram
- ◆ SequenceDiagram
- ◆ Statechart
- ◆ StructureDiagram
- ◆ UseCaseDiagram
- ◆ RpyDefault
- ◆ RpySeparator

Diagram elements

The following elements can be customized for the specified diagrams, as described in [Adding customized diagram elements](#).

Element Name	Metaclass Name	AlternativeDrawingTool Property Required
Object Model Diagram		
Object	Object	
Class	Class	
Composite Class	Class	Yes
Package	Package	
Port	Port	
Inheritance	Generalization	
Association	AssociationEnd	
Directed Association	AssociationEnd	Yes
Composition	AssociationEnd	Yes
Aggregation	AssociationEnd	Yes
Link	Link	
Dependency	Dependency	
Flow	Flow	
Actor	Actor	
Sequence Diagram		
InstanceLine	ClassifierRole	Yes
EnvironmentLine	ClassifierRole	Yes
Message	Message	
ReplyMessage	Message	Yes
CreateMessage	Message	Yes
DestroyMessage	Message	Yes
TimeoutMessage	Message	Yes
CancelTimeoutMessage	Message	Yes
TimeIntervalMessage	Message	Yes
PartitionLine	ClassifierRole	Yes
Condition Mark	Message	Yes
ExecutionOccurrence	ExecutionOccurrence	

Element Name	Metaclass Name	AlternativeDrawingTool Property Required
InteractionOccurrence	InteractionOccurrence	
InteractionOperatorCombinedFragment	CombinedFragment	
InteractionOperand	InteractionOperand	
Use Case Diagram		
UseCase	UseCase	
Actor	Actor	
Package	Package	
Association	AssociationEnd	
Generalization	Generalization	
Dependency	Dependency	
System Border	ClassifierRole	Yes
Flow	Flow	
Collaboration Diagram		
Classifier Role	ClassifierRole	Yes
Multi Object	ClassifierRole	Yes
Classifier Actor	ClassifierRole	Yes
AssociationRole	ClassifierRole	Yes
Link Message	Message	Yes
Reverse Link Message	Message	Yes
Dependency	Dependency	
Structure Diagram		
Composite Class	Class	Yes
Object	Object	
Block	Block	
Port	Port	
Link	Link	
Dependency	Dependency	
Flow	Flow	
Deployment Diagram		
Node	Node	
Component	Component	

Element Name	Metaclass Name	AlternativeDrawingTool Property Required
Dependency	Dependency	
Flow	Flow	
Component Diagram		
Component	Component	
File	File (Component)	
Folder	Folder	
Dependency	Dependency	
Interface	Class	Yes
Realization	Cannot use as base element	
Flow	Flow	
Statechart		
State	State	
ActivityFlow	Transition	
InitialFlow	DefaultTransition	
AndLine	Cannot use as base element	
StateChartConditionConnector	Connector	Yes
HistoryConnector	Connector	Yes
TerminationConnector	Connector	Yes
JunctionConnector	Connector	Yes
DiagramConnector	Connector	Yes
StubConnector	Connector	Yes
JoinConnector	Connector	Yes
ForkConnector	Connector	Yes
TransitionLabel	Transition	Yes
TerminationState	State	Yes
Dependency	Dependency	
Activity Diagram		
Action	State	Yes
ActionBlock	State	Yes
SubActivityState	State	Yes
ObjectNode	State	Yes

Element Name	Metaclass Name	AlternativeDrawingTool Property Required
ReferenceActivity	ReferenceActivity	
Transition	Transition	
DefaultTransition	DefaultTransition	
LoopTransition	Transition	Yes
ActivityChartConditionConnector	Connector	Yes
TerminationState	Connector	Yes
JunctionConnector	Connector	Yes
DiagramConnector	Connector	Yes
JoinConnector	Connector	Yes
ForkConnector	Connector	Yes
TransitionLabel	Transition	Yes
Swimlane Frame	Swimlane	Yes
SwimlaneDivider	Swimlane	Yes
Dependency	Dependency	
ActivityPin	Connector	Yes
ActivityParameter	Connector	Yes

Customize the Add New menu

Rational Rhapsody offers you a number of ways to customize the **Add New** menu. This menu is the one you see when you right-click an item in the Rational Rhapsody browser and select **Add New**.

Re-organizing the common list section of the Add New menu

The top section of the **Add New** menu is known as the common list portion of this menu. The `General::Model::CommonList` property controls which elements appear in this section, when applicable.

To re-organize the common list section of the **Add New** Menu for a project:

1. Open your project in Rational Rhapsody.
2. Open the Features window. Choose **File > Project Properties**.
3. Locate `General::Model::CommonList` and change the values for this property.
4. Click **OK**.
5. To confirm your change, right-click a package in your project and select **Add New**. Only those values you entered in `CommonList` appear, as long as they are applicable to your project.

Note the following information:

- ◆ Whatever element that is removed from the common list group of the **Add New** menu will appear in the middle section of the **Add New** menu if it is relevant for your project. The element must appear somewhere if it is a valid element for your project.
- ◆ If the `AddNewMenuStructure` property is in use, that property overrides the properties mentioned here. See [Customizing the Add New menu completely](#).
- ◆ If you want to use this **Add New** menu customization in other projects (without having to manually change the property for each of them), see [Re-using property changes to the Add New menu](#).
- ◆ See also [Re-organizing the bottom section of the Add New menu](#).

Re-organizing the bottom section of the Add New menu

You can re-organize the groups and their elements located at the bottom section of the **Add New** menu. If there are groups or elements that you do not use by project, you can re-organize this section of the **Add New** menu to list only those groups/elements that you want to use.

To re-organize the **Add New** menu, use the following properties in `General::Model`:

- ◆ SubmenuList
- ◆ Submenu1List
- ◆ Submenu1Name
- ◆ Submenu2List
- ◆ Submenu2Name
- ◆ Submenu3List
- ◆ Submenu3Name
- ◆ Submenu4List
- ◆ Submenu4Name

A definition for each property displays on the **Properties** tab of the Features window.

The following example shows how you could re-organize the groups and their elements located at the bottom of the **Add New** menu. In this scenario, you want to show only the activity diagram, flowchart, and panel diagram in the **Diagrams** group of the **Add New** menu for your project. In addition, you want to remove the **TableMatrix** group.

Note: This example assumes that the properties mentioned in this topic have the default values.

1. Open your project in Rational Rhapsody.
2. Open the Features window. Choose **File > Project Properties**.
3. Locate `General::Model::Submenu1List`.
Note that this property is associated with the `Submenu1Name` property, which has a value of **Diagrams**.
4. Type `ActivityDiagram`, `Flowchart`, and `PanelDiagram` as the values for `Submenu1List`.
5. Click **OK**.
6. Locate `General::Model::SubmenuList`.
This property controls what submenu groups appear at the bottom of the **Add New** menu.
7. Delete `Submenu3` from the values entered for the `SubmenuList` property.
`Submenu3` specifies the **TableMatrix** submenu that can appear on the **Add New** menu.

8. To confirm your changes, right-click a package in your project and select **Add New**. Only those diagram types you entered in Submenu1List appear, as long as they are applicable to your project. Meaning, if a flowchart is not applicable to your project, that choice will not appear on the **Add New** menu under the **Diagrams** category. In addition, the **TableMatrix** submenu item no longer displays on the **Add New** menu.

Note the following information:

- ◆ Whatever element that is removed from a group from the bottom of the **Add New** menu will appear in the middle section of the **Add New** menu if it is relevant for your project. When you remove a group, all the elements in that group will appear in the middle section of the **Add New** menu. Elements must appear somewhere if they are valid elements for your project.
- ◆ The Submenu1List and Submenu1Name properties are also used by **Tools > Diagrams**. When you make a change to Submenu1List, to have it take effect on **Tools > Diagrams**, you must save your project, close it, and then open it again. In addition, if you delete the Submenu1 value from the SubmenuList property, all the Rational Rhapsody diagram choices will appear in the **Tools** menu, instead of under **Tools > Diagrams** (after you save your project and open it again).
- ◆ If the AddNewMenuStructure property is in use, that property overrides the properties mentioned in this topic. See [Customizing the Add New menu completely](#).
- ◆ If you want to use this **Add New** menu customization in other projects (without having to manually change the properties for each of them), see [Re-using property changes to the Add New menu](#).
- ◆ See also [Re-organizing the common list section of the Add New menu](#).

Customizing the Add New menu completely

You can completely customize the choices that appear in the **Add New** menu to focus on a particular process or need.

To completely customize the choices that appear in the **Add New** menu for your project:

1. Open your project in Rational Rhapsody.
2. Open the Features window. Choose **File > Project Properties**.
3. Locate the `General::Model::AddNewMenuStructure` property and set the elements you want to appear for the **Add New** menu.
The property definition on the **Properties** tab of the Features window provides you with more information on how to use this property.
4. Click **OK**.
5. To confirm your changes, right-click a package in your project and select **Add New**. Only those elements you entered in `AddNewMenuStructure` appear on the **Add New** menu, as long as they are applicable to your project.

Note: If you want to use this **Add New** menu customization in other projects (without having to manually change the property for each of them), see [Re-using property changes to the Add New menu](#).

Compare with [Re-organizing the common list section of the Add New menu](#) and [Re-organizing the bottom section of the Add New menu](#).

Re-using property changes to the Add New menu

You can automatically apply the changes you make to the properties that control the **Add New** menu for use in other projects by setting your applicable property changes in a New Term stereotype that is associated with a particular profile that is set to be applicable to a project, and then applying the profile to your other projects.

To apply property changes to the **Add New** menu to other projects:

1. Create a new project (name it, for example, Project2).
2. Create a profile for the project. Right-click the project name, select **Add New > Profile** and name it (for example, MyProfile).
3. Create a stereotype for the profile. Right-click the profile and select **Add New > Stereotype** and name it the same as the profile.
If you name this stereotype the same name as the profile (in this example, MyProfile, Rational Rhapsody will auto-apply the stereotype to a project when you use the profile.
4. Open the Features window for the stereotype:
 - a. On the General tab:
 - Click the **New Term** check box to make this a New Term stereotype for this profile.
 - From the **Applicable to** drop-down list, select the **Project** check box to make this New Term stereotype applicable to a project.
 - b. On the Properties tab, set the elements you want to appear for the **Add New** menu through the use of the following properties:
 - `General::Model::CommonList` property if you want to re-organize the common list section of the **Add New** menu.
 - `General::Model::SubmenuList`, `General::Model::Submenu#List`, and `General::Model::Submenu#Name` if you want to re-organize the bottom section of the **Add New** menu
 - `General::Model::AddNewMenuStructure` property if you want to completely customize the **Add New** menu.
Note that using this property overrides the other properties mentioned in this topic.
 - c. Click **OK**.

5. Set the stereotype for the project (in this example, Project2) to the New Term stereotype (in this example, MyProfile).
 - a. Open the Features window for the project.
 - b. On the General tab, in the Stereotype drop-down list, select the check box for stereotype (in this example, MyProfile).
 - c. Click **OK**.
6. To confirm your changes, right-click a package in your project and select **Add New**. Only those elements you entered in the `CommonList` property and/or `SubmenuList`, `Submenu#List`, and `Submenu#Name` properties, or `AddNewMenuStructure` appear, as long as they are applicable to your project.
7. Make a corresponding text file for your custom profile and give it the same name but with the `.txt` extension. Add a description for the profile. The content of the text files displays on the description area of the New Project window.
For example, if your customized profile is `MyProfile.sbs`, create a `MyProfile.txt` file.
8. Copy the `.sbs` file for the profile and the corresponding `.txt` file to `<Rational Rhapsody installation path>\Share\Profiles\<customized profiles foldername>`. For example, you would place a copy of the `MyProfile.sbs` and `MyProfile.txt` files in, for example, `C:\Rhapsody\Share\Profiles\CustomProfiles`.

If you have packages under your profile, choose to not make each package its own separate unit so that you can keep the entire profile in a single `.sbs` file. Right-click the package, select **Unit > Edit Unit**, and clear the **Store in separate file** check box on the Unit Information window that opens).
9. To use the custom profile, when you create a new project, select the profile from the **Project Type** drop-down list of the New Project window.

Creating a Rational Rhapsody plug-in

Rational Rhapsody plug-ins are Java applications that users can write to extend the capabilities in Rational Rhapsody. Rational Rhapsody loads these applications into its process, and provides the applications with an interface to the functions in Rational Rhapsody.

The capabilities added via plug-ins can be accessed through customized menu items integrated into the out-of-the-box menus in Rational Rhapsody. Plug-ins can also provide capabilities that are not opened directly by the user via the GUI, but rather are triggered by specific Rational Rhapsody events, such as model checking or code generation. Plug-ins can respond to any of the events defined in the Rational Rhapsody Callback API.

To write and prepare a plug-in for use with Rational Rhapsody:

1. Write the Java application.
2. Create a .hep file that contains the information that Rational Rhapsody requires to load the plug-in, or add this information to an existing .hep file if you have already created one.
3. Attach the .hep file to a profile.

Writing a Java plug-in for Rational Rhapsody

While the steps described in this section can be carried out using any Java IDE, the text and screen captures provided demonstrate how to perform these steps in Eclipse (version 3.4).

In terms of writing the Java code for your plug-in, you should:

1. Create a new Java project
2. Add the Rational Rhapsody library to the project's build path
3. Define a Java class for the plug-in
4. Implement the required methods

Creating a new Java project

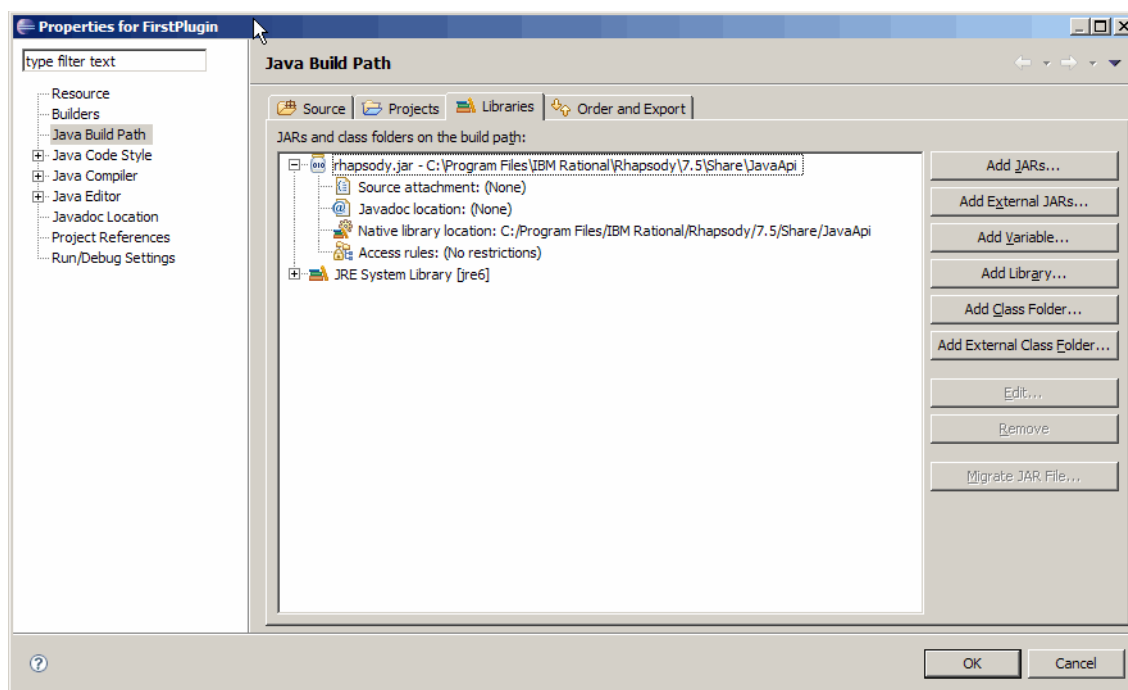
To create a new Java project:

From the Eclipse main menu, select **File > New > Project > Java Project**.

Adding Rational Rhapsody library to project build path

To add the Rational Rhapsody library to the project's build path:

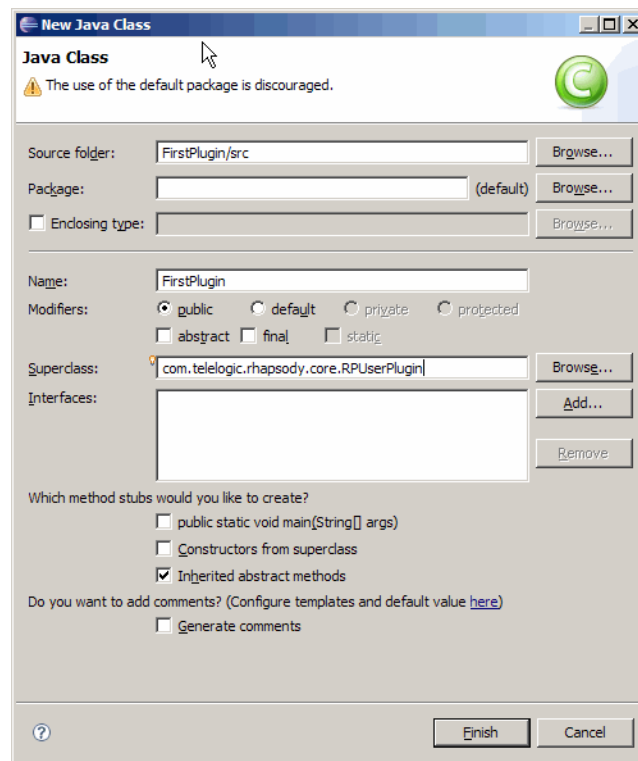
1. In the Package Explorer View, right-click the project you created, and then select **Build Path > Configure Build Path**.
2. Go to the Libraries tab and click **Add External JARs**.
3. Select *rhapsody.jar*, under <Rational Rhapsody installation path>\Share\JavaAPI
4. After *rhapsody.jar* has been added to the list, expand it and set the Native library location to <Rational Rhapsody installation path>\Share\JavaAPI



Defining Java class for plug-in

To define a Java class for the plug-in:

1. In the Package Explorer View, right-click the project, and then select **New > Class**
2. When the New Java Class window is displayed, give the class a name, enter RPUsePlugin for the Superclass, and make sure the **Inherited abstract methods** check box is selected.



Implementing the required methods

Your plug-in class must implement the following methods:

```
//called when the plug-in is loaded
public abstract void RhpPluginInit(final IRPApplication rhApp);

//called when the plug-in's menu item under the "Tools" menu is selected
public void RhpPluginInvokeItem();

//called when the plug-in popup menu (if applicable) is selected
public void OnMenuItemSelect(String menuItem);
```

```
//called when the plug-in popup trigger (if applicable) is fired
public void OnTrigger(String trigger);

//called when the project is closed - if true is returned, the plug-in will be
unloaded
public boolean RhpPluginCleanup();

//called when Rhapsody exits
public void RhpPluginFinalCleanup();
```

Creating a .hep file for the plug-in

To provide Rational Rhapsody with the information necessary to load your plug-in, you must create a .hep file or add this information to an existing .hep file if you have already created one.

.hep file structure

To understand the types of information that must be included for plug-ins in .hep files, it's best to start with the issue of what types of elements can be described in .hep files.

.hep files are used to describe the following items:

- ♦ helpers
- ♦ plug-ins
- ♦ plug-in commands

While these items differ from one another, they use the same .hep file entries to provide Rational Rhapsody with the required information.

Helpers are also used to extend the capabilities in Rational Rhapsody, but they use a different mechanism than plug-ins. Helpers are basically stand-alone applications. Plug-ins, on the other hand, are not stand-alone applications. They just use the Rational Rhapsody callback mechanism to respond to Rational Rhapsody events.

Plug-in commands don't really add any functionality of their own; they just describe context menu items that Rational Rhapsody should add to allow you to open a certain plug-in.

For plug-ins, the .hep file must contain the following information:

- ♦ the number of items (plug-ins/helpers/plug-in commands) defined in the file
- ♦ the name of the plug-in
- ♦ the Java class that implements the required methods
- ♦ the Java classpath used by your plug-in

- ♦ an indication that the item is a plug-in (not a helper or plug-in command)
- ♦ an indication of whether or not a menu item should be added to the Tools menu

The best way to describe the required syntax for the .hep file is to look at an example.

[Helpers]

Category for the entries that follow

Note

The .hep file must contain a [Helpers] section because the helper recognition mechanism is the same one used when you include helper definitions in your *rhapsody.ini* file, which has other sections as well. Since plug-ins are usually designed for use by groups of users, in most cases it does not make sense to include the plug-in definition information in the *rhapsody.ini* file, which is unique to each user.

numberOfElements=2

Number of plug-ins/helpers described in the file

name1=Diagram Formatter

The name that will appear on the Tools menu (if isVisible is set to 1)

JavaMainClass1=JavaPlugin.PluginMainClass

The Java class containing the plug-in code

JavaClassPath1=\$OMROOT\..\Samples\JavaAPI Samples\Plug-in

Path for locating the java classes required by the plug-in. Keep in mind that if .jar files are used, the classpath should include the names of the .jar files.

isPlugin1=1

Indicates the item is a plug-in (as opposed to a helper, which is the default, or a "plug-in command")

isVisible1=1

Indicates that the name should be displayed in the Tools menu.

The entries below describe a "plug-in command"

name2=Format Diagram

The text that will appear in the context menu

isPluginCommand2=1

Indicates that this is a plug-in command (as opposed to a helper or plug-in)

command2=Diagram Formatter

Then name of the plug-in that will be opened by this context menu item

`applicableTo2=ObjectModelDiagram`

Indicates the context to which the menu will be added. In this case, when you right-click an OMD in the browser, you will see the option "Format Diagram".

`isVisible2=1`

Indicates that the menu item should be displayed

If you would like to see another sample .hep file, take a look at the .hep file for the plug-in sample included with <Rational Rhapsody installation path>\Samples\ExtensibilitySamples\Simple Plug-in\SimplePluginProfile.hep).

Attaching a .hep file to a profile

To attach your .hep file to a profile, do one of the following actions:

- ♦ Give the .hep file the same name as the profile, and place it in the same directory as the profile's .sbs file
- ♦ Indicate the path to the .hep file in the value of the property `General::Profile::AdditionalHelpersFiles` for the profile

Troubleshooting Rational Rhapsody plug-ins

If the plug-in does not appear to be loaded, or if it does not respond as expected to Rational Rhapsody events, you can log Rational Rhapsody's attempts to interact with the plug-in. To have Rational Rhapsody create such a file, add the following entry to the [General] section of the *rhapsody.ini* file:

`JavaAPILogFile=[path and filename to use]`

The log file does not clear its contents between Rational Rhapsody sessions. You might want to remove the `JavaAPILogFile` entry from the *rhapsody.ini* file as soon as you have solved the problem.

In the log file, you might encounter the following common errors:

- ♦ *ClassNotFoundException*: <Class name> - check that the classpath is correct
- ♦ *UnsupportedClassVersionError*: <Class name> - indicates that the class was compiled with a newer version of Java than the one that Rational Rhapsody is using. Try to compile your plug-in with lower compliance, or change the JVM used by Rational Rhapsody (specified in the JVM section in the *rhapsody.ini* file).
- ♦ *NoSuchMethodException*: <Class name>.<Method signature> - make sure the method has been defined in your plug-in class

Debugging Rational Rhapsody plug-ins

You can debug your plug-in:

- ♦ as a stand-alone Java application
- ♦ from within a Rational Rhapsody process

Debugging as a stand-alone Java application

To debug as a stand-alone application, you will need to write a main operation that simulates a Rational Rhapsody callback. Below is an example:

```
public static void main(String[] args) {
    //create an instance of my plug-in
    MyPlugin myPlugin = new MyPlugin ();
    //get Rhapsody application that is currently running
    IRPApplication app
        =RhapsodyAppServer.getActiveRhapsodyApplication();
    //init the plug-in
    myPlugin.RhpPluginInit(app);
    //simulate a call to the plug-in
    myPlugin.RhpPluginInvokeItem();
}
```

Once you have included such a main operation, you can run Rational Rhapsody and debug the plug-in as you would any other Java application.

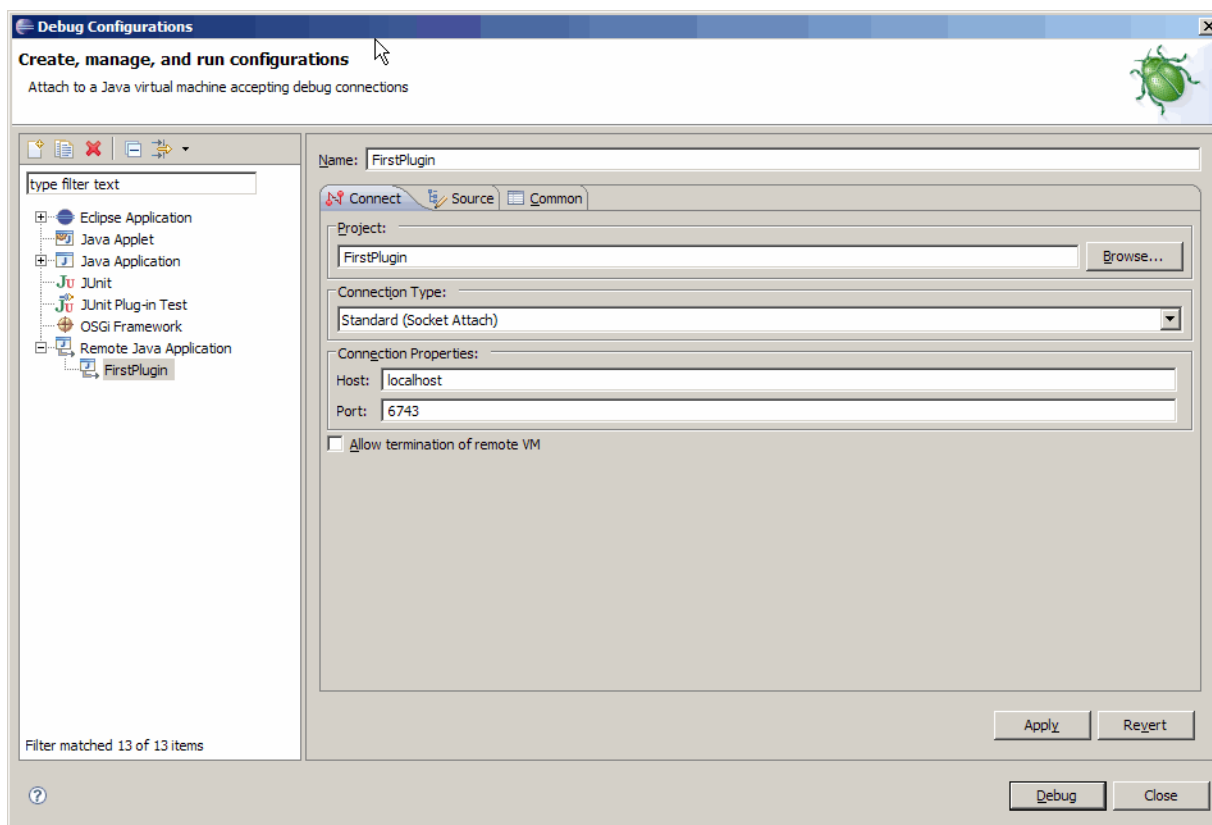
Debugging from within Rational Rhapsody

To debug from within Rational Rhapsody:

1. Add the following debug options to the JVM section of the `rhapsody.ini` file:

```
[JVM]
Options=ClassPath,LibPath,Debug1,Debug2,Debug3
Debug1=-Xnoagent
Debug2=-Xdebug
Debug3=-Xrunjdwptransport=dt_socket,address=6743,server=y,suspend=y
```

2. Open your Java plug-in project in Eclipse and create a *Remote Java Application* configuration as follows:
 - a. Choose **Run > Debug Configurations**.
 - b. In the Debug Configurations window, right-click **Remote Java Application**, and then select **New**.
 - c. Set the port number to 6743 or any other number that you entered for "address" in the JVM settings in the `rhapsody.ini` file.



3. Open Rational Rhapsody and open the project that loads your plug-in. Once the project is loaded, Rational Rhapsody will wait until you start the debug session.
4. Set breakpoints in your code and start the debug session.

The simple plug-in sample

The Rational Rhapsody installation contains a sample plug-in called **Simple Plug-in** (under **ExtensibilitySamples**).

To see the capabilities that this sample plug-in adds to Rational Rhapsody, add `SimplePluginProfile.sbs` to a Rational Rhapsody model "As Reference".

This profile will load the plug-in and a message will be displayed indicating the Rational Rhapsody build number you are using.

The plug-in adds the following menu items:

- ♦ **SimplePlugin** under the **Tools** menu - when you select this menu item, Rational Rhapsody displays properties that were overridden in the project.
- ♦ **Invoke SimplePlugin For OMD**, **Invoke SimplePlugin For Class**, and **Invoke SimplePlugin For Package**, for the context menus for OMDs, classes, and packages, respectively. Selecting these context menu items will display the element's name.

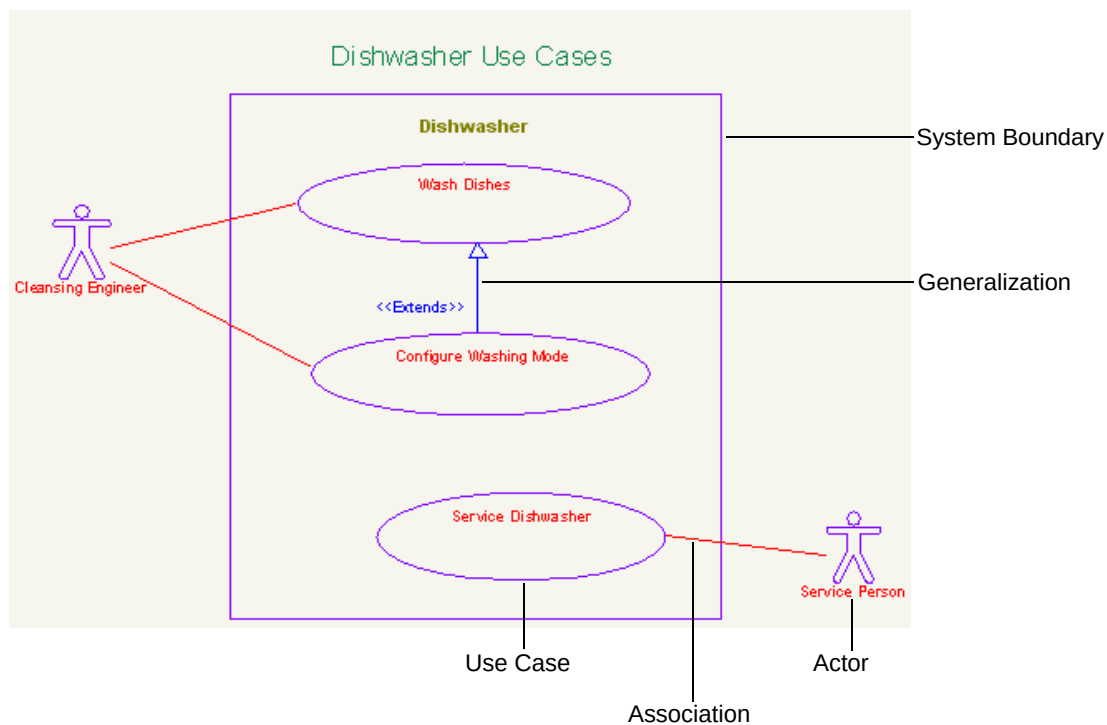
In addition, the plug-in causes messages to be displayed at the following points: before code generation, before project save, after project save, before check model.

Use case diagrams

Use case diagrams (UCDs) model relationships between one or more users (actors) and a system or class (classifier). You use them to specify requirements for system behavior. In addition, Rational Rhapsody UCDs depict generalization relationships between use cases as defined in the UML (see [Creating generalizations](#)). Rational Rhapsody does not generate code for UCDs.

Use case diagrams overview

Use cases (for example, wash dishes and service dishwasher) represent the user's expectation for a system. Actors (a cleansing engineer and service person) represent any external object that interacts with the system. Uses cases reside inside the system boundary, and actors reside outside. Association lines show relationships between the use cases and the actors. The following UCD demonstrates these features.



Opening an existing use case diagram

To open an existing use case diagram in the drawing area:

1. Double-click the diagram name in the browser.
2. Click **OK**. The diagram opens in the drawing area.

As with other Rational Rhapsody elements, use the Features window for the diagram to edit its features, including the name, stereotype, and description. For more information, see [The Features window](#).

Create use case diagram elements

The following sections describe how to use the use case diagram drawing tools to draw the parts of a use case diagram. For basic information on diagrams, including how to create, open, and delete them, see [Graphic editors](#).

Use case diagram drawing tools

The **Diagram Tools** for a use case diagram contains the following tools:

Drawing Tool	Button Name	Description
	Use Case	Draws a representation of a user-visible function. A use case can be large or small, but it must capture an important goal of a user for the system.
	Actor	Represents users of the system or external elements that either provide information to the system or use information provided by the system.
	Package	Groups systems or parts of a system into logical components.
	Association	Shows relationships between actors and use cases.
	Generalization	Shows how one use case is derived from another. The arrow head points to the parent use case.
	Dependency	Defines dependencies between an actor and a use case, between two actors, or between two use cases.
	Boundary box	Delineates the design scope for the system and its external actors with the use cases inside the system boundary and the actors outside.
	Flow	Provides a mechanism for specifying exchange of information between system elements at a high level of abstraction. This functionality enables you to describe the flow of data and commands within a system at a very early stage, before committing to a specific design. As the system specification evolves, you can refine the abstraction to relate to the concrete implementation. For detailed information on flows, see Flows and flowitems .

System boundary box

The system boundary box should be the first element placed in a UCD. It distinguishes the border between use cases and actors (use cases are inside the borders of the system boundary box and actors are outside of it).

To create a system boundary box:

1. Click the **Boundary box** button .
2. Click once in the drawing area. Rational Rhapsody creates a boundary box named System Boundary Box. Alternatively, click-and-drag with the mouse to draw the system boundary box.
3. If wanted, edit the default name and press **Enter**.

Use cases

Use cases represent the externally visible behaviors, or functional aspects, of the system. They consist of the abstract, uninterpreted interactions of the system with external entities. This means that the content of use cases is not used for code generation. A use case is displayed in a UCD as an oval containing a name.

Creating a use case

To create a use case:

1. Click the **Use Case** button .
2. Click once in the diagram or click-and-drag with the mouse to draw a use case of a specific size. By default, the use case is named usecase n , where n is an integer greater than or equal to 0.
3. If wanted, edit the default name and press **Enter**.

The new use case is displayed in both the UCD and the browser. The browser icon for a use case is an oval.

Modify the features of a use case

Use the Features window to define these use case features.

- ◆ **Name** allows you to replace the default name with the name you want for this use case.
- ◆ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ◆ **Stereotype** specifies the stereotype of the use case, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).

Note that the COM stereotypes are constructive; that is, they affect code generation.

- ◆ **Default Package** specifies a group for this use case.

Adding attributes to a use case

Because a use case is a stereotyped class, it can have attributes. No code is generated for these attributes. Rational Rhapsody does not display attributes in the UCD. To access attributes for a use case, use the browser.

To add a new attribute to a use case:

1. Select the use case in the UCD editor.
2. Right-click the use case, and then select **New Attribute**. The Attribute window opens.
3. Type a name for the attribute in the **Name** field.

Use the **L** button next to the name field to assign a logical label. For more information on labels, see [Descriptive labels for elements](#).

4. Select the **Type**, **Visibility**, and **Multiplicity** for the attribute.
5. Type a description in the **Description** tab.
6. Click **OK**.

To add an existing attribute to a use case:

1. In the browser, locate the class that contains the attribute.
2. Click-and-drag the attribute to the use case in the browser. This creates a separate copy of the attribute under the use case.

Note

If you click-and-drag an attribute from one use case to another, the attribute is moved, not copied.

Adding operations to a use case

Because a use case is a stereotyped class, it can have operations.

To add a new operation to a use case:

1. Select the use case in the UCD editor.
2. Right-click the use case, and then select **New Operation**. The Primitive Operation window opens.
3. Type a name for the operation in the **Name** field.
4. If wanted, specify a stereotype.
5. Select the visibility of the operation.
6. Select a type for the operation in the **Type** field.
7. Select the **Return Type** for the operation.
8. Specify the operation modifiers.
9. Add any arguments using the **Arguments** section.
10. Type a description in the **Description** tab.
11. Select the **Implementation** tab.
12. Type the implementation code for the operation in the **Implementation** text box.

Note: Code is not generated for the contents of use cases. This implementation is for descriptive purposes only.

13. Click **OK**.

Adding extension points

To create an extension point for the base use case:

1. Right-click the use case in a diagram.
2. Select **Features** and click **New** on the General tab.
3. Type a name for the new extension point and click **OK**.

Creating a statechart or activity diagram for a use case diagram

Because use cases are stereotyped classes, it is possible to add a statechart or an activity diagram.

To add a statechart or activity diagram to a use case:

1. Select the use case in the UCD editor.
2. Right-click the use case, and then select either **New Statechart** or **New Activity Diagram**.

For more information on these diagrams, see [Statecharts](#) and [Activity diagrams](#).

Actors

Actors are the external entities that interact with a use case. Typical actors that operate on real-time, embedded systems are buses (for example, Ethernet or MIB), sensors, motors, and switches. An actor is represented as a figure in UCDs.

Actors are a kind of UML classifier similar to classes and they can participate in sequences as instances. However, actors have the following constraints imposed on them:

- ♦ They cannot aggregate or compose any elements.
- ♦ They generalize only from other actors.
- ♦ They cannot be converted to classes, or vice versa.

Rational Rhapsody can generate code for an actor, which can be used in simulation testing of the system you are building. For more information, see [Generating Code for Actors](#).

Creating an actor

To create an actor:

1. Click the **Actor** button , and then click once in the UCD. Alternatively, click-and-drag to draw the actor.
2. Edit the default name, and then press **Enter**.

Modify the features of an actor

The Features window enables you to define the features of an actor. An actor has the following features:

- ♦ **Name** specifies the name of the element. The default name is `actor_n`, where *n* is an incremental integer starting with 0.
- ♦ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ **Stereotype** specifies the stereotype of the actor, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating labels, see [Stereotypes](#).

Note: The COM stereotypes are constructive; that is, they affect code generation.

- ♦ **Main Diagram** specifies the main diagram for the actor.

More than one UCD can contain the same use case or actor. You can select one of these diagrams to be the main diagram for the use case or actor. This is the diagram that will open when you select the **Open Main Diagram** option in the browser.

- ♦ **Concurrency** specifies the concurrency of the actor. The possible values are as follows:

- **Sequential** where the element will run with other classes on a single system thread. This means you can access this element only from one active class.
 - **Active** where the element will start its own thread and run concurrently with other active classes.
- ♦ **Defined In** specifies which element owns this actor. An actor can be owned by a package, class, or another actor.
- ♦ **Class Type** specifies the class type. The possible values are as follows:
 - **Regular** creates a regular class.
 - **Template** creates a template. To specify the necessary arguments, click the **Arguments** button.
 - **Instantiation** creates an instantiation of a template.

To create an instance of a class, select the **Instantiation** radio button and select the template that the instance is from. For example, if you have a template class A and create B as an instance of that class, this means that B is created as an instance of class A at run time.

To specify the necessary arguments, click the **Arguments** button.

Adding attributes and operations

Attributes and operations are added to actors just as they are added to classes.

To add an attribute or operation to an actor:

1. Open the Features window for the actor.
2. Select the **Attributes** tab or **Operations** tab, as appropriate.
3. Select the <New> label. A new row is displayed, with the default values filled in.
4. If needed, change the default values for the new attribute or operation.
5. Click **OK**.

For detailed information on creating attributes and operations, see [Defining the attributes of a class](#).

Creating a statechart, activity, or structure diagram

Because an actor is a special type of class, it can have a statechart, an activity diagram, or a structure diagram.

To use the editor to add one of these diagrams to an actor:

1. Select the actor in the UCD editor.
2. Right-click the actor, and then select **New Statechart**, **New Activity Diagram**, or **New Structure Diagram**.

For more information on these diagrams, see [Statecharts](#), [Activity diagrams](#), or [Structure diagrams](#).

Generating code for an actor

To generate code for an actor:

1. Locate the active configuration in the browser.
2. Open the Features window for the active configuration.
3. On the **Initialization** tab, select **Generate Code for Actors**.

Alternatively, you can generate code for the actor by right-clicking on the actor in the UCD and selecting **Generate**.

When you generate code for the configuration, code is also generated for any actors that are part of the configuration. For detailed information on configurations, see [Component configurations in the browser](#).

Creating packages

Packages logically group system components. They are represented in UCDs as a file folder.

To create a package:

1. Click the **Package** button , and then click once in the UCD. Alternatively, click-and-drag with the mouse to draw the package.
2. Edit the default name, and then press **Enter**.

The new package will be displayed in both the diagram and the browser (listed under Packages).

Creating associations

Associations represent lines of communication between actors and use cases. Use cases can associate only with actors, and vice versa.

To create an associations:

1. Click the **Association** button .
2. Click either the actor or the use case. Note that the crosshairs change to a circle with crosshairs when you are on an element that can be part of an association.
3. Move the cursor to the target of the association and click once. If the source is an actor, the target must be a use case, and vice versa.
4. Type a name for the association, then press **Enter**.

The new association is displayed in both the UCD and the browser (under the actor's **Association Ends** category).

For information on modifying an association, see [Association features](#). Association features include the type, name, roles, multiplicity, qualifiers, and description.

Creating generalizations

UML allows for generalization as a way of factoring out commonality between use cases. In other words, it provides a means to derive one use case from another. Generalizations are allowed between use cases and actors.

To create a generalization relationship:

1. Click the **Generalization** button.
2. Click the derived use case.
3. Move the cursor to the closest edge of the super-use case and click once.

Creating dependencies

A *dependency* is a directed relationship from a client to a supplier in which the functioning of a client requires the presence of the supplier, but the functioning of the supplier does not require the presence of the client. Generalizations are allowed between any two UCD elements: use case, actor, or package.

To create a dependency relationship:

1. Click the **Dependency** button .
2. Select the client element.
3. Move the cursor to the closest edge of the supplier element and click once.

You can set the dependency stereotype using the Features window. See [Dependencies](#).

Sequences

UCDs assist in the analysis phase of a project. They capture hard and firm constraints at a high level. As design decisions are made, you further decompose UCDs to create more possible use cases and scenarios, or sequences, that implement the use case. Each use case has a folder in the browser containing some of its possible sequences.

Scenarios describe not only the main path through a use case, but can also include background environmental and situational descriptions to set the stage for future events. In other words, they can provide detailed definitions of preconditions for a use case. Therefore, a sequence describes the main path through a use case, whereas a variant, represented by a child use case, describes alternate paths. For example, consider a VCR. One sequence of the InstallationAndSetup use case might be the following steps:

1. Add the VCR and accessories.
2. Insert batteries in the remote control.
3. Connect the antenna or cable system to the VCR.
4. Set the CH3/CH4 switch.
5. Turn on the VCR and select an active channel.
6. Learn to use the TV/VCR button.
7. Test the VCR connections.

The specific sequence of steps through a particular use case is better expressed through a sequence diagram. For detailed information, see [Sequence diagrams](#).

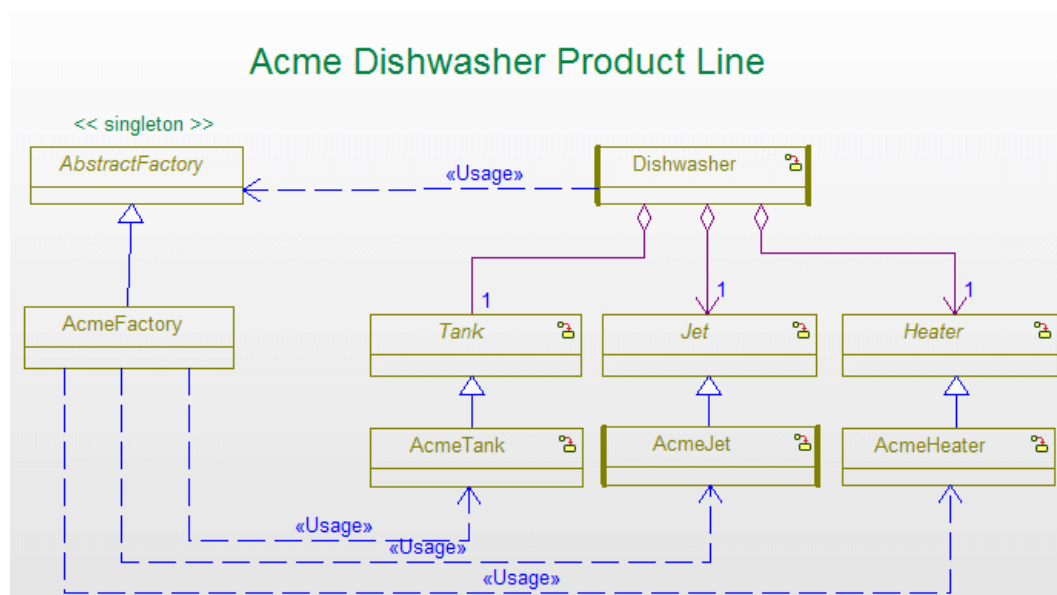
Object model diagrams

Object model diagrams (OMDs) specify the structure and static relationships of the classes in the system. Rational Rhapsody OMDs are both class diagrams and object diagrams, as specified in the UML. They show the classes, objects, interfaces, and attributes in the system and the static relationships that exist between them.

Structure diagrams focus on the objects used in the model. Although you can put classes in structure diagrams and objects in the OMD, the toolbars for the diagrams are different to allow a distinction between the specification of the system and its structure. For more information, see [Structure diagrams](#).

Object model diagrams overview

More than simply being a graphical representation of the system structure, OMDs are constructive. The Rational Rhapsody code generator directly translates the elements and relationships modeled in OMDs into source code in a number of high-level languages. The following sample OMD shows a dishwasher project.



In this diagram, the thick sidebars on the `Dishwasher` class denote that it is the active class.

You can specify and edit operations and attributes directly within class and object boxes. Simply highlight the appropriate element to make it active, then type in your changes. To open the Features window for a given attribute or operation, just double-click the element within the compartment.

Note

To add a new operation or attribute, press the **Insert** key when the appropriate compartment is active.

Object model diagram elements

The following sections describe how to use the object model diagram drawing tools to draw the parts of an object model diagram. For basic information on diagrams, including how to create, open, and delete them, see [Graphic editors](#).

Object model diagram drawing tools

The **Diagram Tools** for an object model diagram includes the following tools:

Drawing Tool	Button Name	Description
	Select	A pointing tool to identify parts of the diagram requiring changes or additions.
	Object	A structural building block of a system. Objects form a cohesive unit of state (data) and services (behavior). Every object has a public part and an private part. For more information, see Objects .
	Class	Defines properties that are common to all objects of that type. For more information, see Creating classes .
	Composite class	A container class. You can create objects and relations inside a composite class. For more information, see Creating composite classes .
	Package	A group of classes. For more information, see Creating a package .
	File	Available only in Rational Rhapsody in C has an additional icon. Use it to create file model elements. A file is a graphical representation of a header (.h) or code (.c) source file. For more information, see Files .
	Port	Draws connection points among objects and their environments.
	Inheritance	Shows the relationship between a derived class and its parent.

Drawing Tool	Button Name	Description
	Association	Creates connections that are necessary for interaction such as messages.
	Directed association	Indicates the only object that can send messages to another object. For more information, see Directed associations .
	Aggregation	Specifies an association between an aggregate (whole) and a component part. For more information, see Aggregation associations .
	Composition	Defines a class that contains another part class. For more information, see Composition associations .
	Link	Creates an association between the base classes of two different objects. For more information, see Links .
	Dependency	Creates a relationship in which the proper functioning of one element requires information provided by another element. For more information, see Dependencies .
	Flow	Specifies the flow of data and commands within a system. For more information, see Flows and flowitems .
	Realization	Specifies a realization relationship between an interface and a class that implements that interface. For more information, see Realization .
	Interface	Adds a set of operations that publicly define a behavior or way of handling something so knowledge of the internals is not needed.
	Actor	Represents an element that is external to the system. For more information, see Actors .

The following sections describe how to use these tools to draw the parts of an OMD. For basic information on diagrams including how to create, open, and delete them, see [Graphic editors](#).

Objects

Rational Rhapsody separates objects from classes in diagrams. There are two types of objects:



Objects with explicit object types specifies only the features that are relevant for the instance. An explicit object instantiates a “normal” class from the model.



Objects with implicit types enables you to specify other features that belong to classes, such as attributes, operations, and so on. An implicit object is a combination of an instance and a class. Technically, the class is hidden.

Note that Rational Rhapsody in J does not support objects with implicit types.

An object is basically an instance of a class; however, you can create an object directly without defining a class. Objects belong to packages and *parts* belong to structured classes; the browser separates parts and objects into separate categories.

Opening an existing object model diagram

To open an existing object model diagram in the drawing area:

1. Double-click the diagram name in the browser.
2. Click **OK**. The diagram opens in the drawing area.

As with other Rational Rhapsody elements, use the Features window for the diagram to edit its features, including the name, stereotype, and description. For more information, see [The Features window](#).

Creating an object

To create an object:

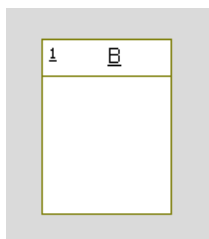
1. Click the **Object** icon in the **Diagram Tools**.
2. Click, or click-and-drag, in the drawing area.
3. Edit the default name, then press **Enter**.

If you specify the name in the format <ObjectName:ClassName> (for an object with explicit type) and the class <ClassName> exists in the model, the new object will reference it. If it does not exist, Rational Rhapsody prompts you to create it.

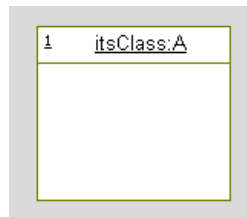
Object characteristics

By default, Rational Rhapsody creates objects with implicit type. In the OMD, an object is shown like a class box, with the following differences:

- ♦ The name of the object is underlined.
- ♦ The multiplicity is displayed in the upper, left-hand corner.



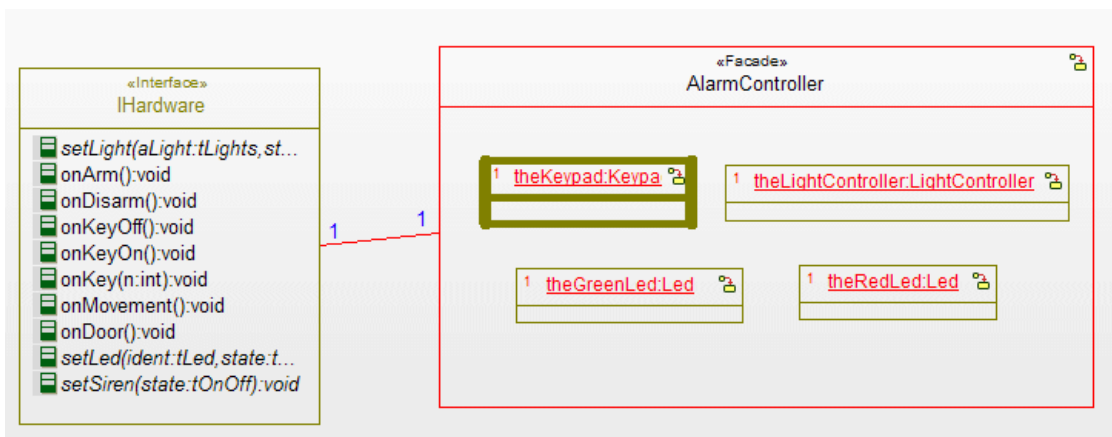
The following example shows an object of explicit type:



Parts in an object model diagram

As with classes, you can display the attributes and operations in the object. For more information, see [Display option settings](#).

The following example shows an object model diagram that contains parts.



Object features

The Features window enables you to change the features of an object, including its concurrency and multiplicity.

An object has the following features:

- ♦ **Name** specifies the name of the element. The default name is `object_n`, where `n` is an incremental integer starting with 0.
- ♦ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ **Stereotype** specifies the stereotype of the object, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).

Note: The COM stereotypes are constructive; that is, they affect code generation.

- ♦ **Main Diagram** specifies the name of the diagram of which this is a part.
- ♦ **Concurrency** specifies the concurrency of the object. This field is available only for objects with implicit type. The possible values are as follows:
 - **Sequential** means the element will run with other classes on a single system thread. This means you can access this element only from one active class.
 - **Active** means the element will start its own thread and run concurrently with other active classes.
- ♦ **Type** specifies the class of which the object is an instance. To view the classes for that class, click the Invoke Feature Dialog button next to the **Type** field.

In addition to the names of all the instantiated classes in the model, this list includes the following choices:

- **<Implicit>** specifies an implicit object
 - **<Explicit>** specifies an explicit object
 - **<New>** enables you to specify a new class
 - **<Select>** enables you to browse for a class using the selection tree
- ♦ **Multiplicity** specifies the number of occurrences of this instance in the project. Common values are one (1), zero or one (0,1), or one or more (1..*).
- ♦ **Initialization** specifies the constructor being called when the object is created. If you click the Ellipsis button, the Actual Call window opens so you can see the details of the call.

If the part does not have a constructor, with parameters, this field is dimmed.

- ♦ **Relation to whole** enables you to name the relation for a part. If the object is part of a composite class, enable the **Knows its whole as** check box and type a name for the relation in the text box. This relation is displayed in the browser under the Association

Ends category under the instantiated class or implicit object.

If the **Relation to whole** field is specified on the **General** tab, the Features window includes tabs to define that relation and its properties. However, on the tab that specifies the features of its whole (in the illustration of the `itsController` tab), only the fields **Name**, **Label**, **Stereotype**, and **Description** can be modified.

Converting object types

You can easily change the type of an object using the Features window for the object.

If you convert an object with implicit type to an object with explicit type (by selecting <Explicit> in the **Type** field), a new class is created. By default, the name of the new class is <object name>_class.

If you convert an object of explicit type to an object of implicit type, the following actions occur:

- ♦ The original class is copied into the object of implicit type.
- ♦ Graphical relations are removed.
- ♦ Symmetric associations become directional.
- ♦ Links disconnect from associations.

Converting classes to objects

To convert a class to an object, right-click the class in the diagram and select **Make an Object**. Rational Rhapsody converts the class to an object named `its<ClassName>:<ClassName>`. For example, if you converted class A to an object, the name of the object would be `itsA:A`.

Code generation for objects

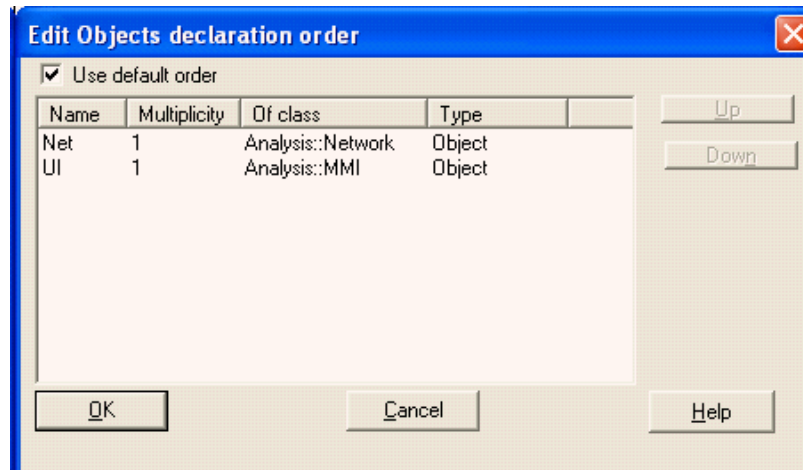
For objects with explicit type, code is generated as in previous versions of Rational Rhapsody. The following table lists the results of generating code for objects with implicit type.

Situation	Results of Code Generation
Implicit type	During code generation, the object is mapped in two parts: • An implicit class with the name <code><object>_C</code>. • The instance of the class in its owner (either a composite class or package). The name of the instance is <code><object></code>.
Implicit type in a package (global)	The code for the instance is generated in the package file and the code for the implicit class is generated into files named <code><object>.h</code> and <code><object>.cpp</code> .
Implicit type in a structured class (part)	The code for the instance is generated in the composite class file and the code for the implicit class is generated as a nested class of the composite (in the file for the composite).
Embeddable objects	The default code scheme for code generation for objects is changed to embeddable. The default values of the following properties were changed: • <code>CPP_CG::Class::Embeddable</code> is Checked • <code>CPP_CG::Relation::ImplementWithStaticArray</code> is FixedAndBounded

Editing the declaration order of objects

To change the order of objects:

1. In the browser, right-click the Objects category icon and then select **Edit Objects Order**. The Edit Objects declaration order window opens and lists all the files and objects in the current package.



2. Unselect the **Use default order** check box.
3. Select the object you want to move.
4. Click **Up** to generate the object earlier or **Down** to generate it later.
5. Click **OK**.

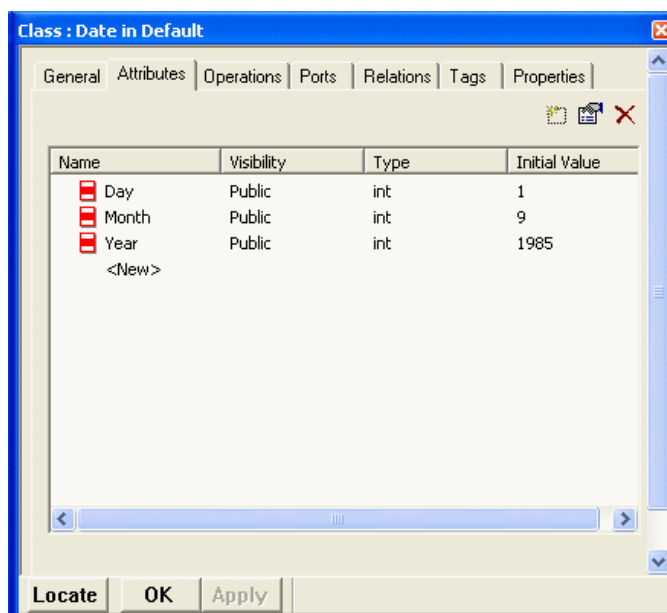
Changing the value of an instance

You can specify the value of attributes for instances. An *attribute value* is the value assigned to an attribute during run time. This functionality enables you to describe a possible setup of objects and parts at a certain point in their lifecycle, you can see a “snapshot” of the system, including the instances that exist and their values. To support this functionality, the Features window for instances includes a new column, **Value**.

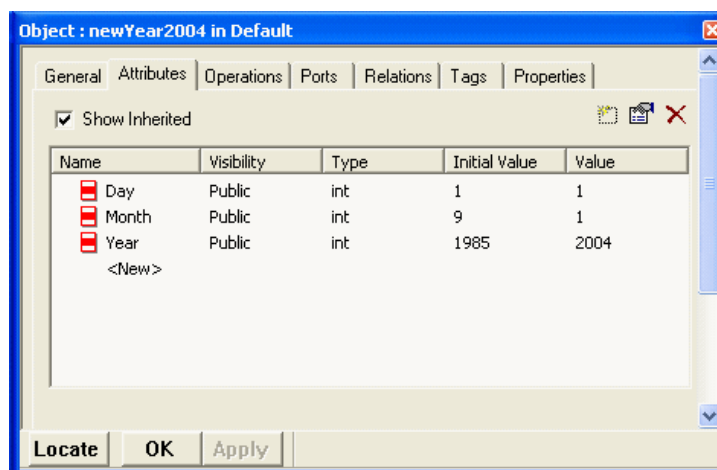
Note

Initial values are features of the attributes of the *class*, whereas instance values characterize the specific *instance* of the class (that object).

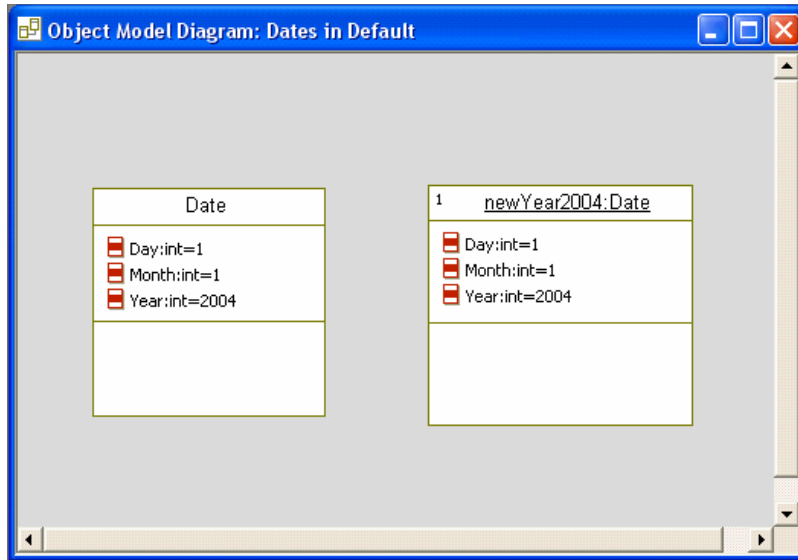
For example, consider the class, Date, and an object of Date called newYear2004. The class Date has the attributes Day, Month, and Year. The following example shows the initial values for the class Date.



The following example shows the attributes for object newYear2004 of class Date. Note that the **Show Inherited** check box specifies whether to display the inherited attributes so you can easily modify them.



Click the **Specification View** icon to view the attributes and operations of an object in the OMD. The following OMD shows the class, Date, and the object of Date called newYear2004.



In the OMD, the object values are displayed using the following format:

[visibility]<attribute name>:<attribute type>=<value>

Note the following information:

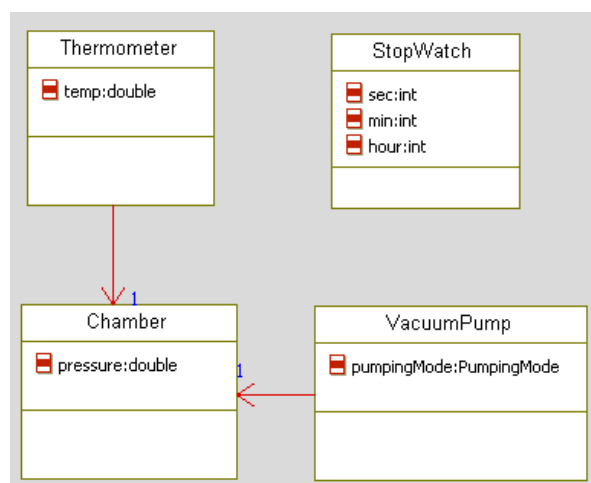
- ♦ Instance values are always displayed. To hide the entire attribute, right-click the object and select **Display Options**.
- ♦ Both the instance and the class must be “available” in the model.

Creating a vacuum pump model as an example

The following model shows how to use instance attribute values to take snapshots of a vacuum pump model at different stages in the lifecycle.

The vacuum pump removes the air from a chamber. The model needs to show the state of the system at various points in time, the initial value, the value after one hour, and the final value.

The following example shows the OMD for the model.



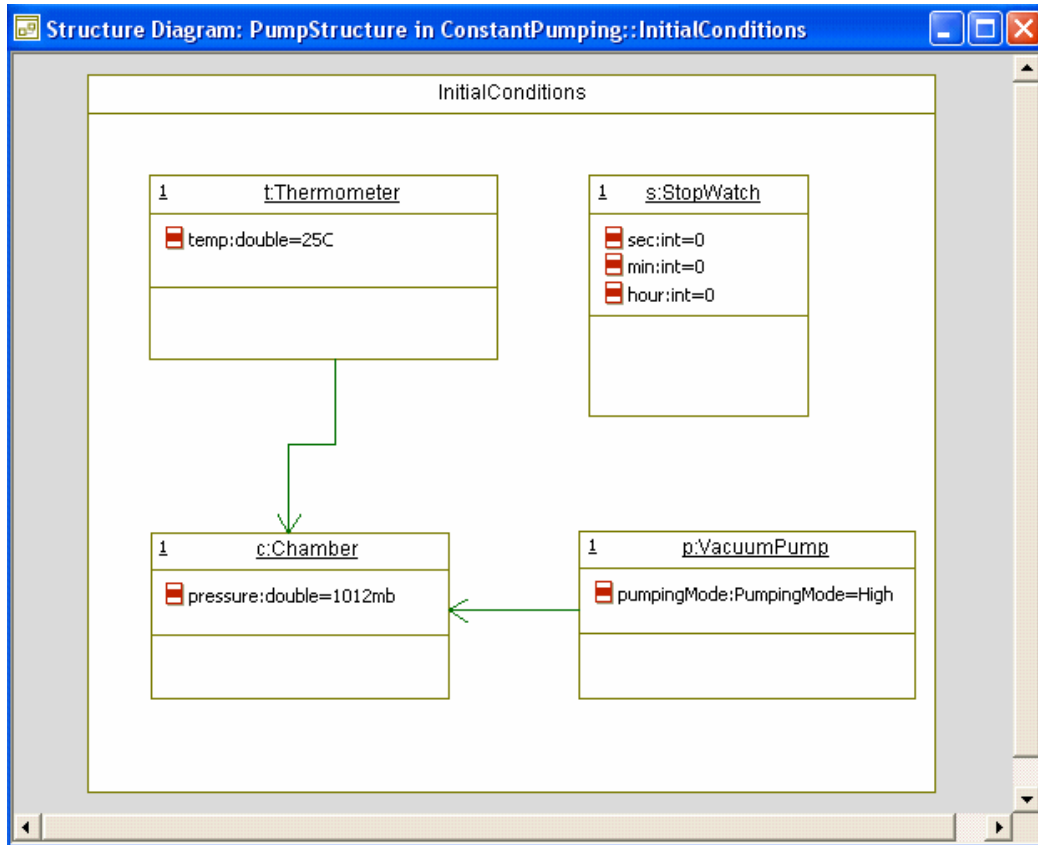
To create a vacuum pump model as an example:

1. Create a package called ConstantPumping.
2. Set the CG::Class::UseAsExternal property to Checked so the package is considered external (and code will not be generated for it).

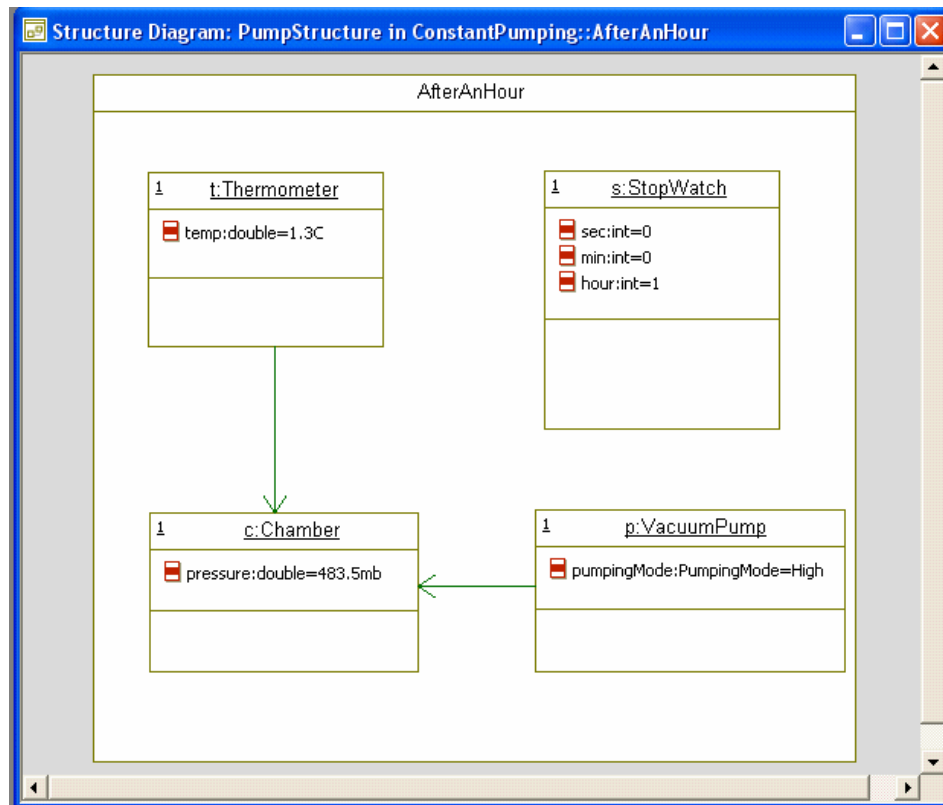
Alternatively, you can create a new stereotype for the class («snapshot»), then set this property to Checked.

3. In this package, each phase is represented by a different class. For the initial conditions, create a class called InitialConditions.

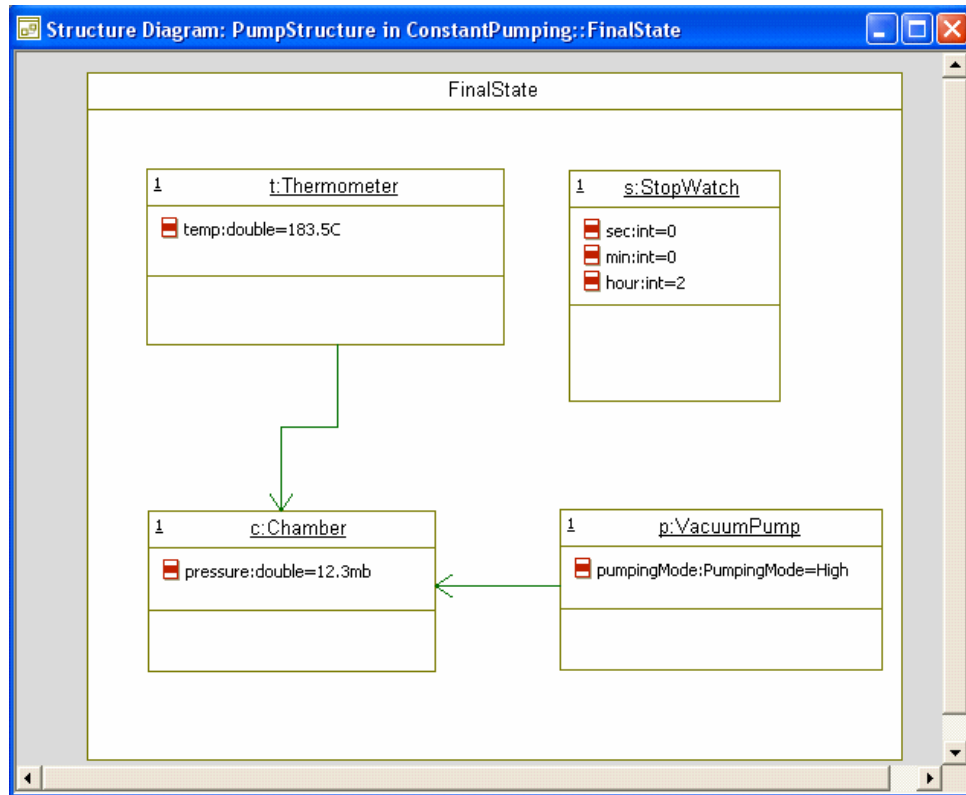
4. Add a structure diagram to InitialConditions and add the elements (and their attribute values) to the diagram.



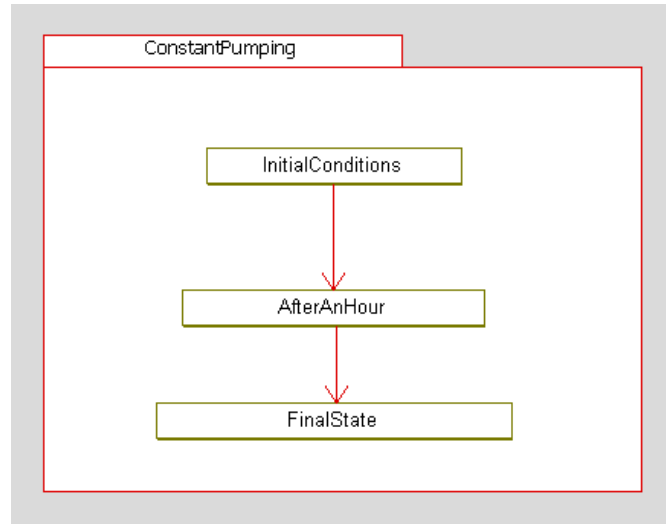
5. To show the conditions after an hour, copy the InitialConditions class and rename it AfterAnHour. Specify the attribute values for this stage in the process. The following example shows the attribute values after the pump has been running for an hour.



6. To show the final values for the system, copy the InitialConditions class and rename it FinalState. Specify the attribute values for this stage in the process. The following example shows the final values.



7. To show the order and transitions between snapshots, you can draw a simple OMD, as shown in the following example.



Creating classes

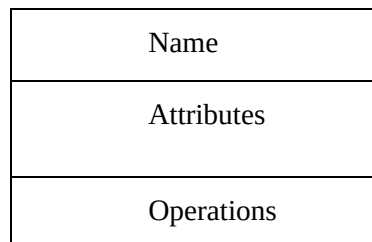
Classes can contain attributes, operations, event receptions, relations, components, superclasses, types, actors, use cases, diagrams, and other classes. The browser icon for a class is a three-compartment box with the top, or name, compartment filled in.

To create a class:

1. In the **Diagram Tools**, click the **Class** tool.
2. Click in the drawing area.
3. Edit the default class name.
4. Press **Enter**.

Class compartments

In the OMD, a class is shown as a rectangle with three sections, for the name, attributes, and operations. You can select and move the line separating the attributes and operations to create more space for either compartment.



If you shrink the box vertically, the operations and attributes sections disappear and the class graphic shows only the class name. The attributes and operations reappear if you enlarge the drawing.

When you rename a class in the OMD editor, the class name is changed throughout the model.

For more information about classes, see [Classes and types](#).

Creating composite classes

Instances in a composite class are called *parts*. To identify a component in code (actions or operations), use the expression *instance-of-composite.name-of-part*. The multiplicity of a component is relative to each instance of the composite containing it. For example, each car has one engine.

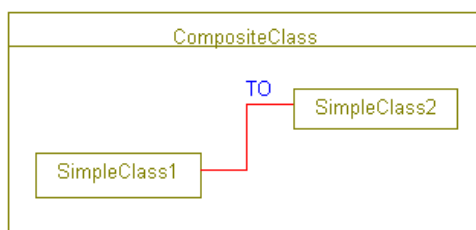
If the multiplicity is well-defined (such as 1 or 5), Rational Rhapsody creates the components at run time, when the composite is instantiated. If an association is instantiated by a link, Rational Rhapsody initializes the association at run time.

When a composite is destroyed, it destroys all its components.

To create a composite class:

1. Click the **Composite Class** tool.
2. Click in the diagram, or click-and-drag to create the composite class. The new composite class is displayed in the diagram.

Because a composite class is a container class, you can create objects and relations inside it, as shown in this example.



A composite class uses the same Features window as objects and parts (see [Class features](#)).

Another way of having the functionality of a composite class is to use a *composition*. For more information, see [Composition associations](#).

Creating a package

In Rational Rhapsody, every class belongs to a package. Classes drawn explicitly in a package are placed in that package in the model. Classes not drawn explicitly in a package are placed in the default package of the diagram. If you move a class to a package, it is also placed in that package in the model. If you do not connect this diagram to a package with the browser, Rational Rhapsody assigns the diagram to the default package of the model.

To draw a package in the diagram:

1. Select the **Package**  icon.
2. Click once in the diagram. Now you must define the package using the Features window.
3. Right-click the package and select **Features**.

Package features

The Features window allows you to define the characteristics of a package, such as its Name or Main Diagram.

- ♦ **Name** specifies the name of the package. Package names cannot contain spaces or begin with numbers.
- ♦ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ **Stereotype** specifies the stereotype of the package, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).

Note that package stereotypes are not constructive (see [Constructive dependencies](#)).

- ♦ **Main Diagram** specifies the name of the diagram of which this is a component.

The **Description** tab allows you to write a detailed description of the package. The **Relations** tab lists all of the relationships of the package. The **Tags** tab lists the available tags for this package. The **Properties** tab enables you to define code generation properties for the package.

Inheritance

Inheritance is the “mechanism by which more specific elements incorporate structure and behavior of more general elements related by behavior.” Inheritance is also known as generalization, or a “taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. An instance of the more specific element may be used where the more general element is allowed.” (Both references are from the UML Specification, v1.3.)

Creating an inheritance with an inheritance arrow

You can create inheritance by drawing an inheritance arrow between two classes or by using the browser. To create an inheritance arrow between two classes:

1. Click the **Inheritance**  icon.
2. Click in the subclass.
3. Move the cursor to the superclass and click once to end the arrow.

An inheritance arrow points from the subclass to the superclass, with a large arrowhead on the superclass end.

The browser icon for a superclass is an inheritance arrow:

- ♦ The icon for the SuperClass category is a black arrow.
- ♦ The icon for an individual SuperClass is a blue arrow.

Double-clicking a SuperClass icon in the browser opens the Features window for the superclass.

Creating inheritance in the browser

To create inheritance using the browser:

1. Right-click a class and then select **Add New > SuperClass**. The Add Superclass window opens.
2. Use the list to specify the superclass.
3. Click **OK**.

Inheriting from an external class

To inherit from a class that is not part of the model, set the `CG::Class::UseAsExternal` property for the superclass to `Checked`. This prevents code from being generated for the superclass.

To generate an `#include` of the superclass header file in the subclass, do one of the following actions:

- ◆ Add the external element to the scope of some component.
- ◆ Map the external element to a file in the component.
- ◆ Set the `CG::Class::FileName` property for the superclass to the name of its specification file (for example, `super.h`). That file is included in the source files for classes that have relations to it. If the `FileName` property is not set, no `#include` is generated.

Another way to inherit from an external class is to exclude the external class from the code generation scope. For example, if you want a class to extend the Java class `javax.swing.JTree` without actually importing it:

1. Draw a package `javax`.
2. Draw a nested package `swing` inside `javax`.
3. Draw a class `JTree` inside the `swing` package.
4. Exclude the `javax` package from the component (do not make it one of the selected elements in the browser). This prevents the component from generating code for anything in the `javax` package.

This gives the rest of the model the ability to reference the `JTree` class without generating code for it. In this way, a class in the model (for example, `MyJTree`) can inherit from `javax.swing.JTree`. If the subclass is public, the generated code is as follows:

```
import javax.swing.JTree;
...
public class MyJTree extends JTree {
...
}
```

If you need a class to import an entire package instead of a specific class, add a dependency (see [Dependencies](#)) with a stereotype of «Usage» to the external package, in this case `javax.swing`. The generated file will then include the following line:

```
import javax.swing.*
```

For more information on using external elements, see [External elements](#).

Realization

Rational Rhapsody allows you to specify a realization relationship between an interface and a class that implements that interface. This type of relationship is specified using the realization connector in the Diagram Tools for object model diagrams.

Note

Realization is a "new term" based on the generalization element. This means that it is also possible to right-click a generalization element in a diagram and then select **Change To > Realization**.

The realization connector only serves a visual purpose when used in an object model diagram. The code generation for realization relationships is not determined by the connector used between the class and the interface, but by the application of the *interface* stereotype to the class element in the diagram that represents the interface.

If you apply the *interface* stereotype to a class element, then the appropriate code will be generated for the interface and the implementing classes in Rational Rhapsody in Java and Rational Rhapsody in C.

For details regarding the code generation for realization relationships in C, see [Components-based Development in C](#).

Associations

In previous versions of Rational Rhapsody, the term “relations” referred to all the different kinds of associations. Note that the term “relations” refers to all the relationships you can define between elements in the model (not just classes), associations, dependencies, generalization, flows, and links.

Associations are links that allow related objects to communicate. Rational Rhapsody supports the following types of associations:

- ♦ **Bi-directional association** where both objects can send messages back and forth. This is also called a *symmetric association*. For more information, see [Bi-directional associations](#).
- ♦ **Directed association** where only one of the objects can send messages to the other. For more information, see [Directed associations](#).
- ♦ **Aggregation association** defines a whole-part relationship. For more information, see [Aggregation associations](#).
- ♦ **Composition aggregation** defines a relationship where one class fully contains the other. For more information, see [Composition associations](#).

Bi-directional associations

Bi-directional (or *symmetric*) *associations* are the simplest way that two classes can relate to each other. A bi-directional association is shown as a line between two classes, and can contain control points. The classes can be any mix of classes, simple classes, or composite classes.

When you create an association or an aggregation association between two classes and give it a role name that already exists, you have created another view of an existing relation.

In previous versions of Rational Rhapsody, an association was described by one or two association ends. An association can be composed of the following elements:

- ♦ **Association ends** means the associated objects
- ♦ **Association element** means a view of the association as a whole
- ♦ **Association class** means an association element that has class characteristics (attributes and operations)

Note

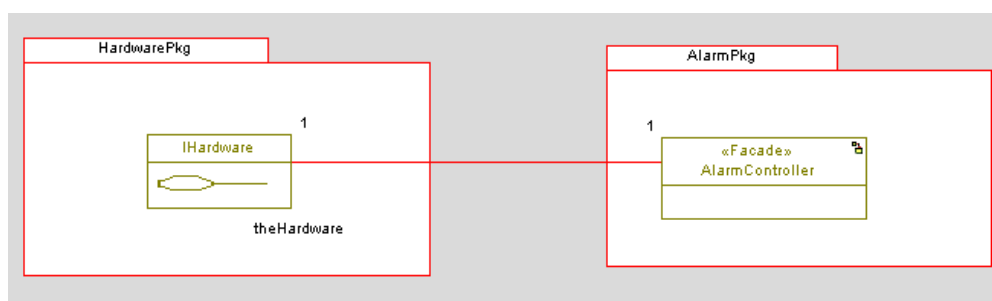
If you draw an anchor from a class to a relation (association, aggregation, or composition), it semantically implies that the class is an association class for this relation. Removing the icon changes the association class into a regular class.

Creating a bi-directional association

To create a bi-directional association between classes:

1. Click the **Create Association** icon .
2. Click in a class.
3. Click in another class.

In this example note the bi-directional Association line between two classes.

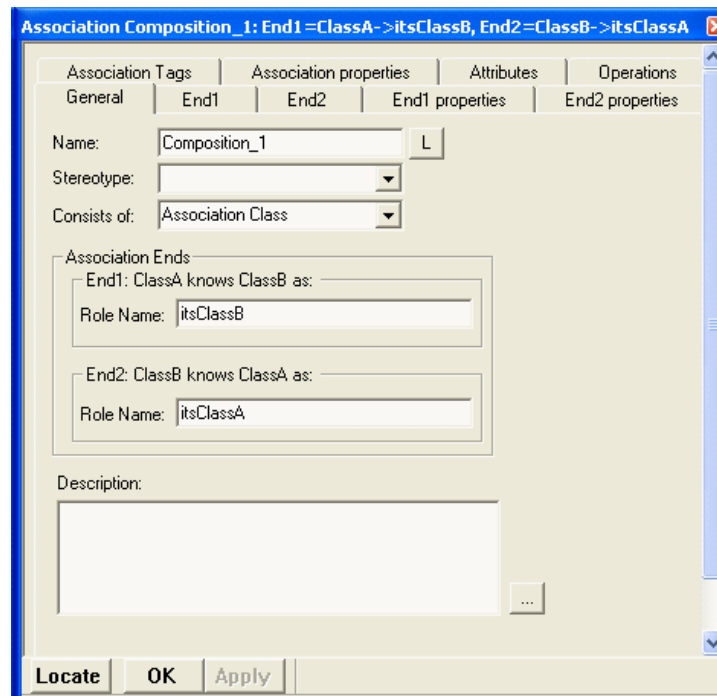


Note the following information:

- ◆ Associations specify how classes relate to each other with role names. The relative numbers of objects participating is shown with multiplicity.
- ◆ You can move an association name freely.
- ◆ If you remove the class at one end of an association from the view, the association is also removed from the view. If you delete a class at one end of an association from the model, the association is also deleted.
- ◆ The role names and multiplicity are set in the Features window for the association. To edit a role name or multiplicity, double-click it.
- ◆ If you move an association line from between class x and class y to between class x and class z, where z is a subclass of y, it is removed from y. But if z is a superclass of y, it remains because all relationships with a superclass are shared by their subclasses. If z and y are independent, Rational Rhapsody moves it from y to z.

Association features

The Features window enables you to change the features of an association, such as what it consists of (for example, two ends or a single end) and its association ends. The following figure shows the Features window for a bi-directional association.



A bi-directional association has the following features:

- ◆ **Name** specifies the name of the association.
- ◆ **L** specifies the label for the element, if any. Rational Rhapsody For information on creating labels, see [Descriptive labels for elements](#).
- ◆ **Stereotype** specifies the stereotype of the association, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).

Note: The COM stereotypes are constructive; that is, they affect code generation.

- ♦ **Consists of** specifies whether the association consists of:
 - A single association end (**End <X>**)
 - Two given ends (**Both Ends**)
 - An association element and two association ends (**Association Element**)
 - An association class and two association ends (**Association Class**)

Note: There is no representation of the association class in the diagram, nor is there code generation for the association class. The only representation of the association class is in the **Consists of** field.

- ♦ **Association Ends** specifies the ends of the association. If only one end is specified, the **Role Name** field for End2 field is unavailable.

Using this group box, you can change the role name of each enabled end. An *enabled end* is an end that is part of the specification of the association. The label under this field contains the type of the association end (the class to which the end is connected), the navigability of the end, and its aggregation kind. For a non-existing end, this label contains only “Role of.”

- ♦ **Description** describes the association. This field can include a hyperlink. For more information, see [Hyperlinks](#).

Note

If the association class or element does not exist, the **Name**, **Stereotype**, **Label**, and **Description** fields are disabled.

In addition to the **General** tab, the Features window for an association contains the following tabs:

- ♦ **End1 or End2**
- ♦ **End1 properties or End2 properties**

If the **Consists of** field is set to **Association Class**, the window also includes tabs for attributes and operations, as shown in this example.

Association Composition_1: End1=Class3->itsClass4, End2=Class4->itsClass3

Association Tags	Association properties	Attributes	Operations
General	End1	End2	End1 properties End2 properties

Name: L

Stereotype:

Consists of:

Association Ends

End1: Class3 knows Class4 as:

End2: Class4 knows Class3 as:

Description:

Locate OK Apply

The End1 and End2 tabs

The **End1** and **End2** tabs enable you to specify features of the individual ends of the association. The following figure shows an **End1** tab.

The **End1** and **End2** tabs contain the following fields:

- ◆ **Name** specifies the name of the element.
- ◆ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ◆ **Stereotype** specifies the stereotype of the element, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).
Note: The COM stereotypes are constructive; that is, they affect code generation.
- ◆ **Role of** a read-only field that specifies the class, actor, or use case that plays a role in the association.
- ◆ **Multiplicity** specifies the number of occurrences of this instance in the project. Common values are one (1), zero or one (0,1), or one or more (1..*).

- ♦ **Qualifier** shows the attributes in the related class that could function as qualifiers.

A *qualifier* is an attribute that can be used to distinguish one object from another. For example, a PIN number can serve as a qualifier on a BankCard class. If a class is associated with many objects of another class, you can select a qualifier to differentiate individual objects. The qualifier becomes an index into the multiple relation. Adding a qualifier makes the relation a qualified association.

- ♦ **Aggregation Kind** specifies the type of aggregation. The possible values are as follows:
 - **None** means no aggregation.
 - **Shared** (empty diamond) means shared aggregation (whole/part relationship).
 - **Composition** (filled diamond) means composition relationship. The instances of the class at this end contains instances of the class at the other end as a part. This part cannot be contained by other instances.
- ♦ **Navigable** specifies whether the association allows access to the other class. Both ends of a bi-directional association are navigable. In a directed association, the element that has the arrow head is navigable; the other end is not. For more information, see [Directed associations](#).

The navigability of an association influences the appearance of the association arrow in the diagram. If one end of a symmetric association is navigable and the other is not, the association line includes an arrow head.

- ♦ **Description** describes the association. This field can include a hyperlink. For more information, see [Hyperlinks](#).

Similarly, the **End2** tab shows the features for the second end of the association.

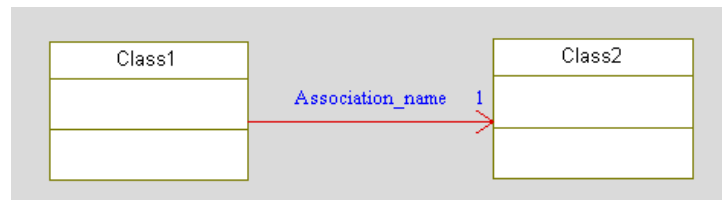
Note that if an end is read only, its feature fields are also read-only.

The End1 and End2 properties tabs

The **End1 properties** and **End2 properties** tabs enable you to specify properties for the individual ends of the association.

Directed associations

In a *directed* (or *one-way*) association, only one of the objects can send messages to the other. It is shown with an arrow, as shown in this example.



Creating a directed association

To create a directed association:

1. Click the **Directed Association** icon .
2. Click in the source class.
3. Click in the destination class.

Rational Rhapsody draws an arrow between the two objects, with the arrow head pointing to the target object.

Directed association features

The Features window enables you to change the features of a directed association, such as what it consists of (for example, two ends or a single end) and its association ends.

The Features window for a directed association is the same as the Features window for a bi-directional association (see [Association features](#)), but the available tabs are different. As shown in the figure, a directed association has one role name and one multiplicity.

If the directed association is not named (as shown in the figure), the **Consists of** field is set to **End <X>** and the window contains only the tabs **General**, **End1**, and **End1 properties**.

However, for a named directed association, the **Consists of** field of the Features window is set to **Association Element**, which means there is one more non-navigable end. The window contains the following additional tabs:

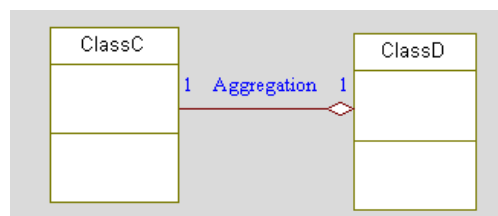
- ♦ **Association Tags** specifies the tags that can be applied to this association. For more information on tags, see [Profiles](#).
- ♦ **Association properties** specifies the properties that affect this association.

Aggregation associations

Associations and aggregation associations are similar in usage. An association portrays a general relationship between two classes; an *aggregation association* shows a whole-part relationship. When you create an association or an aggregation association between two classes and give it a role name that already exists, you have created another view of the existing association.

An aggregation association is a whole-part relationship similar to the relationship between a composite class and a part. Other than their graphic representations, these differ mainly in that the composite/component relationship implies a whole lifetime dependency. Parts are created with their composites and are destroyed with them.

An aggregation association is shown as a line with a diamond on one end. The side with the diamond indicates the whole class, whereas the side with the line is the part class. In the following sample Aggregation Association, the diamond is placed at the first point of the aggregation:



Creating an aggregation association

To create an aggregation:

1. Click the **Aggregation** icon .
2. Click in the class that represents the whole.
3. Click in the class the represents the part.

Aggregation association features

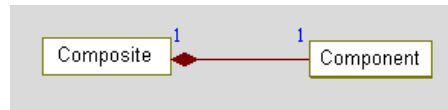
The Features window for an aggregation is the same as that for a bi-directional association. For more information, see [Association features](#).

Note that for an aggregation association:

- ♦ Both ends are navigable.
- ♦ The **Aggregation Kind** value for the diamond end of an aggregation association is set to **None**; the other end must be set to **Shared**.

Composition associations

Another way of drawing a composite class is to use a composition association. A *composition association* is a strong aggregation relation connecting a composite class to a part class (component). The notation is a solid diamond at the composite end of the relationship, as shown in this example.



The composite class has the sole responsibility of memory management for its part classes. As a result, a part class can be included in only one composite class at a time. A composition can contain both classes and associations.

Creating a composite association

To create a composite association:

1. Click the **Composition** icon .
2. Click the composite class.
3. Click the part class.
4. If wanted, name the composition.
5. Press **Enter** to dismiss the text box.

Composition association features

The Features window for compositions is the same as that for associations. For more information about the available fields, see [Association features](#).

Note that for a composition association:

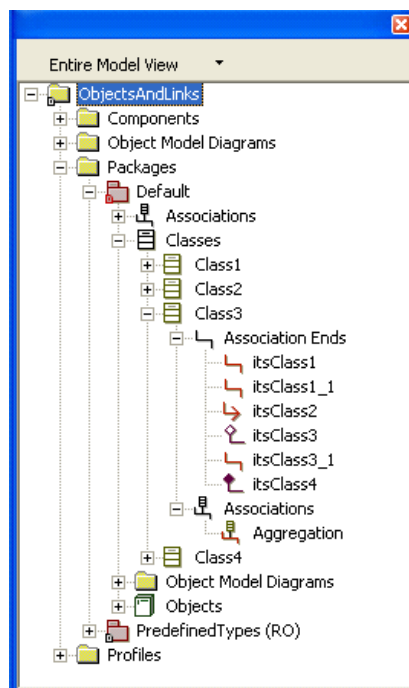
- ♦ Both ends are navigable.
- ♦ The **Aggregation Kind** value for the filled-diamond end of the composition line is set to **None**; the other end must be set to **Composition**.

Associations in the browser

An association can be represented in the browser as:

- ◆ A single association end
- ◆ Two association ends
- ◆ An association element and two association ends
- ◆ An association class and two association ends

Associations are listed in the browser under the category Association Ends under the owning class, as shown in this example. Note that the browser display includes separate icons for each association type.



Associations implementation

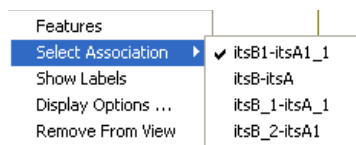
Rational Rhapsody implements associations using containers. *Containers* are objects that store collections of other objects. To properly generate code for associations, you must specify the container set and type within that set you are using for the association.

Generally, you use the same container set for all associations. You specify the container set using the `CG::Configuration::ContainerSet` property. There are many options, depending on the language you are using. You can assign various container types, as defined in the selected container set, to specific relations. Container types include `Fixed`, `StaticArray`, `BoundedOrdered`, `BoundedUnordered`, `UnboundedOrdered`, `UnboundedUnordered`, and `Qualified`, among others. In addition, you can define your own container type called `User`. You specify the container type using the `Implementation` and `ImplementWithStaticArray` properties (under `CG::Relation`).

Associations menu

In addition to the common operations (see [Edit elements](#)), the menu for associations includes the following options:

- ◆ **Select Association** lists the available associations for this class, as shown in this example. This functionality is useful when you have more than one association between the same elements.



For more information, see [Select associations](#).

- ◆ **Show Labels** shows the role labels in the diagram.
- ◆ **Display Options** enables you to specify how associations are displayed in the diagram. By default, Rational Rhapsody displays the name of the association, and the multiplicity and qualifiers for End1 and End2.

Select associations

In OMDs, you can select associations for classes that have more than one association defined between the same two classes. To do this, hold down the right mouse button over an association line to bring up the menu, then select **Select Association**.

Association names are displayed as follows:

- ◆ If the association has a name, the name is listed.
- ◆ If the association does not have a name and it is symmetric, the identifier uses the format `<role_1>-<role_2>`.
- ◆ If the association does not have a name and it is unidirectional, the identifier uses the format `-><role>`.
- ◆ If you select a different association from the list, the association line is directed to reference the selected association.

Links

A *link* is an instance of an association. In previous releases of Rational Rhapsody, you could link objects in the model only if there were an explicit relationship between their corresponding classes. An association line between two objects meant the existence of two different model elements:

- ◆ An association between the objects' classes
- ◆ A link between the objects

Rational Rhapsody separates links from associations so you can have unambiguous model elements for links with a distinct notation in the diagrams. This separation enables you to:

- ◆ Specify links without having to specify the association being instantiated by the link.
- ◆ Specify features of links that are not mapped to an association.

In addition, Rational Rhapsody supports links across packages, including code generation. To support this functionality, the default value of the property `CG::Component::InitializationScheme` was changed to `ByComponent`.

Creating a link

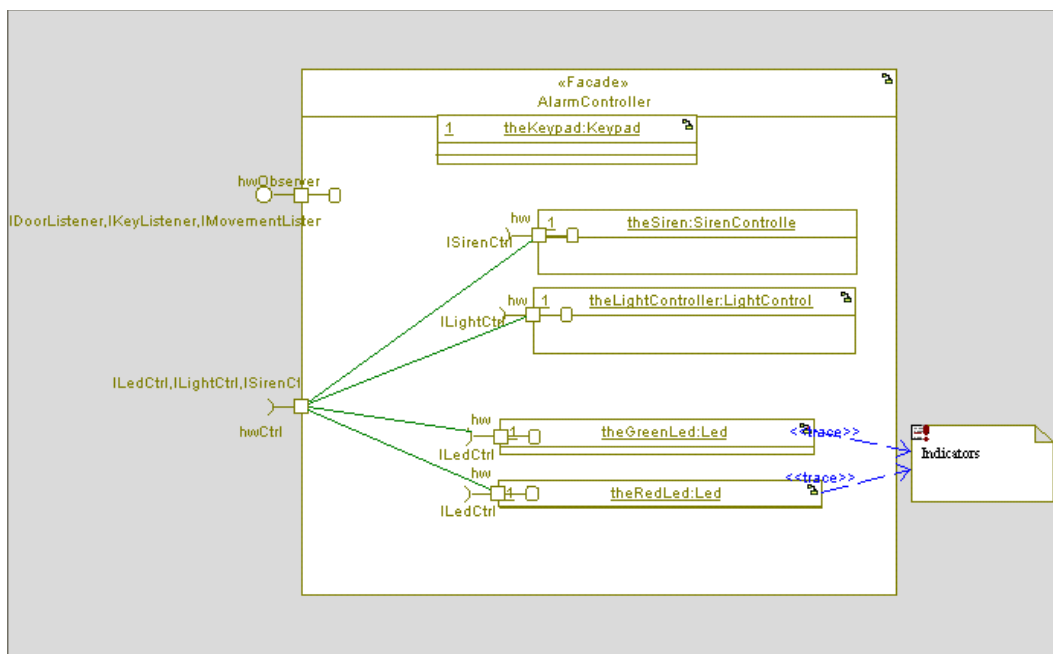
To create a link, there must be at least one association that connects one of the base classes of the type of one of the objects to a base class of the type of the second object.

To create a link:

1. Click the **Link** tool.
2. Click the first object.
3. Click the second object.
4. If wanted, name the link and press **Enter**.

The new link is created in the diagram, and is displayed in the browser under the `Link` category. Note that you can drag-and-drop links in the browser to other classes and packages as needed; however, you cannot create links in the browser.

The following figure shows links in an OMD. Note that links shown in dark green to distinguish them from associations, which are drawn in red. In addition, the names and multiplicity of links are underlined.



By default, the role name and multiplicity of a link are not displayed. Right-click the link and select **Display Options** to select the items you want to display. For more information, see [Link menu](#).

Note the following behavior:

- ♦ Links can be drawn between two objects or ports that belong to objects. One exception is the case when a link is drawn between a port that belongs to a composite class and its part (or a port of its part).
- ♦ When drawing a link, Rational Rhapsody finds the association that can be instantiated by the newly drawn link and automatically maps the link to instantiate the association.
- ♦ If you draw a link between two objects with implicit type and there no associations to instantiate, Rational Rhapsody automatically creates a new, symmetric association.

Link features

The Features window enables you to change the features of a link, such as the association being instantiated by the link.

The title bar is in the form:

Link: [end1 instance](end1 role name)-[end2 instance] (end2 role name)

For example, a(itsA)-b(itsB).

A link has the following features:

- ♦ **Name** specifies the name of the link.
- ♦ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ **Stereotype** specifies the stereotype of the link, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).

Note: The COM stereotypes are constructive; that is, they affect code generation.

- ♦ **Association** specifies the association being instantiated by the link.

Rational Rhapsody allows you to specify a link without having to specify the association being instantiated by the link. Until you specify the association, this field is set to <Unspecified>.

If you select <New> from the list, Rational Rhapsody creates a new, symmetric association based on the data (its name and multiplicity) for the link. Note that once you specify an association for the link, you cannot change the role name or multiplicity for the link (the corresponding fields of the Features window are unavailable).

To change the features of an association of which the link is an instantiation, you must open the Features window of the association itself. Any changes you make to the association are instantly applied to links that are instantiated from it.

- ♦ **End1** and **End2** specifies the two ends of the link, including:
 - **Name** means the name of the link.
 - **Multiplicity** means the multiplicity of the link.
 - **Via Port** means the port used by the link, if any. This is a read-only field.
- ♦ **Description** describes the element. This field can include a hyperlink. For more information, see [Hyperlinks](#).

Link menu

In addition to the common operations (see [Edit elements](#)), the menu for links includes the following options:

When you right-click a link, the menu contains the following options:

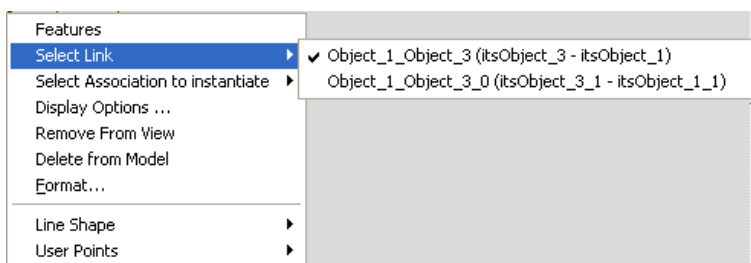
- ♦ **Select Link** lists the available links in the model. For more information, see [Referencing links](#).
- ♦ **Select Association to instantiate** lists the associations available in the model so you can easily select one for the link to instantiate. For more information, see [Mapping a link to an association](#).
- ♦ **Show Labels** specifies whether to display element labels in the diagram.

Note that if you select this option and the link instantiates an association, the link ends will use the labels instead of the role names of the corresponding association ends.

- ♦ **Display Options** determines whether the names and multiplicities of link ends are displayed. For more information, see [Displaying links](#).

Referencing links

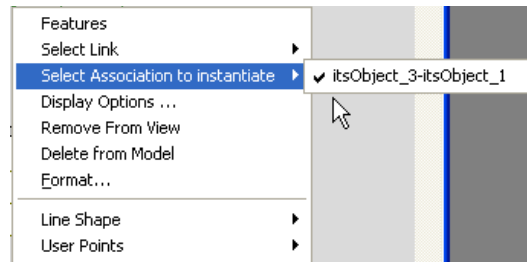
To map a link line to an existing link in the model, right-click the link and select **Select Link**, as shown in this example. This functionality is very useful when you have more than one link between the same elements.



To reference an existing link, select it from the submenu.

Mapping a link to an association

To map a link line to an existing association in the model, right-click the link and select **Select Association to instantiate**, as shown in this example.



To change the association, simply select a different association from the submenu. If you do this and the Active Code View is active, the corresponding code updates to reflect the change.

Displaying links

By default, the names and multiplicities of link ends are not displayed. To change the display, right-click the link and select **Display Options**.

Note

Although the **Link Name** field is available by default, the “generated” link name is not shown on the diagram. However, if you change the name of the link, the new name will be displayed in the diagram.

Enable (check) the fields you want displayed in the diagram.

Using the complete relations functionality

The following table shows the results of using the Complete Relations functionality with associations, generalizations, dependencies, and links.

Completing relations between these elements...	Results in...
Two classes	Rational Rhapsody draws the associations, generalizations, and dependencies but not the links.
Two objects with implicit type	Rational Rhapsody draws the associations, generalizations, dependencies, and links. If the link instantiates an association, the link is drawn, but the association is not.
Two objects with explicit type	Rational Rhapsody draws only the links.

For more information in the Complete Relations functionality, see [Complete relations](#).

Code generation for links

The code for run-time connection of objects is based on links. The connection code is generated when the following conditions are met:

- ◆ Links are specified with an association.
- ◆ The objects connected by the link has an owner (composite class or object) or are global (both objects are owned by a package).

If the objects are parts of a composite, the link is owned by the composite. When the objects are global, the link is owned by a package. Links across packages are initialized by the component.

- ◆ The package and objects for the link are in the scope of the generated component.
- ◆ The `CG::Component::InitializationScheme` property for the component is set to `ByComponent` for links across packages.
- ◆ If more than one link exists between two objects over the same relation, Rational Rhapsody arbitrarily chooses which link to instantiate. The packages that contain the objects are given priority in this decision.

Populating one-to-many associations with objects

If you draw a one-to-many directed association between a class A and a class B, and create an object of A and several objects of B, you can connect the objects with a link that instantiates the association. The Rational Rhapsody-generated code for such a relationship creates a container class for the one-to-many relationship in A, and creates objects of A and B. However, it does not necessarily populate the container for A with the objects of B. When you model a relationship as one-to-n, Rational Rhapsody instantiates n objects in the container. Rational Rhapsody populates only associations with known multiplicity, and graphically shows when an association instance, or link, actually exists and when it does not.

You can populate a one-to-many container by creating the objects in source code and adding them to the container. However, you cannot model a generic one-to-many relationship and populate it with an unknown number of diagrammatically modeled objects. Therefore, it is not possible to populate a one-to-many relationship between classes drawn in one OMD with objects drawn in another OMD.

Restrictions

Note the following limitations and restrictions:

- ♦ Code generation for links across composite classes is not supported.
- ♦ You cannot make a link when one or both sides of the relation are classes (as opposed to objects).

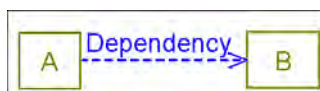
Dependencies

A *dependency* exists when the implementation or functioning of one element (class or package) requires the presence of another element. For example, if class C has an attribute a that is of class D, there is a dependency from C to D.

In the UML, a dependency is a directed relationship from a client (or clients) to a supplier stating that the client is dependent on, and affected by, the supplier. In other words, the client element requires the presence and knowledge of the supplier element. The supplier element is independent of, and unaffected by, the client element.

Dependency arrows

A dependency arrow is a dotted line with an arrow. You can draw a dependency arrow between elements, or you can have one end attached and the other free. It can have a label, which you can move freely. If a dependency arrow is drawn to or from an element, it is attached to the element; the attached end moves with the attached border of the element.



Dependency arrows show that one thing depends on something else:

- ◆ An object that is a (logical) instantiation of another
- ◆ An object that creates or deletes another
- ◆ Constraints attached to an element
- ◆ A class that uses another class or package
- ◆ A package that uses another package or class

You can create a dependency in a diagram or in the browser.

Drawing the dependency

To draw a dependency in the diagram:

1. Click the **Dependency** icon .
2. Click the dependent object.
3. Click the object on which it depends. This object also known as the *provider*.

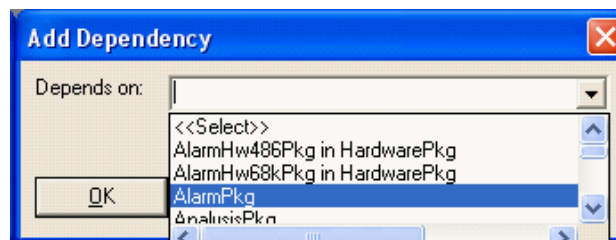
Note that you can create more than one dependency between the same two elements. For example, if you create one dependency from element X to element Y, the default name of the dependency is Y. If you create a second dependency between the same two elements, the second dependency is named Y_0 by default. To rename a dependency, do one of the following actions:

- ◆ Open the Features window for the dependency, and type the new name in the **Name** field.
- ◆ In the browser, left-click the dependency whose name you want to change, and type the new name in the text box.

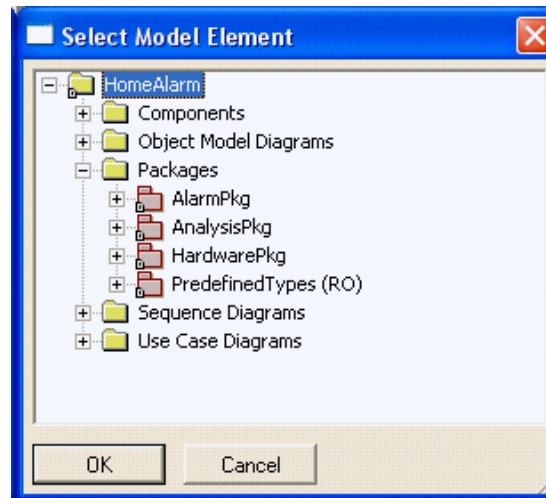
Creating the dependency in the browser

To create a dependency using the browser:

1. In the browser, right-click the element that depends on another element and then select **Add New > Dependency**. The Add Dependency window opens.
2. Use the list to select the element on which the specified element depends.



Click the <<Select>> line to open a browsable tree of the entire project, as shown in the following example.



3. Highlight the appropriate element, then click **OK**.
4. Rational Rhapsody creates the new dependency under the Dependency category for the dependent element, with an open text box so you can easily rename the dependency.



5. If wanted, rename the dependency.

Dependency features

The Features window enables you to change the features of a dependency, including its name and stereotype.

A dependency has the following features:

- ♦ **Name** specifies the name of the dependency.
- ♦ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ **Stereotype** specifies the stereotype of the dependency, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).
Note: The COM stereotypes are constructive; that is, they affect code generation.
- ♦ **Depends On** specifies the class or package that provides information to the dependency.
- ♦ **Description** describes the dependency. This field can include a hyperlink. For more information, see [Hyperlinks](#).

Dependency menu

In addition to the common operations (see [Edit elements](#)), the menu for dependencies includes the following options:

- ♦ **Display Options** specifies how dependencies are displayed. The following figure shows the display options for dependencies.
- ♦ **Select Dependency** enables you to select a dependency. This functionality is useful when you have more than one dependency between the same elements.

Constructive dependencies

Rational Rhapsody supports the dependency stereotypes «Send», «Usage», and «Friend».

Note

If a class has a dependency on another class that is outside the scope of the component, Rational Rhapsody does not automatically generate an `#include` statement for the external class. You must set the «Usage» stereotype and the `<lang>_CG::Class::SpecInclude` property for the dependent class.

Stereotypes are shown between guillemets («. .») and are attached to the dependency line in the OMD, as shown in this example.



The **Properties** tab in the Features window enables you to define the `UsageType` property for the dependency. This property determines how code is generated for dependencies to which a «Usage» stereotype is attached. The possible values for the `UsageType` property are as follows:

- ♦ Specification where an `#include` of the provider is generated in the specification file for the dependent.
- ♦ Implementation where an `#include` of the provider is generated in the implementation file for the dependent.
- ♦ Existence where a forward declaration of the provider is generated in the specification file for the dependent.

For more information on stereotypes, see [Stereotypes](#).

Actors

An actor is a “coherent set of roles that users of use cases play when interacting with these use cases. An actor has one role for each use case with which it communicates.” (UML specification, version 1.3) An actor is a type of class with limited behavior. As such, it can be shown in an OMD.

Creating an actor

To create an actor:

1. Click the **Actor** icon .
2. Click, or click-and-drag, in the diagram.

For a detailed explanation of actors, see [Actors](#). Note that an actor has a Features window that is very similar to that of a class; for more information about the Features window [Creating classes](#).

The actor menu

In addition to the common operations (see [Edit elements](#)), the menu for actors includes the following options:

- ♦ **New Statechart** opens the statechart editor (see [Statecharts](#))
- ♦ **New Activity Diagram** opens the activity diagram editor (see [Activity diagrams](#))
- ♦ **New Attribute** opens the Attribute window (see [Defining the attributes of a class](#))
- ♦ **New Operation** opens the Operation window (see [Class operations](#))
- ♦ **Generate** generates code for the actor (see [Basic code generation concepts](#))
- ♦ **Edit Code** opens the actor code in a text editor (see [Editing Code](#))
- ♦ **Roundtrip** where roundtrips edits made to generated code back into the model (see [The roundtripping process](#))
- ♦ **Open Main Diagram** opens the main diagram for the actor
- ♦ **Display Options** opens the Display Options window
- ♦ **Cut** removes the actor from the view and saves it to the clipboard
- ♦ **Copy** saves a copy of the actor to the clipboard, while leaving it in the view

Flows and flowitems

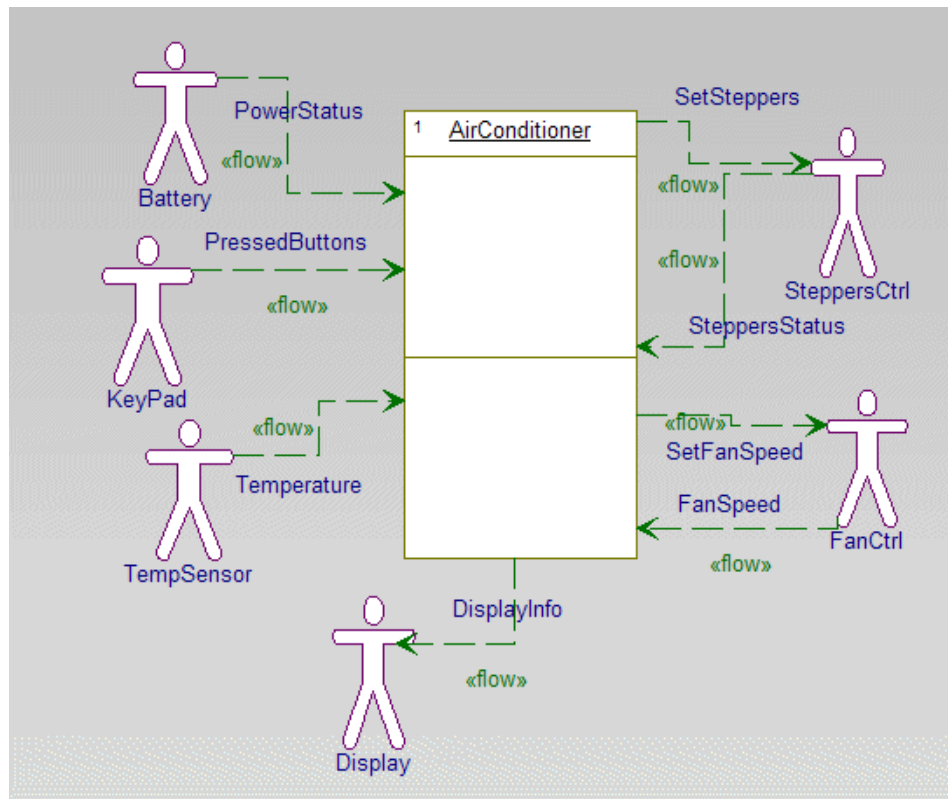
Flows and flowitems provide a mechanism for specifying exchange of information between system elements at a high level of abstraction. This functionality enables you to describe the flow of data and commands within a system at a very early stage, before committing to a specific design. As the system specification evolves, you can refine the abstraction to relate to the concrete implementation.

Flows can convey flowitems, classes, types, events, attributes and variables, parts and objects, and relations. You can draw flows between the following elements:

- ◆ Actors
- ◆ Classes
- ◆ Components
- ◆ Nodes
- ◆ Objects
- ◆ Packages
- ◆ Parts
- ◆ Ports
- ◆ Use cases

You can add flows to all of the static diagrams supported by Rational Rhapsody.

The flows in this object model diagram show the black-box representation of an air conditioning unit and the actors that interact with it. It includes the information that is passed either from an actor to the AC unit or from the AC unit to an actor.



Creating a flow

Every static diagram toolbar includes an **Flow** tool, which is drawn like a link. *Static* (or *structural*) diagrams include object model diagrams, structure diagrams, use case diagrams, component diagrams, and deployment diagrams.

To create a flow:

1. In the **Diagram Tools**, click the **Flow** icon  or choose **Edit > Add New > Relations > Flow**.

Note: **Add New > Relations** is the default menu command structure in Rational Rhapsody. It can be changed by users. This topic assumes that all defaults are in place.

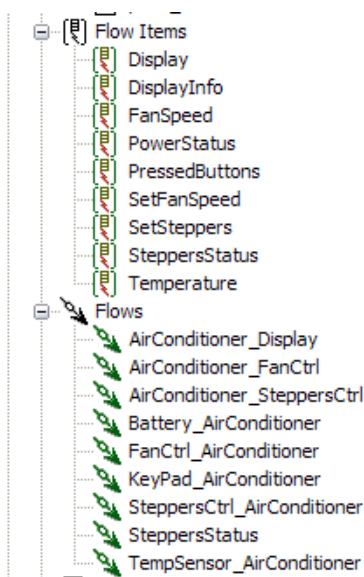
2. In the diagram, click near the first object to anchor the flow.

3. Click to place the end of the flow.
4. In the edit box, type the element conveyed by the flow, then press **Enter**.

By default, a flow is displayed as a green, dashed arrow with the keyword «flow» underneath it.

To suppress the «flow» keyword, open the Display Options window and disable the <<flow>> **keyword** check box. You can also control the display of the <<flow>> keyword for new flows by setting the <Static diagram>::Flow::flowKeyword Boolean property.

The flows and flowitems are both displayed in the browser, as shown in the following example.



Features of a flow

The Features window enables you to change the features a flow, such as its name or flow ends. A flow has the following features:

- ♦ **Name** specifies the name of the flow. By default, the flow is named using the following convention:

`<source>_<target>[_###]`

In this convention, the *source* and *target* are end1 and end2 of the flow, based on the direction (end1 is the source in bidirectional flows as well as flows from end1 to end2); *_###* specifies additional numbering when the name is already used.

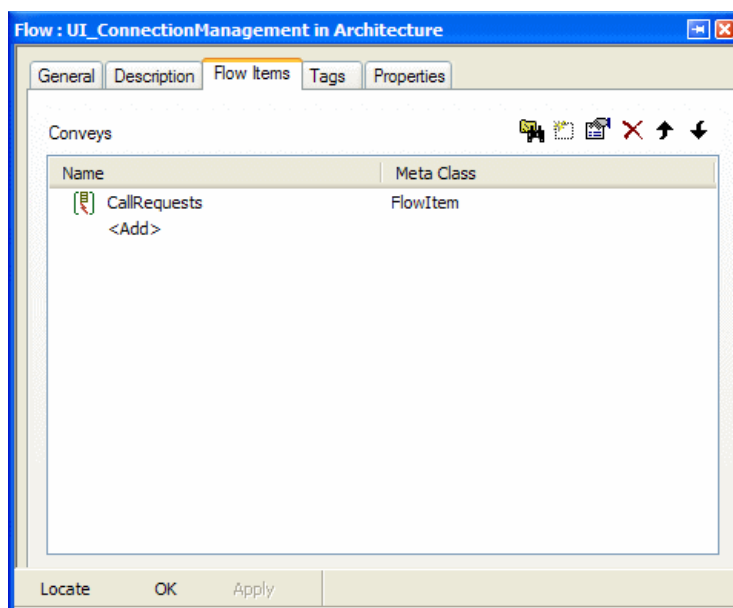
- ♦ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ **Stereotype** specifies the stereotype of the flow, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).

Note: The COM stereotypes are constructive; that is, they affect code generation.

- ♦ **Flow Ends** specifies how the information travels, from the source (**End1**) to the target (**End2**). To change either end, use the pull-down list to select a new source or target from the selection tree.
- ♦ **Direction** specifies the direction in which the information flows. To invert the flow, or to make it bidirectional, use the appropriate value from the pull-down list.
- ♦ **Description** describes the flow. This field can include a hyperlink. For more information, see [Hyperlinks](#).

Conveyed information

All the information elements that are conveyed on the flow are listed on the **Flow Items** tab, as shown in this example.



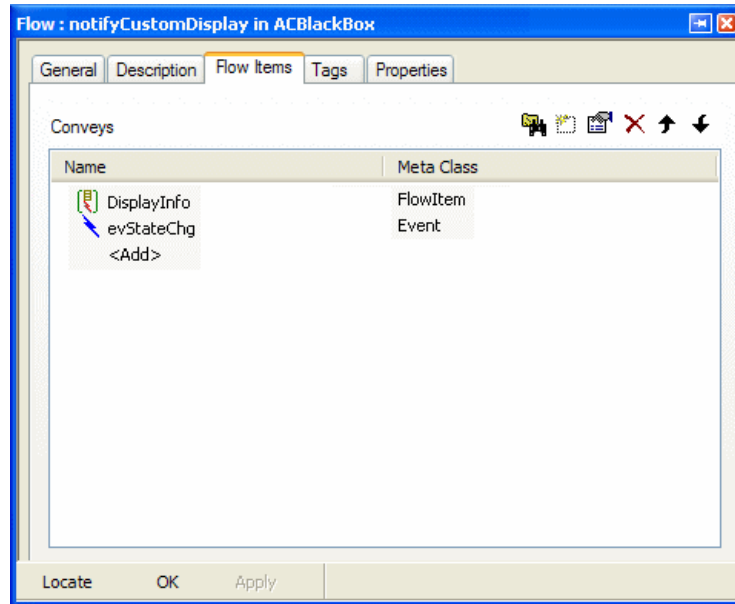
An information element can be any Flowitem, as well as elements that can realize a flowitem (classes, events, types, attributes and variables, parts and objects, and relations).

This tab enables you to perform the following tasks on information elements:

- ◆ Add a new information element.
- ◆ Add an existing information element.
- ◆ Remove an information element.
- ◆ View the features of an information element.

For detailed information on manipulating information elements, see [Flowitems](#).

Note that you can specify multiple information elements using a comma-separated list. For example, in the OMD the flow from the AC unit to the Display actor contains two information elements: the DisplayInfo flowitem and the evStateChg event. The following figure shows the corresponding **Flow Items** tab.



Flow menu

In addition to the common operations (see [Edit elements](#)), the menu for flows includes the following options:

- ◆ **Display Options** opens the Display options window for the flow.
- ◆ **Select Information Flow** provides a list of the flows already defined between these ends so you can easily reuse flows in your model.

Flowitems

A *flowitem* is an abstraction of all kinds of information that can be exchanged between objects. It does not specify the structure, type, or nature of the represented data. You provide details about the information being passed by defining the classifiers that are represented by the flowitem.

Flowitems can be decomposed into more specific flowitems. This enables you to start in a very high level of abstraction and gradually refine it by representing the flowitem with more specific items.

A flowitem can represent either pure data, data instantiation, or commands (events). Flowitems can represent the following elements:

- ♦ Classes
- ♦ Types
- ♦ Events
- ♦ Other information items
- ♦ Attributes and variables
- ♦ Parts and objects
- ♦ Relations

Flowitem features

To view the features of a particular information element, double-click the element in the list on the **Flow Items** tab for the flow. The corresponding Features window opens.

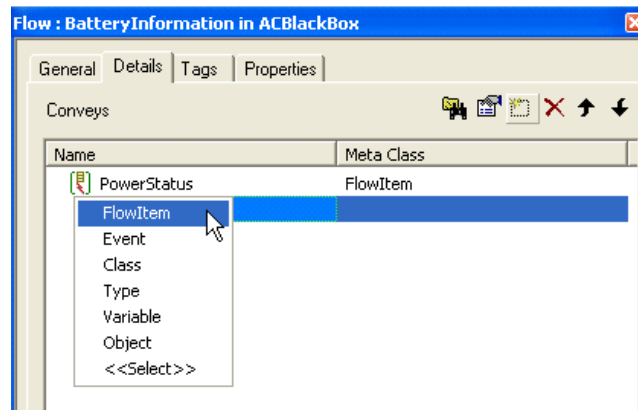
A flowitem has the following features:

- ♦ **Name** specifies the name of the flowitem.
- ♦ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ **Stereotype** specifies the stereotype of the flowitem, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).
Note: The COM stereotypes are constructive; that is, they affect code generation.
- ♦ **Represented** lists all the information elements that are represented by this flowitem. In this example, the DisplayInfo flowitem represents the ACInfo class.
- ♦ **Description** describes the information element. This field can include a hyperlink. For more information, see [Hyperlinks](#).

Adding a new information element

To add a new information element to the flow:

1. To create a new attribute, either click the **<Add>** row in the list of information elements, or click the **New** icon in the upper, right corner of the window and select the appropriate element from the list.



The new element is added to the list and its Features window automatically opens.

2. Set the new values for the element.
3. Click **OK** twice.

Adding an existing information element to the flow

To add an existing information element to the flow:

1. On the **Flow Items** tab for the flow, highlight the **<Add>** row and select the **<Select>** option in the menu.
2. Expand each subcategory as needed to select the information element from the tree, then click **OK**.
3. You return to the **Flow Items** tab, where the specified information element now displays in the list of elements.
4. Click **OK**.

Embedded flows

In SysML notation, flows can be embedded in links. Rational Rhapsody allows you to use this notation in object model diagrams.

Creating an embedded flow

To add a flow to a link:

1. Click the **Flow** icon .
2. Click the link to which you want the flow added.

Once the flow is created, it has the same features as an ordinary flow element, representing the flow of data between the two objects that are linked. Visually, the flow is displayed on top of the link, and it is depicted by an arrow.

To select the embedded flow element, double-click the arrow.

To move the embedded flow diagram element, drag the arrow to a new position on the link.

Changing the flow direction

To change the flow direction, right-click the flow, and then select **Flip Flow Direction**.

Changing display options for embedded flows

For an ordinary flow diagram element, the <flow> keyword is displayed alongside the flow element. To display this for embedded flows:

1. Select the flow.
2. Right-click and select **Display Options**.
3. Select **<flow> keyword or Stereotype**.
4. Click **OK**.

Restrictions

Note the following restrictions and limitations:

- ◆ Flows cannot be animated.
- ◆ There is no code generation for flows.

Files

Rational Rhapsody Developer for C allows you to create model elements that represent files. A *file* is a graphical representation of a specification (.h) or implementation (.c) source file. This new model element enables you to use functional modeling and take advantage of the capabilities of Rational Rhapsody (modeling, execution, code generation, and reverse engineering), without radically changing the existing files.

Note

Files are not the same as the file functionality in components that existed in previous versions of Rational Rhapsody. To differentiate between the two, the new file is called `File` in `Package` and the old file element is called `File` in `Component`. A `File` in `Component` includes only references to primary model elements (package, class, and object) and shows their mapping to physical files during code generation.

A file element can include variables, functions, dependencies, types, parts, aggregate classes, and other model elements. However, nested files **are not** allowed.

Rational Rhapsody supports the following modeling behavior for files:

- ◆ You can drag files onto object model diagrams and structure diagrams.
- ◆ If you use the FunctionalC profile, then the **File** tool is available in the **Diagram Tools** for object model diagrams and structure diagrams.
- ◆ You can drag files onto a sequence diagram, or realize instance lines as files.
- ◆ A file can have a statechart or activity diagram.
- ◆ Files are implicit and always have a multiplicity of 1.
- ◆ Files are listed in the component scope and the initialization tree of a configuration. They have influence in the initialization tree only in the case of a **Derived** scope.
- ◆ Files can be defined as separate units, and can have configuration management performed on them. For more information, see [Using project units](#).
- ◆ Files can be owned by packages only.

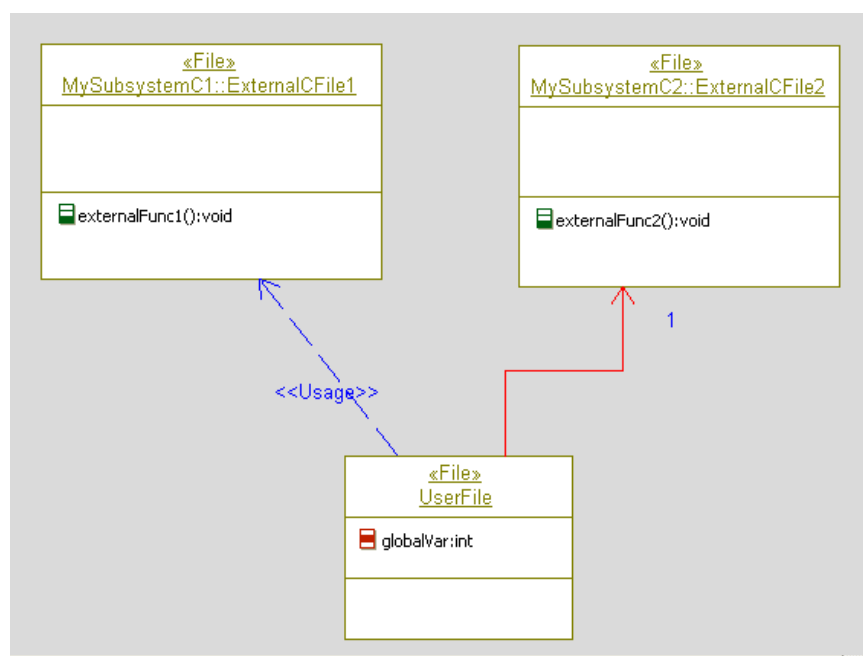
Creating a file

To create a file element:

1. Click the **File** icon in the **Diagram Tools**, or select **Edit > Add New > File**.
2. Click, or click-and-drag, in the drawing area. By default, files are named `file_n`, where *n* is an integer greater than or equal to 0.
3. Edit the default name, then press **Enter**.

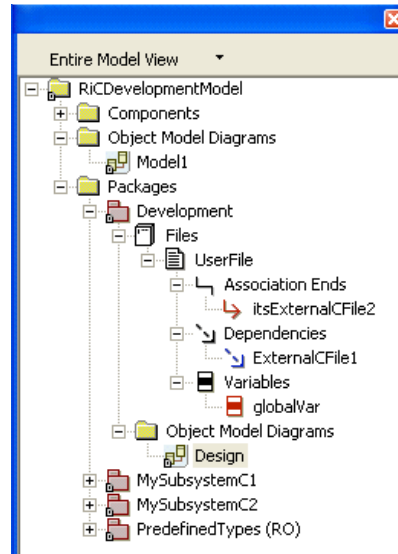
The file is shown as a box-like element in the diagram, with the «File» notation in the top of the box.

The following figure shows an OMD that contains files.



You can specify whether file variables and functions are displayed in diagrams using the **Display Options** feature. The Display Options window for files is identical to that for classes, except the tab names are Variables instead of Attributes and Functions instead of Operations. For more information, see [Display option settings](#).

Files can be owned by packages only. File elements are listed in the browser under the `Files` category under the owning package, as shown in this example.



File features

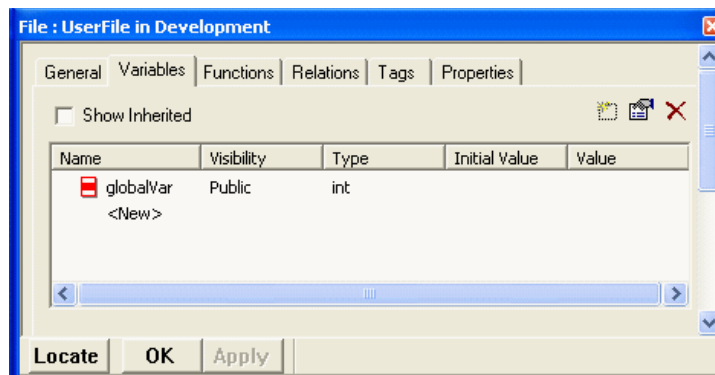
The Features window enables you to change the features of an file, including its name, stereotype, and main diagram. The following figure shows the Features window for a file.

The **General** tab for a file is very similar to that of an object (see [Object features](#)), with the following differences:

- ♦ The **Type**, **Initialization**, **Multiplicity**, and **Relation to whole** fields are unavailable. Multiplicity has no meaning with files, because a file is simply a file (not an object that can be instantiated). Similarly, a file cannot be an instantiation of a class (it is always implicit).
- ♦ The **Path** field is a read-only field that displays the path to the file. Click the icon to navigate directly to the specified source file.

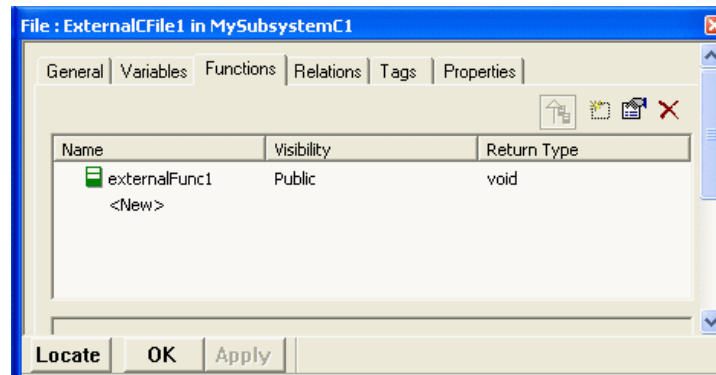
The Variables tab

The **Variables** tab of the file Features window enables you to add, edit, or remove variables from the file. It contains a list of all the variables belonging to the file. The following figure shows the **Variables** tab.



The Functions tab

The **Functions** tab of the file Features window enables you to add, edit, or remove functions from the file. It contains a list of all the functions defined in the file. The following figure shows the **Functions** tab.



Converting files

You can easily convert a file to an object or vice versa by simply highlighting the object in the browser, then selecting **Change to** and the intended result.

To convert a file to a class:

1. Highlight the file in the browser, right-click and then select **Change to > Object**. The file changes to an object and moves to the Objects category in the browser.
2. Highlight the object in the browser, right-click and then select **Expose Class**. This creates a new class with the name `<object> class`. It contains all of the content of the copied object including the attributes, operations, and statechart. This option is only available for an implicit object.

Note the following information:

- ◆ If you are trying to convert an object to a file and there are aggregates that are not allowed for files, Rational Rhapsody issues a warning message.
- ◆ Objects that are owned by another class or object cannot be converted to files.

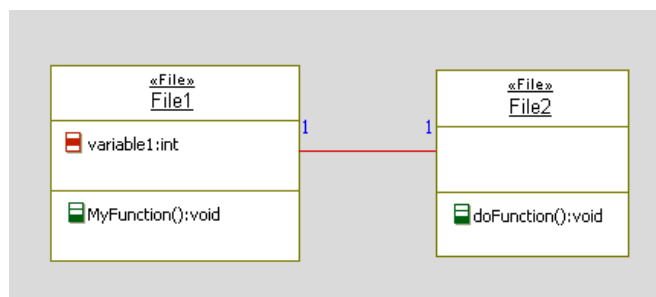
When the element has been converted, the graphical representations change in the diagrams and the converted element is moved to the appropriate category in the browser.

For information on changing the order of files, see [Editing the declaration order of objects](#).

Associations and dependencies

Files can be connected through associations or «Usage» dependencies. As standard practice, you should use associations.

For example, consider the files shown in the following figure.



Because these files are connected through bi-directional association, File1 can call `doFunction()` directly from an operation or action on behavior.

Code generation for files

During code generation, files produce full production code, including behavioral code. In terms of their modeling properties, modeled files are similar to implicit singleton objects.

For an active or a reactive file, Rational Rhapsody generates a public, implicit object (singleton) that uses the active or reactive functionality. The name of the singleton is the name of the file.

Note

The singleton instance is defined in the implementation source file, not in the package source file.

For a variable with a **Constant** modifier, Rhapsody generates a `#define` statement. For example:

```
#define MAX 66
```

The following table shows the differences between code generation of an object and a file.

Model Element	File Code	Object Code
Data member (attribute, association, or object)	A global variable	A member in the singleton struct
Function name ¹	The function name pattern is <code><Function></code> .	The name pattern for public functions is <code><Singleton>_<function></code> . The pattern for private functions is <code><Function></code> .
Function signature	The <code>me</code> argument is generated when required to comply with the signature of framework callback functions (for reactive behavior).	The same.
Initialization	Variables and associations are initialized directly in the definition. For example: <code>int x=5;</code> Objects are initialized in a generated <code>Init</code> function.	Done in the <code>initialize</code> function.
Type name	The name pattern for types (regardless of visibility) is <code><Type></code> .	The name pattern for public types is <code><Singleton>_type</code> . The name pattern for private functions is <code><Type></code> . The name pattern can be configured using the properties <code><lang>_CG::Type::PublicName</code> and <code>PrivateName</code> .

Model Element	File Code	Object Code
Visibility	Public members are declared as extern in the specification (.h) file and defined in the implementation (.c) file. For example: <pre>extern int volume;</pre> Private members are declared and defined in the implementation file as static. For example: <pre>static int volume;</pre>	Member visibility is ignored; the visibility is a result of the visibility of the struct. For example: <pre>struct Ob_t { int volume; };</pre>
<i>Auto-generated</i>		
Initialization and cleanup	Only algorithmic initialization is done in the initialization method (creating parts; initializing links, behavior, and animation). The initialization and cleanup methods are created only when needed. The name of the initialization function is <file>_Init; the cleanup function is <file>_Cleanup.	Any initialization is done in the Init method. Init and Cleanup methods are generated by default.
Framework data members	Rational Rhapsody generates a designated struct that holds only the framework members, and a single instance of the struct named <file>. The struct name is <file>_t. For example: <pre>struct Motor_t { RiCReactive ric_reactive; };</pre>	Framework members are generated as part of the object struct declaration.
Call framework operations	Framework operations on the file are called using the file. For example: <pre>CGEN(Motor, ev());</pre>	Framework operations on the singleton are called passing the singleton instance. For example: <pre>CGEN(Motor, ev());</pre>
Statechart data members	Statechart data members are generated as attributes of the generated structure. For example: <pre>struct F_t { ... enum F_Enum { F_RiCNonState=0, F_ready=1} F_EnumVar; int rootState_subState; int rootState_active; };</pre>	Statechart data members are generated as part of the struct.

Model Element	File Code	Object Code
Statechart function names	Public statechart functions are generated using the prefix <file>_. For example: myFile_sIN()	Use the same naming convention as any other operation.

1. You can configure the name pattern for functions (for files, objects, and other elements) using the properties <lang>_CG::Operation::PublicName and PrivateName.

Files with other tools

The following table lists the effect of files on both Rational Rhapsody and third-party tools.

Tool	Description
COM API	Files are supported by the COM API via the IRPFile and IRPModule interfaces
Complete Relation	When you select Layout > Complete Relations , files and their relations are part of the information added to the diagram. For more information on this functionality, see Complete relations .
DiffMerge	Files are included in difference and merge operations, completely separate from objects.
Java API	Files are supported by the Java API
Populate Diagram	Files and their relations are fully supported
References	If you use the References functionality for a file, the tool lists the owning package for the file and the diagrams in which the specified file displays. When you select a diagram from the returned list, the file is highlighted in the diagram. For more information on this functionality, see Searching in the model .
Report on model	In a report, the objects and files in the package are listed in separate groups in that order. For more information on this reporting tool, see Reports .
Search in model	You can search for files in the model and select their type from the list of possible types. When selected, the file is highlighted in the browser. For more information on this functionality, see Searching in the model .
XMI Toolkit	When you export a file to XMI, it is converted to an object with a «File» stereotype. Files imported from XMI are imported into Rational Rhapsody as files.

Attributes, operations, variables, functions, and types

In object model diagrams, attributes and operations are contained in classes. Therefore, they are not included as separate items in the **Diagram Tools**. Similarly, variables, functions, and types are not included in the **Diagram Tools**.

There might, however, be situations where you will want to show a higher level of detail and include attributes, operations, variables, functions, or types as individual diagram elements. Rational Rhapsody provides a solution for these situations by allowing you to drag these elements from the browser to an OMD diagram

Adding details to the object model diagram

To add an attribute, operation, variable, function, or type to the diagram:

1. Select the relevant item in the browser.
2. Drag the item to the diagram window.

Note

Rational Rhapsody allows these types of elements to be dragged from the browser to any of the static diagrams, not just object model diagrams.

When an item of this type is added to a diagram, the graphic element will display by default, the element name, the stereotype applied (if there is one) or the metatype of the element, and the associated image (if one has been defined).

Like all diagram elements, the Features window for these elements can be opened by double-clicking on the element in the diagram.

The connectors provided in the **Diagram Tools** can be used to connect individual elements of these types if the connection is semantically logical.

Once an element has been added to a diagram, the element can be added to a container element by dragging the element into the container element, for example, an attribute on the diagram can be dragged into a class.

Note

Graphic representations for these types of items can only be created by dragging them from the browser to the diagram. There is no API equivalent for this action.

Flow ports

Flow ports allow you to represent the flow of data between objects in an object model diagram, without defining events and operations. Flow ports can be added to objects and classes in object model diagrams. They allow you to update an attribute in one object automatically when an attribute of the same type in another object changes its value.

Note

Flow ports are not supported in Rational Rhapsody in J.

Adding a flow port

To add a flow port, right-click the object or class and then select **Ports > New Flowport**.

The method used for specifying the data that is to be sent/received via the flow port depends upon the type of flow port used - atomic or non-atomic. Non-atomic flow ports can only be used if your model uses the SysML profile. The following sections describe these two types of flow ports.

Atomic flow ports

Atomic flow ports can be input or output flow ports, but not bidirectional. To specify the flow direction, open the Features window for the flow port and select the appropriate direction.

You specify the attribute that is to be sent/received via the flow port by giving the attribute and flow port the same name. If no attribute name matches the name of the flow port, a warning to this effect will be issued during code generation.

Atomic flow ports allow the flow of only a single primitive attribute.

When connecting two atomic flow ports, you have to make sure that one is an input flow port and one is an output flow port. The type of the attribute in the sending and receiving objects must match.

To connect two flow ports, use the Link connector.

Defining non-atomic flow ports

Non-atomic flow ports are available only in models that use the SysML profile.

Non-atomic flow ports can transfer a list of flow properties (a *flow specification*), which can be made up of flow properties of different types. For each flow property in the list, you indicate the direction of the flow. (Non-atomic flow ports are bi-directional.)

To define the flow properties to be sent/received via the flow port:

1. Create a flow specification.
2. Add flow properties to the flow specification. This can be done using the relevant browser context menu, or directly on the **FlowProperties** tab of the Features window for the flow specification. You can also use drag-and-drop in the browser to add existing flow properties to the flow specification. (If you want to use an existing attribute, you can convert the attribute to a flow property by selecting **Change To > FlowProperty** from the attribute context menu.)
3. For each of the flow properties defined, specify the direction.
4. Create two objects and add a flow port to each.
5. In the Features window for each of the two flow ports (the sending and receiving), set the Type to the name of the flow specification you created previously.
6. For one of the flow ports, open the Features window and select the **Reversed** check box on the **General** tab
7. Connect the two flow ports with a link.

Updating attribute values

To have the value of an attribute updated when the attribute on the other end of flow is updated, you must use the function `set/flowportname`, for example, if you have a flow port called `x`, you would call `setX(5)`.

When this function is called, there is also an event generated called `ch/flowportname`. In our example, it would be `chX`. In order to be able to react to this event, you must define an event with this name in your model.

For both of these functions, the first letter of the flow port name is upper-case even if the actual name of the flow port begins with a lower-case letter.

Note

For details regarding the use of flow ports when importing Simulink models, see [Integrating Simulink components](#).

External elements

Rational Rhapsody enables you to visualize frozen legacy code or edit external code as *external elements*. This external code is code that is developed and maintained outside of Rational Rhapsody. This code will not be regenerated by Rational Rhapsody, but will participate in code generation of Rational Rhapsody models that interact or interface with this external code so, for example, the appropriate `#include` statement is generated. This functionality provides easy modeling with code written outside of Rational Rhapsody, and a better understanding of a proven system.

Rational Rhapsody supports the following functionality for external elements:

- ◆ Reverse engineering can import elements as external.
- ◆ Reverse engineering populates the model with enough information to:
 - Model external elements in the model.
 - Enable you to open the source of the external elements, even if the element is not included in the scope of the active component.
- ◆ Rational Rhapsody generates the correct `#include` for references to external elements.
- ◆ Elements inherit their externality from the parent. For example, if a package is external, all its aggregates are also external.
- ◆ You can add external elements to component files to define the exact location of the source code.
- ◆ Rational Rhapsody displays external elements in the scope tree of the component.

There are two ways to create external elements:

- ◆ By reverse engineering the files
- ◆ By modeling

Reverse engineering

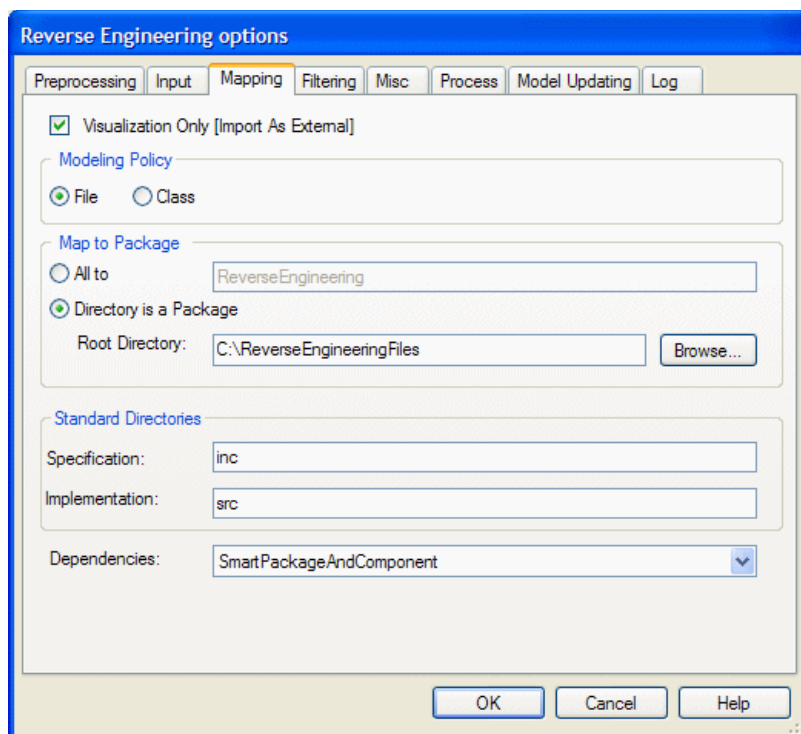
When creating external elements using reverse engineering, the preferred method depends on whether the code is frozen legacy code or the code is still being modified.

Reverse engineering a single iteration

For legacy code or a library that will not change, it is appropriate to model external code for referencing, without regenerating it.

To use reverse engineering to create the external elements a single time:

1. Create a new model or open an existing one.
2. Add a new component for the reverse engineered, external code.
3. Set your new component (created in the previous step) to be the active component (right-click it in the Rational Rhapsody browser and select **Set as Active Component**).
4. Choose **Tools > Reverse Engineering** to open the Reverse Engineering window.
5. Specify the files or folders you want to reverse engineer.
6. Click the **Advanced** button to open the Reverse Engineering Options window.
7. On the **Mapping** tab, specify the following settings:
 - a. Select the **Visualization Only (Import as External)** check box.
The following figure shows an example for Rational Rhapsody in C.



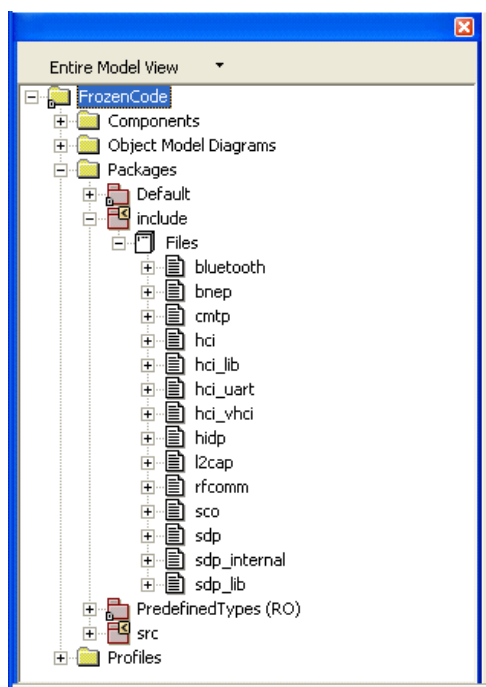
- b. For Rational Rhapsody in C, select the **Files** radio button (default) in the **Modeling Policy** area; for the other languages, select the appropriate option for your situation. The availability of these radio buttons might depend on whether you select the **Visualization Only (Import as External)** check box.
8. Set the other reverse engineering options as appropriate for your model. (For more information on the available options on the **Mapping** tab, see [Mapping classes to types and packages.](#))
9. Click **OK**.
10. Click the **Start** button on the Reverse Engineering window. The specified files are imported into Rational Rhapsody as external elements.

As a result of the import:

- ◆ The imported elements are added to the scope of the configuration.
- ◆ All the imported packages have the property `CG::Package::UseAsExternal` set to Checked.
- ◆ The **Include Path** or **Directory** of the Features window for the configuration (in the example, `ExternalComponent`) is set to the correct include path.

- ◆ In Rational Rhapsody in C, when the **Directory is a Package** radio button is selected, the `C.CG::Package::GeneratedDirectory` property is set to Checked for the configuration.

Note that external elements include a special icon in the upper, right corner of the regular icon, as shown in this example.



11. Verify the import to make sure the implementation and specification files are named correctly, the correct folders were created, and so on. Make any necessary changes.
12. Set the original component to active.
13. For the original component, create a dependency with a «Usage» stereotype to the ExternalComponent.
14. Make sure that the external elements are included in the scope of the ExternalComponent only.

Reverse engineering multiple iterations

Suppose you want to model external code for referencing, without regenerating it (but the external code might change and the external element should be updated according to the changes in the code).

Set up your model as follows:

1. Complete the steps in the previous procedure (see [Reverse engineering](#)) to create a new external model (for example, `ExternalModel`).
2. Save your model, then close it.
3. Open a new, development model.
4. Choose **File > Add to Model**, then select the external model. Select **As Reference** and select all the top-most packages and the component (`ExternalModel`). The elements are imported as read-only (RO).
5. Create a dependency with a «Usage» stereotype to the `ExternalModel`.

To synchronize the code changes:

1. Open the external model.
2. Update the reverse engineering options as needed to include the code modifications (such as including new folders), and then click **Import**.
3. Close the external model.
4. Open the development model.
5. Update the model according to the changes in the external model:
 - a. Remove references to elements deleted from the external model.
 - b. Update references to renamed elements from the external model (they become unresolved).
 - c. New elements are simply added to the model.

Creating external elements in pre-V5.2 models

To add external elements to models created before Version 5.2:

1. Unoverride the following properties:
 - ◆ `CG::Configuration::StrictExternalElementsGeneration`
 - ◆ `CG::Component::SupportExternalElementsInScope`

These properties are automatically overridden by Rational Rhapsody when you load an older model.
2. Follow the procedure described in [Reverse engineering a single iteration](#).

External elements created by modeling

Alternatively, you can add external elements to the model manually. This option is used when there are very few elements to be modeled as external.

There are two ways to model the elements manually:

- ◆ Using rapid external modeling
- ◆ Using the component model

Using rapid external modeling

To model the elements manually using rapid external modeling:

1. Open an existing model or create a new one.
2. Create the external elements:
 - ◆ Create the new element to be referenced.
 - ◆ Set its `CG::Class::UseAsExternal` property to `Checked`.
 - ◆ Set its `CG::Class::FileName` property to the value expected in the `#include`. For example, `MySubsystem\C`.
3. Add the rest of the path to the **Include Path** field of the component. In the example, this would be `C:\MyProjects\Project1`.
4. Add the external elements to the scope of the component.
5. Add relations to the external elements (for more information, see [External element code access](#)).

Using the component model

To model the elements manually using the component model:

1. Open an existing model or create a new one.
2. Add a new component for the external elements (for example, `ExternalComponent`).
3. Set the scope of the component to **Selected Elements**.
4. Create a package that will contain all the external elements, and set its `CG::Package::UseAsExternal` property to `Checked`.

Note: This step is optional; you can also add external elements to existing packages.
5. Add the package that contains the external elements to the scope of the external component. Make sure that the package is not included in the scope of other components.
6. Create a new element that will be referenced in the package.
7. Provide the following information about the source files of the external elements:
 - a. Create a hierarchy of packages as needed for the proper `#include` path. For example, suppose you want reference class `C`, which is defined in `C:\MyProjects\Project1\MySubsystem\C.h`; you would create the package `MySubsystem`.
 - b. Add a file with the necessary name to the folder and map the external element to it. You do not need to do this if the external element has the same name as the file.
 - c. Create a usage dependency to the external component.
8. Add relations to the external elements (for more information, see [External element code access](#)).

In the generated files, the following `#include` is generated for the example element:

```
#include <MySubsystem\C.h>
```

Creating a shortcut for Rational Rhapsody Developer for C

In Rational Rhapsody Developer for C, you can use the following shortcut in place of Steps 2–7 in the previous procedure:

1. Create a hierarchy of packages as needed for the proper `#include` path.

For example, suppose you want reference class `C`, which is defined in `C:\MyProjects\Project1\MySubsystem\C.h`; you would create the package `MySubsystem`.
2. Set the `CG::Package::UseAsExternal` property for the top-most package to `Checked`.
3. Create the appropriate files (for more information, see [Creating a file](#)). Continuing the example, you would simply create the file `C`.
4. Create a new element that will be referenced in the file.
5. Add the rest of the path to the **Include Path** field of the component. In the example, this would be `C:\MyProjects\Project1`.
6. Set the property `C_CG::Package::GenerateDirectory` to `Checked` for the component.

Converting external elements

You can convert external elements so they are no longer external (and therefore include them in code generation). This functionality enables you to gradually move code that was developed outside of Rational Rhapsody to an application being developed using Rational Rhapsody.

To convert all the external elements at once:

1. Open the model.
2. Change the following properties:
 - ◆ Unoverride the property `CG::Package::UseAsExternal` for the top-most packages.
 - ◆ Unoverride the properties `Generate` and `AddToMakefile` (under `CG::File`) for the top-most folders in the external element component.
3. Add the packages and classes to the scope of the development component and remove the packages and classes from the external element component.
4. Delete the external component.
5. Generate and build the code.
6. Continue development in Rational Rhapsody as usual.

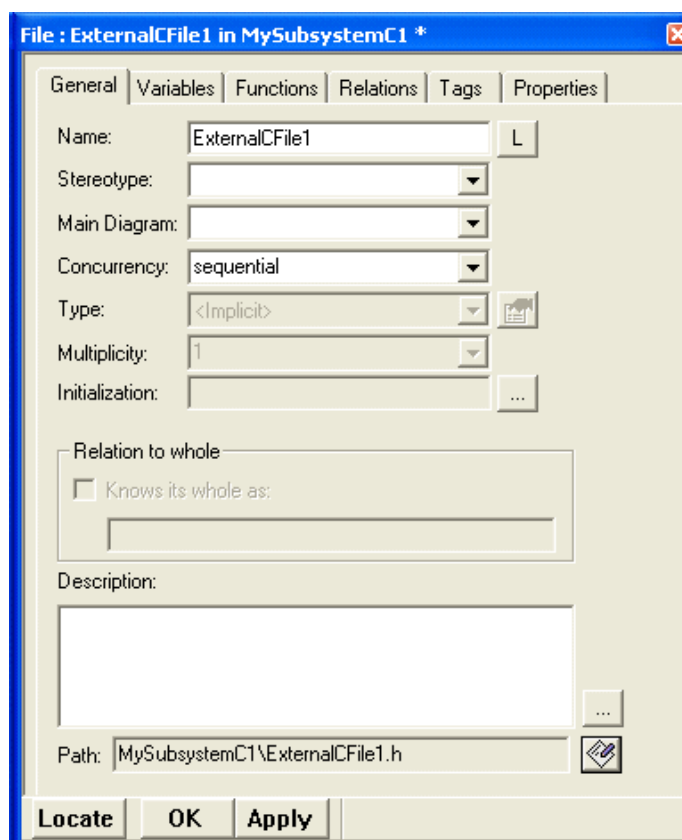
To convert specified external elements:

1. Create a new package for the converted elements (for example, `RedesignPackage`).
2. Add the new package to the scope of the development component.
3. Move one class or file from the external package to the `RedesignPackage`.
4. Generate, build, and test the code. Repeat as necessary.
5. Repeat Steps 3 and 4 for as many classes or files as you want to convert.
6. Continue development in Rational Rhapsody as usual.

Viewing the path to the source file

The Features window for an external file in Rational Rhapsody in C includes a new field, **Path**, which shows the full path to the specification source file. The path is read-only, but you can copy it.

The following figure shows the updated window.



Click the icon to navigate directly to the specified source file.

External element code access

To access the code in the external files, create relations to them (such as dependencies with «Usage» stereotypes, generalizations, associations, and so on). Note that the resultant source code will automatically contain the correct `#include` statements for the external elements.

In the Features window for the configuration, set the scope to **Selected Elements**, and verify that the external files are not checked.

When you generate code that includes relations to external elements:

- ◆ Dependencies are converted to `#includes`.
- ◆ Generalizations are converted to inheritance and `#include`.
- ◆ Associations are converted to data members and `#includes`.
- ◆ Types are converted to type and `#include`.
- ◆ Objects and parts are instantiated.

Once you have created the external elements, you can edit the code using the Edit menu, edit options in the menu, or active code view window, just as you do for any Rational Rhapsody code.

Adding source files to the build

To add the source files of an external element to the build:

1. Add the component file that contains the mapping of the necessary external elements to the component.
2. Set the property `CG::File::Generate` to `Cleared`.
3. Set the property `CG::File::AddToMakefile` to `Checked`.

Code generation for external elements

The following table lists how Rational Rhapsody generates code for external elements.

Element Type	Description
Package	The code generator does not generate code for an external package. However, you can map the package to a file or folder for a component (and then relate it to a file or directory). You can include the package in the component scope. During code generation, a relation to a package is converted an <code>#include</code> to a file, if the package is mapped to a file for the component.
Class, object, or file	The code generator does not generate code for an external class, object, or file. During code generation, a relation to a class, object, or file is converted to an <code>#include</code> or a forward declaration.
Type	The code generator does not generate code for an external type. A relation to a type is converted to an <code>#include</code> of its parent.
File (component)	A file is external if all its elements are external. If the <code>CG::File::Generate</code> property for a file is set to <code>Cleared</code>, the file becomes external and code is not generated for it. To include a file in the build, set its <code>CG::File::AddToMakefile</code> property to <code>Checked</code>.

Code generation for relations

During code generation, Rational Rhapsody generates either an `#include` or a forward declaration for a relation in the source file of the dependent element.

Forward declaration (class)

If a dependency has a «Usage» stereotype and the `CG::Dependency::UsageType` property is set to `Existence`, it is generated as a forward declaration. For example:

```
class ExternalClass;
```

#includes for a class, object, or file

External dependencies (dependencies with a «Usage» stereotype and the `CG::Dependency::UsageType` property set to `Specification/Implementation`) and implicit dependencies (such as associations and generalizations) are generated as forward declarations and `#include` statements.

To generate a local `#include` statement (for example, `#include <C.h>`), set the property `CG::File::IncludeScheme` to `LocalOnly`.

To generate a relative `#include` statement (for example, `#include <MySubsystem\C.h>`), set the `CG::File::IncludeScheme` to `RelativeToConfiguration`.

You can also use the `C_CG/ CPP_CG/ JAVA_CG::Package::GeneratedDirectory` and `CG::Class/Package::FileName` properties to set relative paths. See the definition of this property in the Features window.

Limitations

Note the following restrictions and behavior of external elements:

- ♦ External elements are not animated; they behave like the system border.
- ♦ Changes in the source files of external elements are not roundtripped. If necessary, use reverse engineering to update external elements.
- ♦ Only the following elements can be external: class, implicit object or file, package, and type. Components, folders, variables, and functions cannot be external.
- ♦ You cannot use **Add to Model** to add an external element as a unit. However, you can add the unit under another name, and then set that unit to be external.

Implementation of the base classes

In Rational Rhapsody Developer for C++ and for Java, you can easily convey model elements defined at the interface level to the implementing class level. Using this functionality, classes automatically realize the implementing interfaces and help you synchronize the changes in the interface to the realizing classes.

There are two ways to start this functionality: implicitly and explicitly.

Implicit invocation

When you connect two classes with a generalization realization, the base classes are implemented implicitly. However, a generalization will not trigger base class implementation in the following cases:

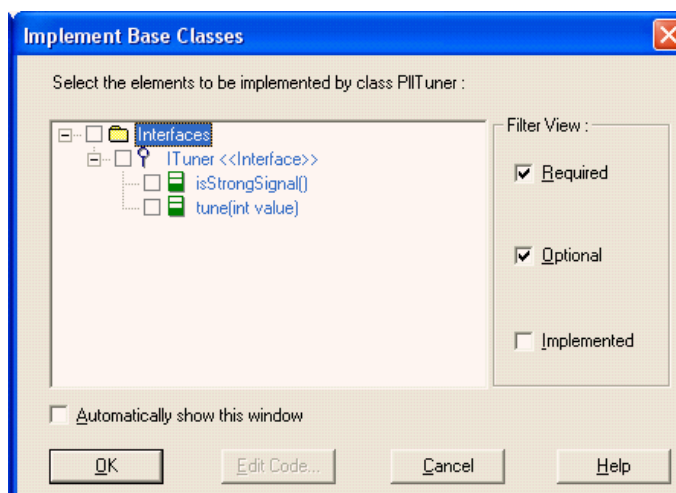
- ◆ Inheritance between two COM interfaces
- ◆ Inheritance between two CORBA interfaces
- ◆ Inheritance between two Java interfaces

If there are no operations or attributes to be overridden because of the current action, the Implement Base Classes window is not displayed.

Explicit invocation

To access this functionality explicitly, in the browser or OMD, right-click a class and select **Implement Base Classes**.

The Implement Base Classes window opens, as shown in this example.



Implement base classes window

This window provides a tree-like view of all the interfaces (including methods, attributes, and stereotypes) that can be implemented by the class.

The window contains three filters to control the contents of the tree view:

- ♦ **Required** displays the operations that must be implemented.
- ♦ **Optional** displays the operations that can be implemented. By default, Rational Rhapsody displays the required and optional operations.
- ♦ **Implemented** displays the operations that are already implemented.

Base class tree view

Depending on the base class, Rational Rhapsody displays different items in the tree view. The following table shows which items are displayed.

Base Class	Items Displayed in the Tree View
C++ class	All virtuals and pure virtuals. You must implement the pure virtual methods.
Java class or Java interface	All the methods. You must implement the interfaces. The GUI takes into account the “final” option for Java methods and classes.
COM interface	All methods and attributes.
CORBA interface	All methods.

Rational Rhapsody uses the following colors to differentiate the different method types:

- ♦ **Blue** denotes a virtual method.
- ♦ **Bold, blue** denotes an abstract method.
- ♦ **Gray** denotes a method that has already been implemented.

If you try to open the Implement Base Classes functionality for a read-only class, Rational Rhapsody displays a warning message informing you that the class cannot be modified. However, the Implement Base Classes window opens in read-only mode so you can analyze the class. You can view the code by selecting **Edit Code**, but the **OK** button will be disabled.

Editing the implementation code

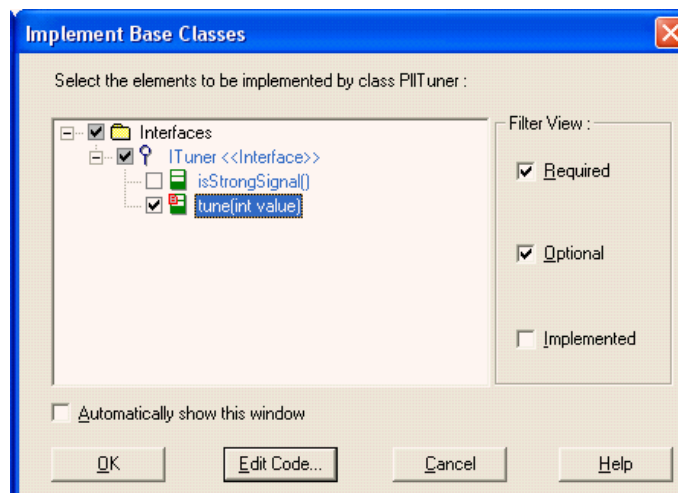
To change the implementation code for an operation:

1. Select the operation in the tree view.
2. Click the **Edit Code** button. Rational Rhapsody displays the code in the default text editor.
3. Type in the new implementation code in the text box.
4. Click **OK**.

Note that if you try to edit code for an operation that has already been implemented, the text editor displays the implementation code in read-only mode.

When you edit the implementation code, Rational Rhapsody overlays a red icon in the upper, left corner of the class icon in the tree view, as shown in this example.

For more information on the internal code editor and its properties, see [The Internal code editor](#).



Controlling the display of the window

The **Automatically show this window** check box controls whether the window is displayed on implicit requests. By default, this check box is unavailable, so the window is displayed only when you explicitly open it.

If you select this check box, Rational Rhapsody writes the following line to the [General] section of the `rhapsody.ini` file:

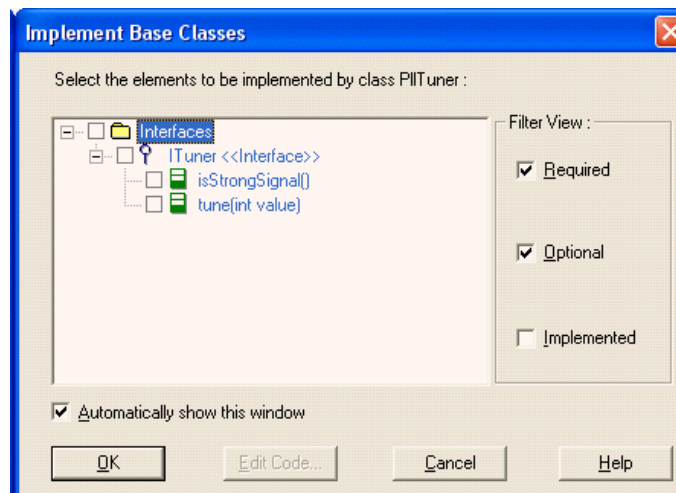
```
ImplementBaseClasses=TRUE
```

If wanted, you can add this line directly to the `rhapsody.ini` file to automatically display the window.

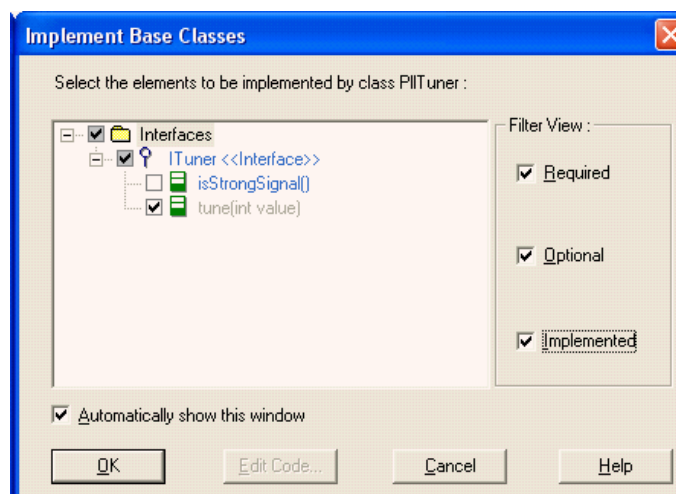
Realizing the elements

To realize an element, select it and click **OK**.

For example, suppose you want to implement the tune operation. In the window, select tune, then click **OK**.



The `PIITuner` class implements the tune operation and displays it in the browser. If you select **Implement Base Classes** for the `PIITuner` class again, the tune operation is no longer listed as a required or optional element. Click **Implemented** to see that the tune operation was implemented and is now displayed in gray, as shown in this example.



If an element has been implemented (it displays in gray in the tree and is checked), you cannot uncheck (“unrealize”) it.

Note

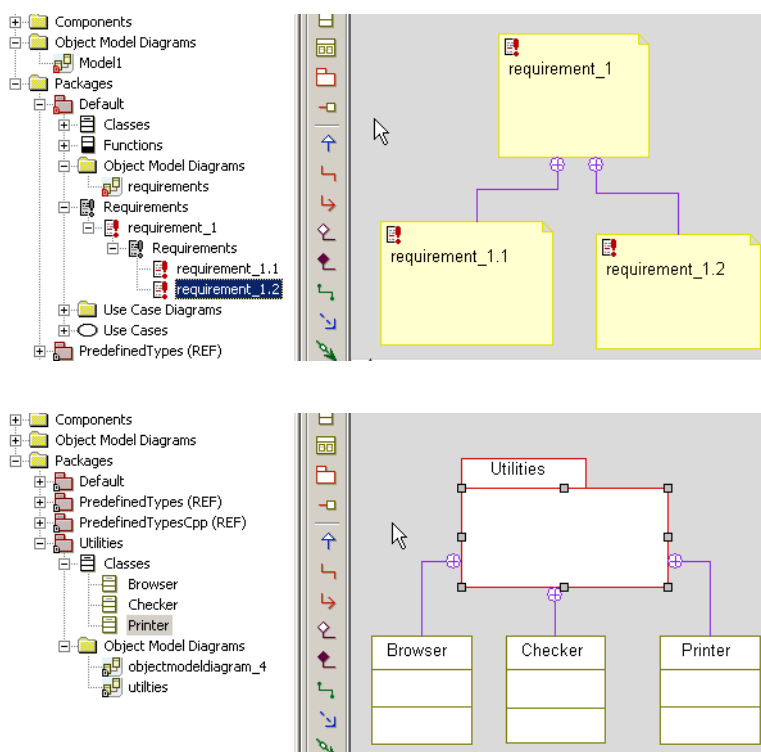
If you choose **Undo**, Rational Rhapsody unimplements all the implemented classes, not just the last one, because the implementation is viewed as a single, atomic operation. For example, if you implement five elements during one operation, then select **Undo**, all five are removed.

Namespace containment

Rational Rhapsody allows you to display namespace containment in object model diagrams. This type of notation is also referred to as “alternative membership notation.” It depicts the hierarchical relationship between elements and the element that contains them, for example:

- ◆ requirements that contain other requirements
- ◆ packages that contain classes
- ◆ classes that contain other classes

The following screen captures illustrate the use of this notation.



Property that controls display of namespace containment

The display of namespace containment is controlled by the boolean property `ObjectModelGe:ClassDiagram:TreeContainmentStyle`, which can be set at the diagram, package, or project level. Namespace containment is displayed when the property is set to `Checked`.

The default value of the property is `Cleared`. However, in the SysML profile the default value of the property is `Checked`.

Displaying namespace containment

To display namespace containment in a diagram:

1. Drag the “container” element and the “contained” elements to the diagram.
2. From the menu, select **Layout > Complete Relations > All**

The hierarchical relationship between the elements will be depicted in the diagram.

Alternatively, you can select the **Populate Diagram** check box when creating a new diagram. If you then select elements that have a hierarchical relationship, the diagram created will display the namespace containment for the elements.

Note

There is no drawing tool to manually draw this type of relationship on the canvas. Containment relationships between elements can only be displayed automatically based on existing relationships, using one of the methods described above.

Activity diagrams

Activity diagrams specify a workflow, or process, for classes, use cases, and operations. As opposed to statecharts, activity diagrams are preferable when behavior is not event driven. A class (use case/operation) can have either an activity diagram or a statechart, but not both. However, a class, object, or use case might have more than one activity diagram with one of the diagrams designated as the *main behavior*.

Note

It is possible to change the main behavior between different activities within the same classifier.

Activity diagram features

One useful application of activity diagrams is in the definition of algorithms. *Algorithms* are essentially decompositions of functions into smaller functions that specify the activities encompassed within a given process.

Note

Sequence diagrams can show algorithms of execution within objects, but activity diagrams are more useful for this purpose because they are better at showing concurrency.

Activity diagrams have several elements in common with statecharts, including start and end activities, forks, joins, guards, and states (called *actions*). Unlike statecharts, activity diagrams have the following elements:

- ♦ **Decision points** show branching points in the program flow based on guard conditions.
- ♦ **Actions** represent function invocations with a single exit activity flow taken when the function completes. It is not necessary for all actions to be within the same object.
- ♦ **Action blocks** represent compound actions that can be decomposed into actions.
- ♦ **Subactivities** represent nested activity diagrams.
- ♦ **Object nodes** represents an object passed from the output of the action for one state to the input of the actions for another state.
- ♦ **Swimlanes** visually organize responsibility for actions and subactions. They often correspond to organizational units in a business model.

- ◆ **Reference activities** references an activity in another activity chart, or to the entire activity chart itself.

Advanced features of activity diagrams

You might also use these advanced features of the activity diagrams:

- ◆ Naming and renaming activity diagrams
- ◆ Include an activity diagram, but not a statechart, with a package without creating a class
- ◆ Support multiple activities in a package
- ◆ [Associate an object node with a class](#)
- ◆ Create [Adding call behaviors](#) in the activity diagram or by dropping an operation from the browser into the diagram
- ◆ Swimlane association (representing field population) to a class (only) can be created by dragging the class from the browser to the Swimlane name compartment. For more information, see [Swimlanes](#).
- ◆ Reference an alternate activity diagram within the main behavior activity diagram

Actions

Activity diagrams are flowcharts that decompose a system into activities that correspond to states. These diagrammatic elements, called *actions*, are member function calls within a given operation. In contrast to normal states (as in statecharts), actions in activity diagrams terminate on completion of the activity, rather than as a reaction to an externally generated event.

Each action can have an entry action, and must have at least one outgoing activity flow. The implicit event trigger on the outgoing activity flow is the completion of the entry action. If the action has several outgoing transitions, each must have its own guard condition.

Actions have the following constraints:

- ◆ Outgoing transitions from actions do not include an event signature. They can include guard conditions and actions.
- ◆ Actions have non-empty entry actions.
- ◆ Actions do not have internal transitions or exit actions, nor do activities.
- ◆ Outgoing transitions on actions have no triggering events.

Activity diagram elements

The following sections describe how to use the activity diagram drawing tools to draw the parts of an activity diagram. For basic information on diagrams, including how to create, open, and delete them, see [Graphic editors](#).

Activity diagram drawing tools

The **Diagram Tools** toolbar for an activity diagram contains the following tools.

Drawing Tool	Icon Name	Description
	Accept Event Action	Adds this element to an activity diagram so that you can connect it to an action to show the resulting action for an event. This element can specify the following actions: - Event to send - Event target - Values for event arguments This button displays on Diagram Tools when you select the Analysis Only check box when you define the General features of the activity diagram.
	Accept Time Event	Adds an element that denotes the time elapsed since the current state was entered.
	Action	Shows member function calls within a given operation.
	Action Block	Represents compound actions that can be decomposed into actions. Action blocks can show more detail than is possible in a single, top-level action
	Action Pin	Adds an element to represent the inputs and outputs for the relevant action or action block. An action pin can be used on a Call Operation (derived from the arguments). This button displays in on Diagram Tools when you select the Analysis Only check box when defining the general features of the activity diagram. For more information, see Add action pins/activity parameters to diagrams .
	Activity Final	Signifies either local or global termination, depending on where they are placed in the diagram.
	Activity Flow Label	Add or modify a text describing a transition.
	Activity Parameter	Defines a characteristic of an action block. This button displays on Diagram Tools when you select the Analysis Only check box when defining the general features of the activity diagram. For more information, see Add action pins/activity parameters to diagrams .
	Call Behavior	Represents a call to an operation of a classifier. This is only used in modeling and does not generate code for the call operation.
	Call Operation	Represents a call to an operation of a classifier.

Activity diagrams

Drawing Tool	Icon Name	Description
	ControlFlow	Represents a message or event that causes an object to transition from one state to another.
	Decision Node	Shows a branching condition. A decision node can have only one incoming transition and two or more outgoing transitions.
	Dependency	Indicates a dependent relationship between two items in the diagram.
	Diagram Connector	Connects one part of an activity diagram to another part on the same diagram. They represent another way to show looping behavior.
	Fork Node	Separates a single incoming activity flow into multiple outgoing activity flows.
	Initial Flow	Points to the state that the object, use case, or operation enters when the activity starts.
	Join Node	Merges multiple incoming activity flows into a single outgoing activity flow.
	Merge Node	Joins multiple activity flows into a single, outgoing activity flow.
	Object Node	Represents an object passed from the output of the action for one state to the input of the action for another state.
	Send Action	Represents sending actions to external entities. The Send Action is a language-independent element, which is translated into the relevant implementation language during code generation.
	Subactivity	Represents a nested activity diagram. Subactivities represent the execution of a non-atomic sequence of steps of some duration nested within another activity.
	Swimlane Frame	Organizes activity diagrams into sections of responsibility for actions and subactions.
	Swimlane Divider	Divides the swimlane frame using vertical, solid lines to separate each swimlane (actions and subactions) from adjacent swimlanes.

Drawing an action

To draw an action for your activity diagram:

1. Click the Action button  on **Diagram Tools**.
2. Click or click-and-drag in the activity diagram to place the action at the intended location. An action appears on an activity diagram as a rectangle with curved edges.
3. Type a name for the action.
4. Press **Ctrl+Enter** or click the Select arrow in the toolbar to terminate editing.

By default, the action expression, which does not need to be unique within the diagram, is displayed inside the action symbol. For information on modifying the display, see [Displaying an action](#).

Modify the features of an action

Use the Features window for an action to change its features, including its name and action. An action has the following features:

- ♦ **Name** specifies the name of the action. The description of the action can be entered into the text area on the **Description** tab. This description can include a hyperlink. For more information, see [Hyperlinks](#).
 - ♦ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
 - ♦ **Stereotype** specifies the stereotype of the element, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).
 - To select from a list of current stereotypes in the project, click the folder with binoculars  button.
 - To sort the order of the stereotypes, click the up and down arrows  button.
- Note:** The COM stereotypes are constructive; that is, they affect code generation.
- ♦ **Action** specifies an action in an activity diagram. This is the text you typed into the diagram when you created the action.

Displaying an action

You can show the name, action, or description of the action in the activity diagram.

To specify which attribute to display:

1. Right-click the action and select **Display Options** to open the Display Options window.
2. Select the appropriate values.

Activity frames

You might want to create an activity frame in the activity diagram to hold a group of elements and then assign it activity parameters. The parameters indicate inputs and outputs of data to the frame. The frame can also be synchronized with a call behavior.

Creating an activity frame manually

To create an activity frame manually:

1. In the Rhapsody project, add a class.
2. Right-click the class and select **Add New > Diagrams > Activity**.
3. Right-click in the diagram drawing area and select the **Show/Hide Activity Frame** option. The activity frame displays in the diagram.

You may use the Features window to define the frame.

Creating an activity frame automatically

To create an activity frame automatically for each new activity diagram:

1. Open your Rhapsody project.
2. Right-click the project or a package in the browser and select **Features** to open the Features window.
3. On the **Properties** tab, navigate to the `General::Graphics::ShowActivityFrame` property and check the selection box.
4. Click **OK**.
5. Right-click a class and select **Add New > Diagrams > Activity**. The new diagram contains an activity frame.

Synchronizing the pins

To synchronize the call behavior pin with the activity frame pin:

1. Right-click in the activity frame and select **Features**.
2. Select the **General** tab and check **Analysis Only** and click **OK**.
3. Add some activity parameters to the activity frame and set the name, type, and direction of the pins.
4. Create another activity diagram and check **Analysis Only** in the General tab.
5. Drag and drop the first activity frame into the second activity diagram.

This action creates call behavior, that is referencing to the first activity, and small pins with the same name, type and direction as in the referenced activity.

Updating activity pins

To update the activity pins in synchronized activity diagrams, right-click the call behavior and select **Update/Create Activity Pins**.

Action blocks

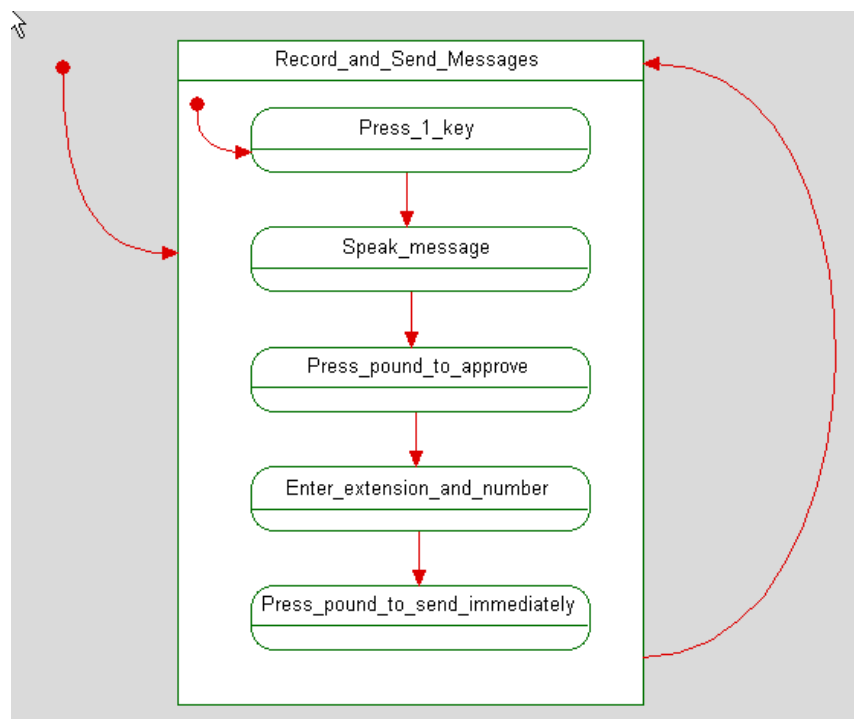
Action blocks represent compound actions that can be decomposed into actions. Action blocks can show more detail than is possible in a single, top-level action. You can also use pseudocode, text, or mathematical formulas as alternative notations. Activity flows inside an action block cannot cross the action block boundary.

Creating an action block

To define the activity, draw an action block:

1. Click the Action Block button  on **Diagram Tools**.
2. Click or click-and-drag in the activity diagram to place the action block at the intended location. An action block appears on an activity diagram as a rectangle.
3. Draw actions and activity flows inside the action block to express the activity being modeled.

The Record_and_Send_Messages activity shown in the following sample action block shows several activities.



Modify the features of an action block

Use the Features window for an action box to change its features, including its name and description. An action block has the following features:

- ♦ **Name** specifies the name of the action block. The description of the action block can be entered into the text area on the **Description** tab. This description can include a hyperlink. For more information, see [Hyperlinks](#).
- ♦ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ **Stereotype** specifies the stereotype of the action block, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).
 - To select from a list of current stereotypes in the project, click the folder with binoculars  button.
 - To sort the order of the stereotypes, click the up and down arrows  button.

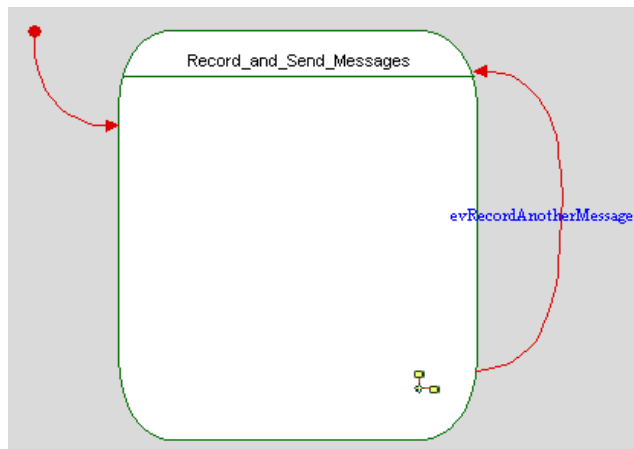
Note: The COM stereotypes are constructive; that is, they affect code generation.

Creating a subactivity from an action block

You can convert an action block to a subactivity. This moves the contents of the block into a separate subchart, and simplifies the diagram containing the action block.

To create a subactivity from an action block, right-click the action block and select **Create Sub Activity Diagram**. Rational Rhapsody creates a new subchart containing the former contents of the action block.

Consider the action block shown in the following figure. When you create a subactivity for the action block, the parent state changes to that shown in the following figure.



The icon in the lower right corner indicates that the action block has a subchart.

Subactivities

Subactivities represent nested activity diagrams. Subactivities represent the execution of a non-atomic sequence of steps of some duration nested within another activity. Internally, a subactivity consists of a set of actions, and possibly a wait for events. In other words, a subactivity is a hierarchical action during which an associated subactivity diagram is executed. Therefore, a subactivity is a submachine state that executes an activity diagram.

The nested activity diagram must have an initial (default) state and a final activity (see [Local termination semantics](#)). When an input activity flow to the subactivity is triggered, execution begins with the initial state of the nested activity diagram. The outgoing activity flows from a subactivity are enabled when the final activity of the nested activity diagram is reached (when the activity completes) or when triggering events occur on the transitions. Because states in activity diagrams normally do not have triggering events, subactivities are normally exited when their nested graph is finished.

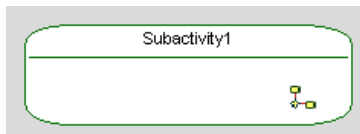
Many subactivities can start a single-nested activity diagram.

Creating a subactivity

To draw a subactivity:

1. Click the Subactivity button  on **Diagram Tools**.
2. Click (or click-and-drag) in the activity diagram to place the subactivity at the intended location.

A subactivity looks like an action.



Opening a subactivity diagram

To open a subactivity diagram, right-click the subactivity and select **Open Sub Activity Diagram**.

Creating a final activity

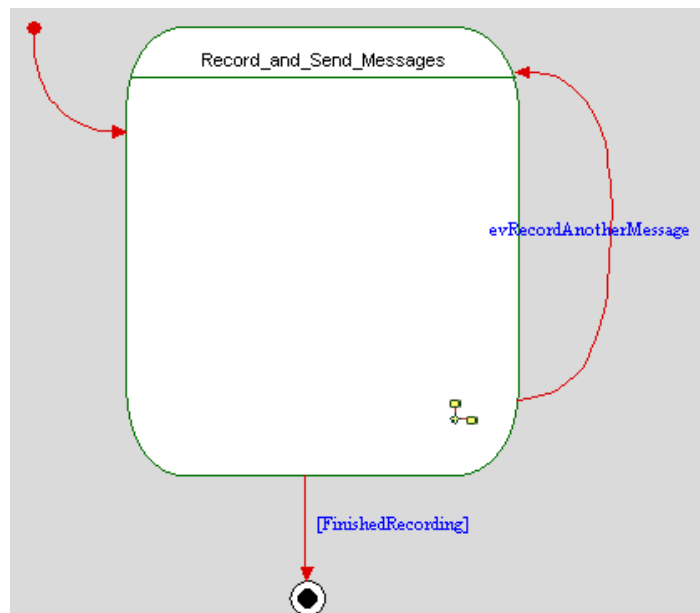
UML final state (“*activity final*” in Rational Rhapsody) can signify either local or global termination, depending on where they are placed in the diagram. When the state is drawn inside a composite (block) state, it is considered a final state. This terminates the activity represented by the composite state, but not the instance performing the activity. For more information, see [Local termination semantics](#). When the state is drawn inside the top state, it is considered a final activity. This terminates the state machine causing the instance to be destroyed.

Note

The behavior of “activity final” is controlled by the `CG::Statechart::LocalTerminationSemantics` property.

To create a final activity:

1. Click the Activity Final button .
2. Click in the activity diagram to place the final activity at the intended location.
3. Draw an activity flow from any kind of state to the final activity.
4. If wanted, enter a guard condition to signal the end of the activity. A final activity appears as a circle with a black dot in the middle:



As with the other connectors, final activities and their activity flows are included in the browser view.

Object nodes

Actions operate on objects and are used by objects. These objects either initiate an action or are used by an action. Normally, actions specify calls sent between the object owning the activity diagram (which initiates actions) and the objects that are the targets of the actions. An *object node* represents an object passed from the output of the action for one state to the input of the action for another state.

Note

An object node can only be a leaf element; its meaning cannot contain other elements.

As with other states, only one object node can have the same name in the same parent state.

Creating an object node

To create an object node:

1. Click the Object Node button on **Diagram Tools**.
2. Click or click-and-drag in the activity diagram to place the node at the intended location.

Display options for an object node

After creating an object node, right-click the diagram and select **Display Options**. Make your selections from these settings:

- ♦ **Display name - Represents only, Name, or Label.**
- ♦ **Show Stereotype Label**
- ♦ **View Image - Enable Image View** activates **Use Associated Image** or **Select on Image** with **Image Path**
- ♦ **Advanced view image features - Image Only, Structured, or Compartment**

Object node features

The Features window enables you to change the features of an object node, including its name and stereotype.

- ♦ **Name** specifies the name of the object node. The default name is `state_n`, where `n` is an incremental integer starting with 0. The description of the action block can be entered into the text area on the **Description** tab.
- ♦ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ **Stereotype** specifies the stereotype of the object node, if any. They are enclosed in guillemets, for example «`s1`» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).
 - To select from a list of current stereotypes in the project, click the folder with binoculars  button.
 - To sort the order of the stereotypes, click the up and down arrows  button.

Note: The COM stereotypes are constructive; that is, they affect code generation.

- ♦ **In State** specifies the required states of the object node.
- ♦ **Represents** lets you select a class/type to associate with the node from a list of project classes/types or enter the name of a new class. For more information, see [Associate an object node with a class](#).

Associate an object node with a class

In an activity diagram, you can associate an object node to a class/type to represent the class type using one of the following methods:

- ♦ Right-click the object node in the diagram to display the Features window and select the class/type from the list in the **Represents** field.
- ♦ Dragging a class/type from the browser into the activity diagram automatically associates the class/type with the object node.

In the activity diagram, the associated node displays the name of its associated class/type in the name compartment.

Adding call behaviors

A *call behavior* represents a call to an operation of a classifier. This is only used in modeling and does not generate code for the call operation. However, code can be inserted manually into the **Action** field in the Call Behavior Features window. To create a call operation node, use one of these methods:

- ◆ Click the Call Behavior button  on **Diagram Tools** and drawing the behavior operation in the diagram.
- ◆ Click or click-and-drag an operation from the browser in the activity diagram and place the behavior at the intended location.

Note

If the behavior operation dropped in the activity diagram has no association with the classifier, an empty call is created.

Activity flows

Activity diagrams can have activity flows, default flows, and loop activity flows on action blocks and join activity flows. These activity flows in activity diagrams are the same as the corresponding activity flows in statecharts, with the following exceptions:

- ◆ Outgoing activity flows from states can have triggers, but those from actions, action blocks, or subactivities cannot.
- ◆ Outgoing activity flows from actions, action blocks, and subactivities can have only guards and actions.

Activity flows exiting or entering fork or join bars have the following constraints:

- ◆ They cannot have labels.
- ◆ They must originate in, or target, either states or actions (not connectors).

Creating an activity flow

To draw an activity flow from one state to another:

1. Click the Activity Flow button  on **Diagram Tools**.
2. Click the edge of the source state.
3. Drag the cursor to the edge of the target state and release to anchor the activity flow.
4. If wanted, enter a trigger for the activity flow.
5. If wanted, enter a guard and action for the activity flow.

Completion activity flows

An activity flow to a final activity is called a *completion activity flow*. Neither final states nor a final activity can have outgoing transitions. A completion activity flow does not have an explicit trigger, although it can have a guard condition.

Drawing initial flows

The default flow points to the state that the object, use case, or operation enters when the activity starts.

To draw a default flow:

1. Click the Initial Flow button  on **Diagram Tools**.
2. Click in the activity diagram outside the default state.
3. Drag the cursor to the edge of the default state of the activity and release the mouse button.

For more information on default flows, see [Statecharts](#).

Drawing loop activity flows

Loop activity flows (also known as *self transitions*) represent looping behavior in a program. Loop activity flows are often used on action blocks to indicate that the block should loop until some exit condition becomes true.

To draw a loop activity flow:

1. Click the Loop Activity Flow button  on **Diagram Tools**.
2. Click the edge of any kind of state.
3. Label the loop activity flow.

For an example of an action block with a loop activity flow, see [Creating a final activity](#). For more information on loop (self) transitions, see [Statecharts](#).

Adding or modifying activity flow labels

To add or modify a activity flow label:

1. Click the Activity Flow Label button  on **Diagram Tools**.
2. Select the transition you want to label.
3. In the edit box, type the new label (or modify the existing one).
4. Press **Ctrl+Enter** or click outside the label to terminate editing.

Modify activity flows

As with all other elements, you can modify the features of a activity flow using the Features window. For more information, see [Features of transitions](#).

Connectors

Activity diagrams can have the following connectors:

- ♦ Merge nodes
- ♦ Condition
- ♦ Diagram

The following sections describe these connectors in detail.

Drawing merge nodes

To draw a merge node:

1. Click the Merge Node button .
2. Click in the activity diagram to place the junction at the intended location.
3. Draw activity flows going into, and one activity flow going out of the junction.
4. Label the activity flows as wanted.

For more information on merge nodes, see [Statecharts](#).

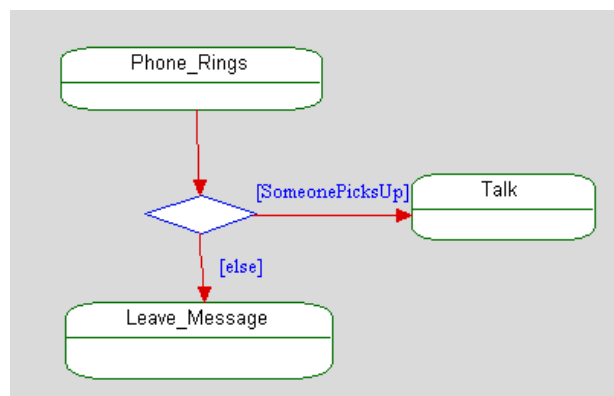
Drawing decision nodes

Decision nodes show branching conditions. A decision node can have only one incoming activity flow and two or more outgoing activity flows. The outgoing activity flows are labeled with a distinct guard condition and no event trigger. A predefined guard, denoted `[else]`, can be used for no more than one outgoing activity flow.

To draw a decision node:

1. Click the Decision Node button .
2. Click, or click-and-drag, in the activity diagram to position the decision node at the intended location. A decision node appears on an activity diagram as a diamond.
3. Draw at least two states that will become targets of the outgoing activity flows.
4. Draw an incoming activity flow from the source state to the decision node.
5. Draw and label the outgoing activity flows from the decision node to the target states.

This activity diagram shows the following behavior: When the phone rings, if someone picks up on the other end, you can talk; otherwise, you must leave a message. The decision node represents the decision point. In other words, after the `PhoneRings()` operation, if `SomeonePicksUp` resolves to `True`, the `Talk()` operation is called. Otherwise, the `LeaveMessage()` operation is called.



Use the Display Options window to determine whether to display the name, label, or nothing for the decision node.

Drawing diagram connectors

Diagram connectors connect one part of an activity diagram to another part on the same diagram. They represent another way to show looping behavior.

To draw a diagram connector:

1. Click the Diagram Connector button  on **Diagram Tools**.
2. Click to place the source diagram connector at the intended location and label the connector.
3. Repeat to place the target diagram connector at the intended location in the diagram, and give it the same name as the source connector.

For more information on diagram connectors, see [Statecharts](#).

Join or fork bars

Activity diagrams can include join or fork bars. A *join or fork bar* depicts either a join or a fork operation. You can draw join or fork bars in activity diagrams for objects, use cases, and operations.

Rational Rhapsody defines activity diagrams as meaningful only if join and fork bars are well-structured in the same sense as well-structured parentheses. In other words, they must use proper nesting. The only exception to this rule is that a join or fork connector with multiple ingoing/outgoing activity flows can be used in place of a number of join or fork connectors with only two ingoing or outgoing activity flows each.

Rational Rhapsody tolerates less-than-meaningful activity charts, provided that they can be extended into meaningful ones by adding activity flows (for example, a fork with activity flows that never merge back).

As you draw activity diagrams, Rational Rhapsody prevents you from drawing constructs that violate the meaningfulness of the activity diagram by displaying a “no entry” symbol.

Creating join nodes

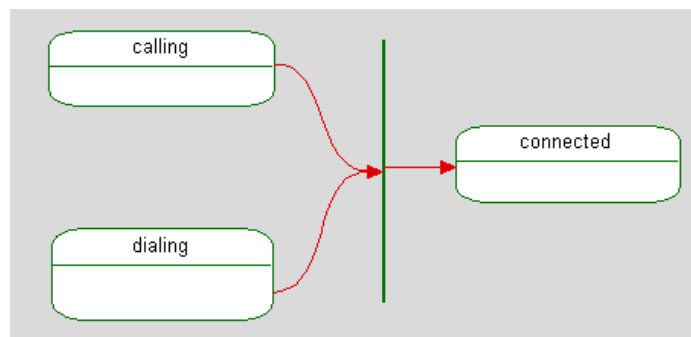
To draw a join nodes:

1. Draw at least two states that you want to synchronize.
2. Click the Join Node button  on **Diagram Tools**.
3. Click or click-and-drag in the activity diagram to place the join node bar in the intended location.

By default, the join bar is drawn horizontally. To flip it, see [Rotating join or fork bars](#).

4. Draw incoming activity flows from each of the source states to the join node bar.
5. Draw a state that will be the target of the outgoing activity flow.
6. Draw an outgoing activity flow from the join node bar to the target state.

A join node is shown as a bar with two or more incoming activity flow arrows and one outgoing activity flow arrow.



Creating fork nodes

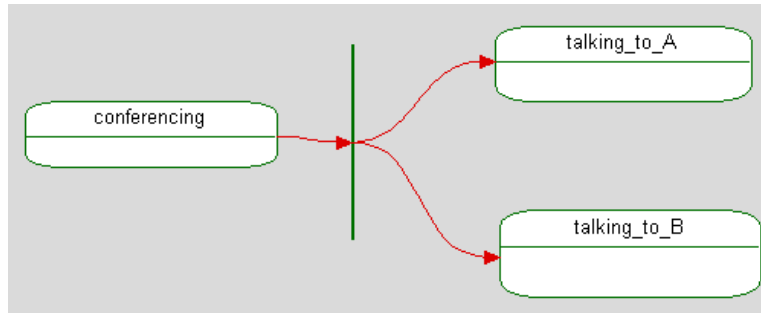
To draw a fork nodes:

1. Click the Fork Node button  on **Diagram Tools**.
2. Click, or click-and-drag, in the activity diagram to place the fork node bar at the intended location. Fork node bars can be vertical or horizontal only.

By default, the join line is drawn horizontally. To flip it, see [Rotating join or fork bars](#).

3. Draw a source state and an incoming activity flow coming into the fork node bar.
4. Draw the target states and the outgoing activity flows from the fork node bar.

A fork node is shown as a heavy bar with one incoming activity flow and two or more outgoing activity flow arrows.



Rotating join or fork bars

You can rotate a join or fork bar to the right (clockwise) or left (counter-clockwise).

To rotate a join or fork bar, right-click the join or fork bar and select either **Flip Right** or **Flip Left**.

Stretching join or fork bars

To stretch a join or fork bar:

1. Select the join or fork bar.
2. Click-and-drag one of the highlighted selection handles until the join or fork bar is the wanted length.

Moving join or fork bars

To move a join or fork bar to a new location:

1. Click in the middle of the join or fork bar, away from the selection handles, and drag the bar to the intended location.
2. Release the mouse button to place the join or fork bar at the new location.

Modify join or fork bars

As with all other elements, you can modify the features of a join or fork bar using the Features window. The fields and buttons are as follows:

- ♦ **Name** specifies the name of the element. The default name is <element>_n, where *n* is an incremental integer starting with 0. Add any additional information using the **Description** tab.
- ♦ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ **Stereotype** specifies the stereotype of the element, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).
 - To select from a list of current stereotypes in the project, click the folder with binoculars  button.
 - To sort the order of the stereotypes, click the up and down arrows  button.

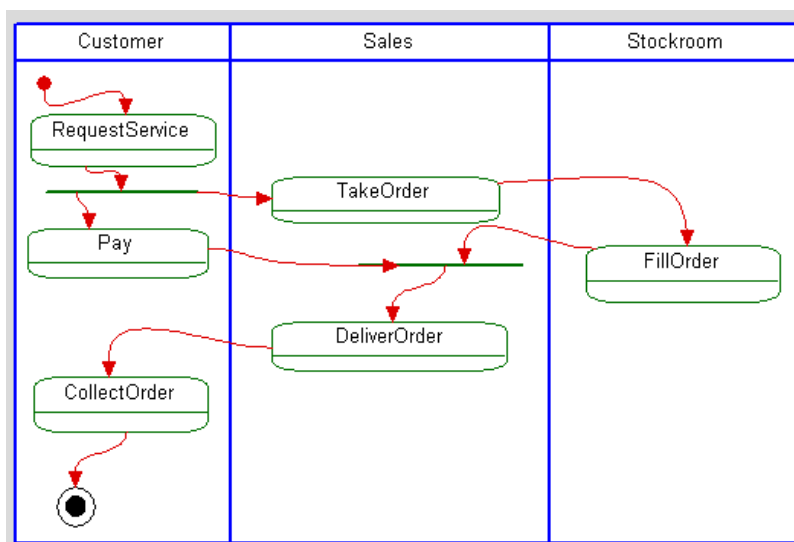
Note: The COM stereotypes are constructive; that is, they affect code generation.

Swimlanes

Swimlanes divide activity diagrams into sections. Each swimlane is separated from adjacent swimlanes by vertical, solid lines on both sides. These are the features of swimlanes:

- ◆ Each action is assigned to one swimlane.
- ◆ Activity flows can cross lanes.
- ◆ Swimlanes do not change ownership hierarchy.
- ◆ Swimlane association (representing field population) to a class (only) can be created by dragging the class from the browser to the Swimlane name compartment.
- ◆ The relative ordering of swimlanes has no semantic significance.
- ◆ There is no significance to the routing of an activity flow path.

The following figure shows an activity diagram with swimlanes.



Creating swimlanes

To use swimlanes in an activity diagram, you first need to create a swimlane frame. If you do not, Rational Rhapsody generates an error message.

Note

There can be only one swimlane frame in an activity diagram. Once you have created a frame, the **Swimlane Frame** tool is unavailable.

To draw a swimlane:

1. Click the Swimlane Frame button  on **Diagram Tools**.
2. The cursor turns into crosshairs. In the drawing area, click one corner to draw the swimlane frame (a box).
3. Click the Swimlane Divider button  on **Diagram Tools**.
4. The cursor turns into a vertical bar. When it is at the intended location, left-click to place the divider. Rational Rhapsody creates two swimlanes, named `swimlane_n` and `swimlane_n+1`, where n is an incremental integer starting at 0.

If you draw another divider, it divides the existing swimlane into two swimlanes, with the new swimlane positioned to the *left* of the divider.

Note: You cannot draw a swimlane on an existing state.

5. If wanted, rename the swimlanes using the Features window.

Note the following information:

- ♦ Swimlanes have a minimum width. If you enlarge a swimlane, the extra space is added to the right of the swimlane. To resize a swimlane, move the divider to the left or the right.
- ♦ If a swimlane contains activity diagram elements, you cannot reduce the size of that swimlane so its divider is positioned to the left of any of those elements, because that would force the elements into a different swimlane.
- ♦ A swimlane maps into a partition of states in the activity diagram. A state symbol in a swimlane cases the corresponding state to belong to the corresponding partition.

Modify the features of a swimlane

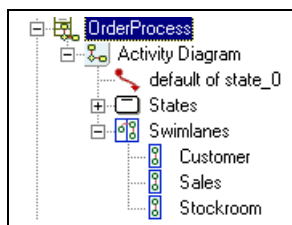
The Features window enables you to change the features of a swimlane, including its name and description. A swimlane has the following features:

- ♦ **Name** specifies the name of the element. The default name is `swimlane_n`, where *n* is an incremental integer starting with 0. Add any additional information using the **Description** tab.
- ♦ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ **Stereotype** specifies the stereotype of the swimlane, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).
 - To select from a list of current stereotypes in the project, click the folder with binoculars  button.
 - To sort the order of the stereotypes, click the up and down arrows  button.
- ♦ **Represents** specifies the class to which the swimlane applies.

Note: The COM stereotypes are constructive; that is, they affect code generation.

View swimlanes in the browser

Swimlanes are displayed in the browser under the activity diagram, as shown in the following figure.



Note

Swimlane nodes cannot be deleted.

Deleting swimlane dividers

To delete a swimlane:

1. Select the divider you want to delete.
2. Click the **Delete** button on your keyboard.

Deleting the swimlane frame

To delete the swimlane frame and all its swimlanes:

1. Select the frame.
2. Click the **Delete** button on your keyboard.

Note that after the deletion, the frame and swimlanes are removed, but the elements they contained still exist in the activity diagram.

Swimlane limitations

Note the following design and behavior limitations apply to swimlanes:

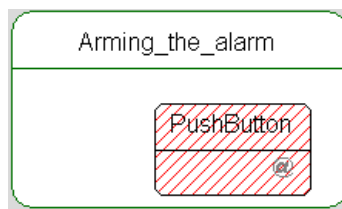
- ◆ You should not draw actions on swimlane divider lines. In other words, do not draw actions that overlap into another swimlane.
- ◆ Swimlanes in subactivity diagrams are not supported.
- ◆ You cannot use the browser to create or drag-and-drop swimlanes.

Adding calls to behaviors

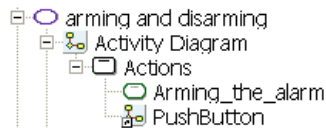
You can add a call to a behavior in another activity diagram or to the entire activity diagram. You can add calls to both activity diagrams and subactivity diagrams.

To add a call, either click the **Call Behavior** button  on **Diagram Tools**, or drag-and-drop the activity (or activity diagram) from the browser into the activity diagram. Rational Rhapsody creates the call in the activity diagram and in the browser.

The called behavior has the same name as the called object. The called behavior, `PushButton`, is marked with an at-symbol icon, as shown in the following figure.



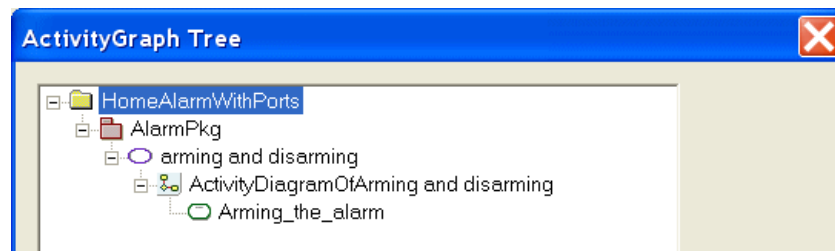
This is displayed in the project browser under the activity diagram containing the behavior.



Modifying a called behavior

To modify the features of a called behavior:

1. Highlight the called behavior in either the browser or diagram. Right-click and select **Features**.
2. Click the right arrow button beside the **Reference** field in the **General** tab.
3. Use the ActivityGraph Tree window to navigate to the activity diagram that the called behavior represents.



Display called behaviors

As with most elements, use the Display Options window to define the display of called behaviors with one of these options:

- ◆ Show the name, label, or nothing for the activity
- ◆ Show the name, label, or icon for the stereotype

Called behavior limitations

Note the following behavior and restrictions:

- ◆ You cannot “undo” changes to called behaviors.
- ◆ A called behavior cannot be created in the browser.
- ◆ There is no code generation for called behaviors because the code generator ignores these calls and the activity flows that go in and out of them.

Add action pins/activity parameters to diagrams

Action pins can be added to actions and activity parameters can be added to action blocks in an activity diagram. These elements represent the inputs and outputs for the relevant action/action block. Action pins can also be added to subactivities.

Action pins and activity parameters are diagram elements and appear in the browser. However, there is no code generated for these elements.

Making the action pin tool available

To make the action pin tool available for use:

1. Highlight an element in the browser that needs an activity diagram for analysis purposes only.
2. Right-click the element and select **Add New > Diagrams > Activity Diagram**.
3. Right-click the Activity Diagram now listed in the browser and select **Features** to open the Features window.
4. Select the **Analysis Only** check box on the **General** tab.
5. Click **OK**. Rational Rhapsody displays a message asking if you intend to create an analysis only activity diagram. If you click **Yes**, this diagram cannot be used for other purposes than analysis.
6. The system then places the **Action Pin** and **Activity Parameter** buttons on **Diagram Tools**.

Using the action pin

To use the action pin:

1. Click the small Action Pin button  at the bottom of the available **Diagram Tools**.
2. Click the action to which the pin should be added.

The pin displays on the border of the action closest to the point that was clicked. The name displays alongside the pin. The default name is `pin_n`.

Adding an activity parameter

To add an activity parameter:

1. Click the Activity Parameter button  at the bottom of the **Diagram Tools**.
2. Click the action block to which the activity parameter should be added.

The activity parameter displays on the border of the action block closest to the point that was clicked. The name displays inside the activity parameter node. The default name is `parameter_n`.

Modify features of action pins / activity parameters

The Features window enables you to change the following features of action pins / activity parameters, in addition to the standard fields:

- ◆ Direction (in, out, inOut)
- ◆ Argument Type (for example, int)

Graphical characteristics of action pins / activity parameters

Action pins and activity parameter nodes have the following graphical characteristics:

- ◆ Pins/parameters always remain attached to their action/action block. However, they can be moved around the perimeter of the action/action block.
- ◆ Action pins are a fixed size. Activity parameters, however, can be resized.
- ◆ Pins/parameters cannot be copied and pasted.
- ◆ All pin/parameter operations can be undone.
- ◆ In the browser, pins/parameters appear beneath the action/action block to which they belong.
- ◆ When an action pin / activity parameter is deleted from a diagram, it is removed from both the view and the model.

Other characteristics of action pins / activity parameters

- ♦ Appear in reports generated by the internal Rational Rhapsody reporter.
- ♦ Appear in reports generated by ReporterPLUS.
- ♦ Are exported to DOORS.
- ♦ Are exported to XMI files using the XMI toolkit.
- ♦ Supported in DiffMerge.
- ♦ Appear in the Search window.
- ♦ Supported in the Rational Rhapsody API. It returns the pins/parameters associated with an action, as well as the type of each pin/parameter.

Local termination semantics

Local termination means that once an action block is entered, it can be exited by a null transition only if its final state has been reached. A null transition is any transition without a trigger (event or timeout). A null transition can have a guard.

Statechart mode

The following sections describe how local termination is implemented in statechart mode for various kinds of states.

Or states

The following local termination rules apply to Or states:

- ♦ An Or state that has a final activity is completed when the final activity is reached.
- ♦ An Or state without a final activity is completed after finishing its entry action.
- ♦ An outgoing activity flow from an Or state can have a trigger (as with statecharts).
- ♦ An outgoing null transition (activity flow without a trigger) from an Or state can be taken only if the Or state is completed.
- ♦ If an Or state with a final activity has a history connector, the last state of the history connector is always the final activity, after it has been reached once.
- ♦ An Or state can be exited by any activity flow, including a null transition, from one of its substates (as with statecharts).

Leaf states

A leaf state is completed after finishing its entry action.

Component states

The following local termination rules apply to component states:

- ♦ Because a component state is a kind of Or state, all the local termination rules for Or states also apply to component states.
- ♦ A join transition (from several component states) is activated only if all the sources (component states) are completed.

And states

An outgoing null transition from an And state is activated only if all of its components are completed.

IS_COMPLETED() macro

You can use the `IS_COMPLETED()` macro in a statechart to test whether a state is completed. Completion means that any of the conditions for local termination described in the previous sections are true. The macro works the same for both flat and reusable implementations of statecharts.

The `CG::Class::IsCompletedForAllStates` property specifies whether the `IS_COMPLETED()` macro can be used for all kinds of states. The default value of `Cleared` means that the macro can be used only for states that have a final activity. `Checked` means that it can be used for all states.

Activity diagram mode

All of the local termination rules for statechart mode also apply in activity diagram mode, with the following exceptions:

- ♦ An Or state that has a final activity is completed when the final activity is reached.
- ♦ An Or state without a final activity is never completed.

Code generation

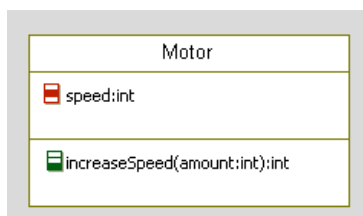
In previous releases, Rational Rhapsody supported code generation only from activity diagrams associated with classes, not from activity diagrams associated with operations or use cases.

Rational Rhapsody Developer for C++ generates functor-based code for activity diagrams associated with operations. *Functor-based code* reuses the code generation functionality for activity diagrams of classes. Code is generated into a new class (called the *functor class*), which implements an activity diagram on the class level. The task of executing the *modeled operation* (an operation associated with an activity diagram) is delegated to the new class. The class that delegates the task is known as the *owner class*.

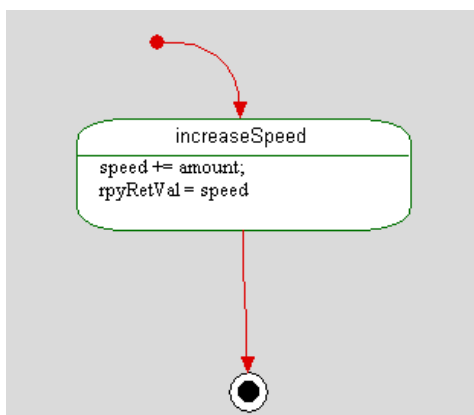
You specify whether to generate code for an activity diagram by setting the property `CPP_CG::Operation::ImplementActivityDiagram` to `Checked`. This property is set to `Cleared` by default.

Functor classes

Consider the following class:



The following figure shows the activity diagram associated with the `increaseSpeed` modeled operation.



In the example model, `Motor` is the owner class, and `FunctorIncreaseSpeed_int` executes the modeled operation `increaseSpeed`.

This activity diagram increments the value of the class attribute `speed` by the value specified by the `amount` argument, then returns the incremental value. Note the following information:

- ◆ An activity diagram should never contain code with the `return` keyword because the current implementation executes the code fragments of the diagram (that appear in actions, activity flows, and so on) in contexts of different operations. Returning from those contexts (operations) does not have the same effect as returning from the body of a regular operation.

Instead, you should set the return value to the `rpyRetVal` variable (shown in the previous figure), which is always generated for operations that return values.

- ◆ There are two ways an operation can finish and return control to its caller:
 - Once the diagram reaches a “stable” state (there are no more activity flows to take), the operation is considered to have finished its job and it returns control to its caller.
 - The diagram reaches a termination connector at the top level.
- ◆ The actual execution of the code does not occur within the scope of the object that owns the operation. Instead, the code is executed in another object whose sole purpose is to execute the diagram. During the execution of the code in the diagram, the `this` pointer references a special-purpose object (the *functor object*) that contains the code in the diagram (in states, on activity flows, and so on). To refer to the owner object, you can use the `this_` variable, which always exists for this purpose.

To make coding more natural, direct access to the attributes of the class that owns the operation (without the need for `this_`) is made possible by supplying references in the functor object. The functor class contains references that correspond to each owner class attribute and have the same name. The constructor of the functor class contains arguments to initialize each attribute; initialization is done when the owner class creates an object of the functor class.

You can control the code generation of the attribute references by setting the value of the `CPP_CG::Operation::ActivityReferenceToAttributes` property to `Checked` (the default value).

Limitations and specified behavior

Note the following restrictions and behavior:

- ♦ Activity diagrams do not require “And” states to represent concurrent behavior. Activity diagrams cannot include “And” states.
- ♦ Forks can end in states that are not within two orthogonal states.
- ♦ Activity diagrams for operations cannot receive events, nor can they have state nodes.
- ♦ You cannot animate the activity diagrams associated with operations. However, an animated sequence diagram for the model records the fact that a modeled operation was called. In addition, the call and object stacks record the owner object (not the functor object) as the object that receives the message.
- ♦ This feature supports only a subclass of diagrams that do not contain events (including timeout events) or triggered operations.
- ♦ If a class attribute and an argument of the modeled operation have the same name, there will be a name collision that results in a compiler error. To avoid this problem before attempting to generate code, omit the class attribute that causes the collision. This should not affect the semantics of the operation, because operation arguments hide class attributes.
- ♦ The name of the functor class will contain the signature of the modeled operation, thereby supporting overloaded operations. Complicated type names will be converted to strings appropriate for building C++ identifiers. For example, “::” characters are replaced with underscores.
- ♦ Because the code in the diagram does not execute within the scope of the modeled operation, there is no direct access to the `this` variable. Instead, access this variable using the `this_` attribute of the functor class. This is also the preferred way to start methods in the owner class.
- ♦ Operations with variable-length argument lists are not supported.
- ♦ Global operations with activity diagrams are not supported.

Flow charts

A *flow chart* is a schematic representation of an algorithm or a process. In UML and Rational Rhapsody, you can think of a flow chart as a subset of an activity diagram that is defined on methods and functions.

You can model methods and functions using flow charts in all Rational Rhapsody programming languages. Only in Rational Rhapsody in C and Rational Rhapsody in C++ can readable structured code be generated from a flow chart. During code generation, for the actions defined in a flow chart Rational Rhapsody can generate structured code for If/Then/Else, Do/While, and While/Loops.

The code generator algorithm for a flow chart can identify Loops and Ifs, the expressions for these constructs is on the guards of the action flows.

For more information about activity diagrams, see [Activity diagrams](#).

Define algorithms with flow charts

One useful application of flow charts is in the definition of algorithms. *Algorithms* are essentially decompositions of functions into smaller functions that specify the activities encompassed within a given process.

This flow chart approach to code generation is to reduce the diagram to blocks of sequential code and then search for If/Loop patterns in those blocks. The following structured programming control structures are supported in flow charts:

- ◆ Simple If
- ◆ If/Then/Else
- ◆ While loops (where the test is at the start of the loop)
- ◆ Do/While loops (where the test is at the end of the loop)

If the algorithm does not succeed to impose the above structure, then it will need to use a GoTo.

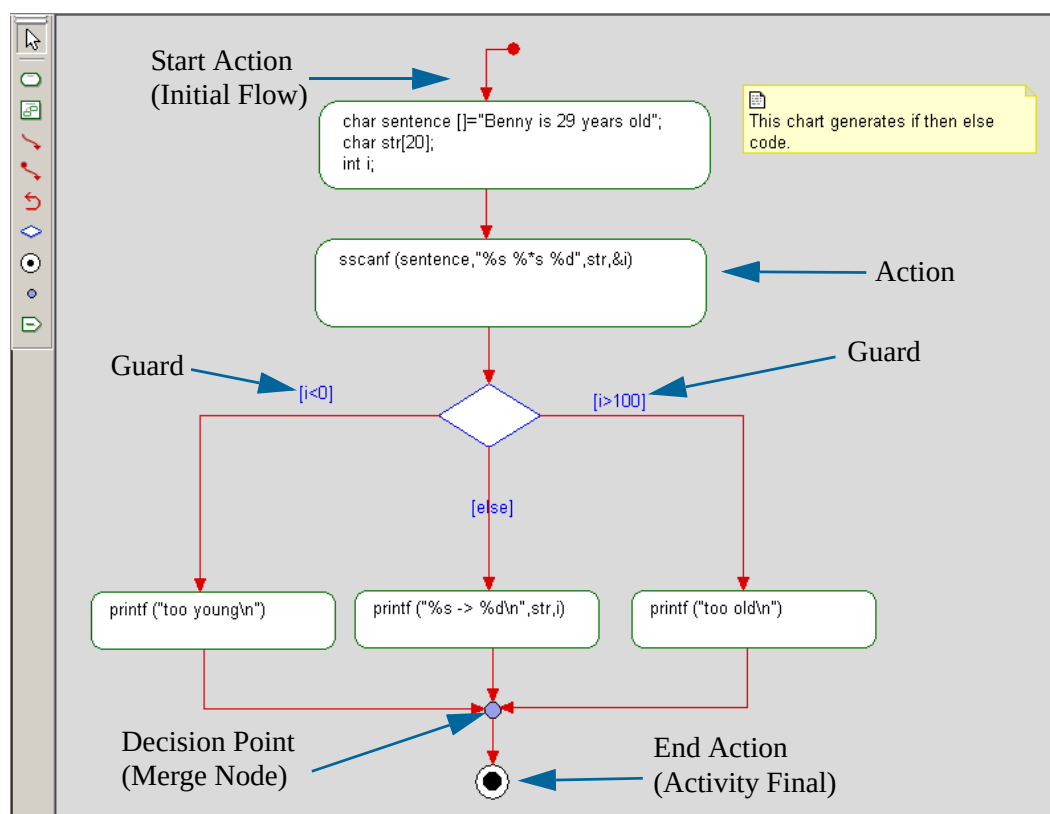
Flow charts similarity to activity diagrams

Flow charts have the following elements in common with activity diagrams including start and end activities and *actions*:

- ◆ **Decision points** that show branching points in the program flow based on guard conditions.
- ◆ **Actions** that represent function invocations with a single exit action flow taken when the function completes. It is not necessary for all actions to be within the same object.
- ◆ **Action blocks** that represent compound actions that can be decomposed into actions.

However, flow charts *do not* include And states, and flow charts for operations cannot receive events.

Rational Rhapsody in C includes a Flowchart model located in the <Rational Rhapsody installation>\Samples\CSamples\Flowchart folder. These sample flow charts show which flow chart patterns are recognized in order to generate structured code. The following illustration shows the main elements of a flow chart:



Create flow chart elements

You use the flow chart drawing tools to draw the parts of a flow chart. For basic information on diagrams, including how to create, open, and delete them, see [Graphic editors](#).

Tools for drawing flow charts

The **Diagram Tools** for a flow chart contains the following tools.

Drawing Buttons	Button Name	Description
	Action	Represents the member function call within a given operation. For more information, see Actions .
	Action Block	Represents compound actions that can be decomposed into actions. For more information, see Action blocks .
	Activity Flow	Defines the flow and its guards to supply the test for ifs or Loops. Activity flows without guards define the default sequential flow of the flow chart. For more information, see Activity flows .
	Initial Flow	Shows the flow origination point from an element. For flow charts, this default flow is the initial flow, and for code purposes, indicates the start of code. For more information, see Drawing initial flows .
	Loop Activity Flow	Represents behavior repeating in a program. For more information, see Drawing loop activity flows .
	DecisionNode	Shows branching conditions. For more information, see Drawing decision nodes .
	ActivityFinal	Provides local termination semantics. The flow chart returns at this point to the operation/function that started it. For more information, see Activity final .
	MergeNode	Combines different flows to a common target. For more information, see Drawing merge nodes .
	Send Action	Represents the sending of events to external entities. For more information, see Send action elements .

Actions

Flow charts decompose a system into actions that correspond to activities. These diagrammatic elements, called *actions*, are member function calls within a given operation. In contrast to normal states (as in statecharts), actions in flow charts terminate on completion of the activity, rather than as a reaction to an externally generated event.

Each action can have an entry action, and must have at least one outgoing action flow. The implicit event trigger on the outgoing action flow is the completion of the entry action. If the action has several outgoing action flows, each must have its own guard condition.

During code generation, code is derived from the actions on a flow chart.

Actions have the following constraints:

- ◆ Outgoing activity flows can include only guard conditions.
- ◆ Actions have non-empty entry actions.
- ◆ Actions do not have internal action flows or exit actions, nor do activities.
- ◆ Outgoing action flows on actions have no triggering events.

Creating a flow chart

You can create a flow chart on any function or class method in the same way as you can an activity diagram.

To create a flow chart:

1. In the browser, right-click the model element for which you want to create a flow chart, such as a function.
2. Select **Add New > Diagrams > Flowchart**.

The flow chart displays in the Drawing area.

Drawing an action

To draw an action:

1. Create a flow chart and click the Action button  on **Diagram Tools**.
2. Click or click-and-drag in the flow chart to place the action where you want it. An action appears on a flow chart as a rectangle with curved edges.
3. Type a name for the action, then press **Ctrl+Enter** or click the Select arrow in the toolbar to terminate typing mode.

By default, the action expression, which does not need to be unique within the flow chart, is displayed inside the action symbol. For information on modifying the display, see [Displaying an action](#).

Modify the features of an action

The Features window enables you to add and change the features of an action, including its name and action. An action has the following features:

- ♦ **Name** specifies the name of the action. The description of the action can be entered into the text area on the **Description** tab. This description can include a hyperlink. For more information, see [Hyperlinks](#).
- ♦ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ **Stereotype** specifies the stereotype of the element, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#). By default, flow charts have a new term stereotype of “flowchart.”
Note: The COM stereotypes are constructive; that is, they affect code generation.
- ♦ **Action** specifies an action in a flow chart. This is the text you typed into the flow chart when you created the action.

The **Overridden** check box allows you to toggle the check box on and off to view the inherited information in each of the window fields and decide whether to apply the information or revert back to the currently overridden information.

For more information about the Features window, see [The Features window](#).

Displaying an action

You can show the name, action, or description of the action in the flow chart.

To specify which attribute to display:

1. Right-click the action and select **Display Options** to open the Display Options window.
2. Select the appropriate values and click **OK**.

Action blocks

Action blocks represent compound actions that can be decomposed into actions. Action blocks can show more detail than might be possible in a single, top-level action. You can also use pseudocode, text, or mathematical formulas as alternative notations.

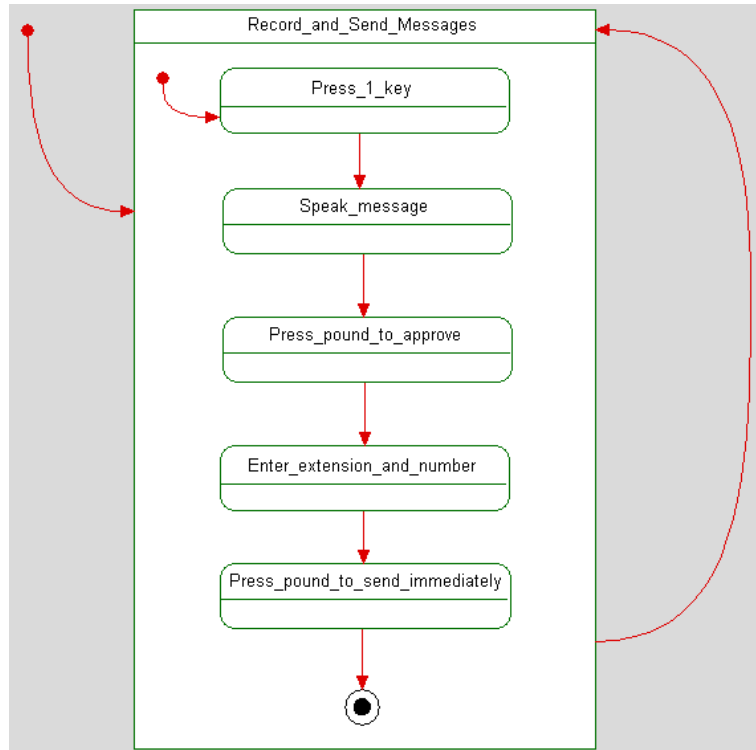
Note that for action blocks, there must be a default flow at the start and a final activity at the end, and activity flows cannot cross the blocks boundaries to actions in the block (though they might enter and leave the block itself). The code generator will put blocks code in curly braces and this has language significance regarding variable scope and lifetime.

Creating an action block

To define the activity, draw an action block:

1. Click the Action Block button  on **Diagram Tools**.
2. Click or click-and-drag in the flow chart to place the action block where you want it. Action blocks appear as rectangles on a flow chart.
3. Draw actions and activity flows inside the action block to express the activity being modeled.

The Record_and_Send_Messages activity shown in the following sample action block encompasses several activities.



Modify the features of an action block

The Features window enables you to change the features of an action block, including its name and description. An action block has the following features:

- ◆ **Name** specifies the name of the action block. The description of the action block can be entered into the text area on the **Description** tab. This description can include a hyperlink. For more information, see [Hyperlinks](#).
- ◆ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ◆ **Stereotype** specifies the stereotype of the action block, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#). By default, flow charts have a new term stereotype of “flowchart.”

Note: The COM stereotypes are constructive; that is, they affect code generation.

Activity final

An *activity final* provides local termination semantics. The flow chart returns at this point to the operation/function that started it.

Note

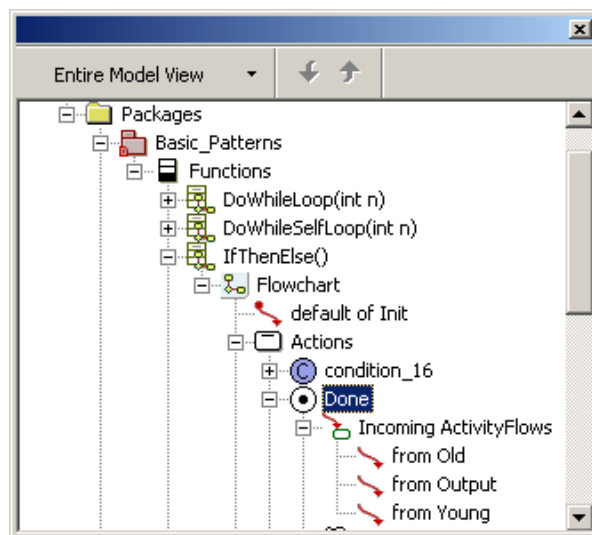
The behavior of the “activity final” is controlled by the `CG::Statechart::LocalTerminationSemantics` property. See the definition provided for the property on the applicable **Properties** tab of the Features window.

Creating an activity final

To create a final activity:

1. Click the ActivityFinal button  on **Diagram Tools**.
2. Click in the flow chart to place the final activity where you want it.
A final activity looks like a circle with a black dot in its center.
3. Draw an activity flow from an action to the final activity.
4. If you want, enter a guard condition to signal the end of the activity.

As with the other connectors, final activities and their flows are included in the Rational Rhapsody browser.



Activity flows

Flow charts can have activity flows (such as activity flows, default flows, and loop activity flows) on actions and action blocks. Activity flows define flow and their guards supply the test for Ifs or Loops. Activity flows without guards define the default sequential flow of the flow chart.

In addition, note the following information:

- ◆ When an “If” flow is detected, then the activity flow with a guard defined on it is the body of the “if.”
- ◆ For all multiple exit activity flows there should always be one without a guard to define the sequential flow.

Activity flows in flow charts are the same as the corresponding transitions in activity diagrams, with the following exceptions:

- ◆ Outgoing activity flows and action blocks cannot have triggers.
- ◆ Outgoing activity flows from actions and action blocks can only have guards.

Drawing activity flows

To draw an activity from one action to another:

1. Click the Activity Flow button  on **Diagram Tools**.
2. Click the edge of the source action.
3. Drag the cursor to the edge of the target action and release to anchor the activity flow.
4. If you want, enter a guard for the activity flow.

Completion action flows

An action flow to a final activity is called a *completion action flow*. Final activities cannot have outgoing action flows. A completion action flow can only have a guard condition.

Drawing initial flows

One of the action elements must be the default action flow. The flow chart flow originates from the element pointed to by the default flow. For flow charts, this default flow is the initial flow, and for code purposes, indicates the start of code.

To draw a default flow:

1. Click the Initial Flow button  on **Diagram Tools**.
2. Click in the flow chart outside the default action.
3. Drag the cursor to the edge of the default action of the activity and release the mouse button.

Drawing loop activity flows

Loop activity flows represent looping behavior in a program. Loop activity flows are often used on action blocks to indicate that the block should loop until some exit condition becomes true. A loop activity flow with a guard is in effect a Do-While statement.

To draw a loop activity flow:

1. Click the Loop Activity Flow button  on **Diagram Tools**.
2. Click the edge of an action.
3. Label the loop activity flow and press **Ctrl+Enter**.

For an example of an action block with a loop activity flow, see [Activity final](#).

Modify action flows

As with all other elements, you can modify the features of an action flow using the Features window. For more information, see [Features of transitions](#).

Connectors

Flow charts can have the following connectors:

- ♦ MergeNode
- ♦ DecisionNode

Drawing merge nodes

A merge node combines different flows to a common target.

To draw a merge node:

1. Click the MergeNode button  on **Diagram Tools**.
2. Click in the flow chart to place the junction where you want it.
3. Draw flows going into, and one flow going out of the junction.
4. Label the flows if you want.

For more information on merge nodes, see [Statecharts](#).

Drawing decision nodes

Decision Nodes show branching conditions. A decision node can have only one incoming activity and two or more outgoing activity flows. The outgoing action flows are labeled with a distinct guard condition. A predefined guard, denoted [else], can be used for no more than one outgoing flow.

A decision node appears as a diamond shape on a flow chart.

To draw a decision node:

1. Click the DecisionNode button  on **Diagram Tools**.
2. Click, or click-and-drag, in the flow chart to position the decision node where you want it.
3. Draw at least two actions that will become targets of the outgoing action flows.
4. Draw an incoming action flow from the source action to the decision node.
5. Draw and label the outgoing action flows from the decision node to the target actions.

This flow chart shows the following behavior: When the phone rings, if someone picks up on the other end, you can talk; otherwise, you must leave a message. The decision node represents the decision point. In other words, after the `PhoneRings()` operation, if `SomeonePicksUp` resolves to `True`, the `Talk()` operation is called. Otherwise, the `LeaveMessage()` operation is called.

Use the Display Options window for the decision node to determine whether to display its name, label, or nothing.

Code generation

You specify whether to generate code for a flow chart by setting the `C_CG::Operation::ImplementFlowchart` property to `Checked` (which is the default).

Flow chart limitations and specified behavior

Note the following restrictions and behavior:

- ◆ You cannot animate or reverse engineer flow charts.
- ◆ Code generation from flow charts is not supported in Rational Rhapsody in Java and Rational Rhapsody in Ada.
- ◆ Flow chart code generation will never write the same action twice.
- ◆ This feature supports only a subclass of diagrams that do not contain events (including timeout events) or triggered operations.
- ◆ Rational Rhapsody makes the following checks:
 - If a function already has a body.
 - On guards that will be ignored because they are not part of an If/Then/Else or Loop.
 - That the flow chart and all its blocks have one and only one reachable final activity. If there are no reachable states or more than one, a message will display.
 - That there are no elements with more than one flow between them in the same direction. If there are more than one, a message will display.
 - That all the elements in the flow chart are supported. If unsupported elements are found, a message will display.
- ◆ If code generated will contain GoTos, Rational Rhapsody will display a warning message with an indication as to which flows are causing the warning. Note the following information:
 - Flow charts will normally generate structured code using If, If/Then/Else, Do, and While blocks. Rational Rhapsody in C provides you with a Flowchart model located in the `<Rational Rhapsody installation>\Samples\CSamples\Flowchart` folder. The Flowchart model contains a number of sample flow charts patterns that show you which ones are recognized in order to generate structured code. For example, the Flowchart model includes flow charts that show the DoWhileLoop, IfThenElse, and the WhileLoop.

- If the code is not structured, then the flow charts will generate GoTo code. To avoid GoTo code, use the sample patterns for structured blocks as shown in the flow charts that are illustrated and documented in the Flowchart model provided with Rational Rhapsody in C, which is located in the path noted above.

Sequence diagrams

Sequence diagrams (SDs) describe message exchanges within your project. You can place messages in a sequence diagram as part of developing the software system. You can also run an animated sequence diagram to watch messages as they occur in an executing program.

Sequence diagram show scenarios of messages exchanges between roles played by objects. This functionality can be used in numerous ways, including analysis and design scenarios, execution traces, expected behavior in test cases, and so on.

Sequence diagrams help you understand the interactions and relationships between objects by displaying the messages that they send to each other over time. In addition, they are the key tool for viewing animated execution. When you run an animated program, its system dynamics are shown as interactions between objects and the relative timing of events.

Sequence diagrams are the most common type of interaction diagrams.

Note

Rational Rhapsody message diagrams are based on sequence diagrams. Message diagrams, available in the FunctionalC profile, show how the files functionality might interact through messaging (through synchronous function calls or asynchronous communication). Message diagrams can be used at different levels of abstraction. At higher levels of abstractions, message diagrams show the interactions between actors, use cases, and objects. At lower levels of abstraction and for implementation, message diagrams show the communication between classes and objects.

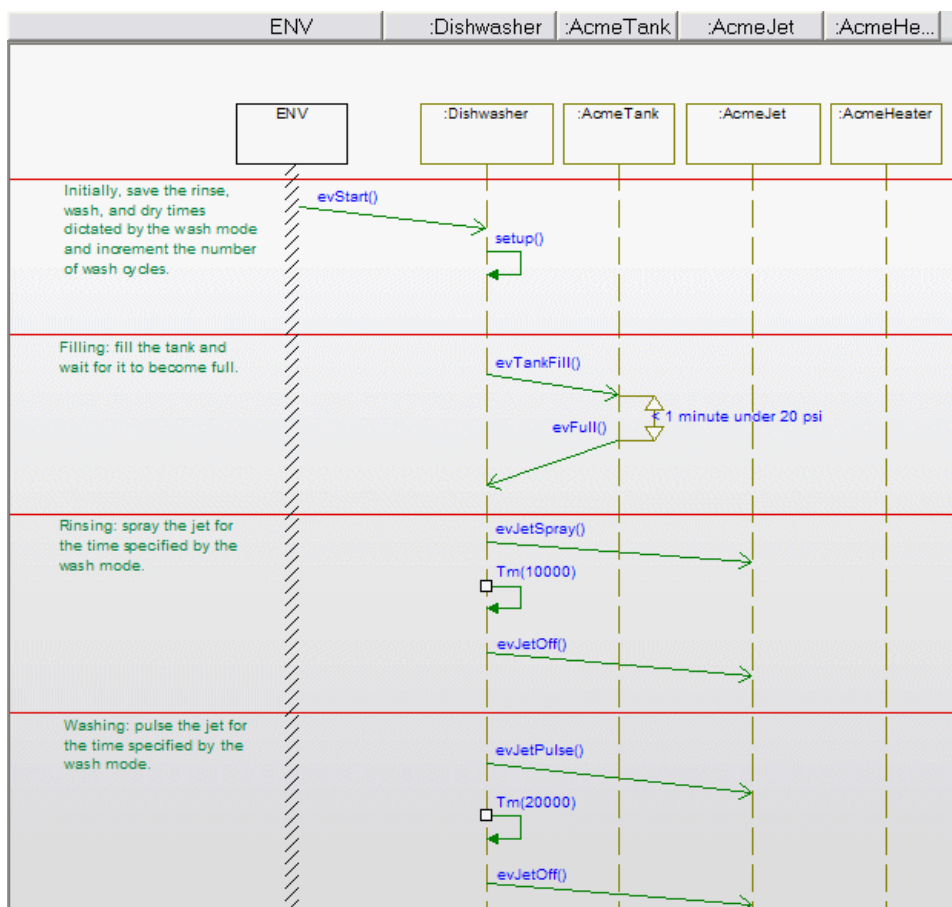
Message diagrams have an executable aspect and are a key animation tool. When you animate a model, Rational Rhapsody dynamically builds message diagrams that record the object-to-object messaging.

For more information about the FunctionalC profile, see [Profiles](#).

Sequence diagram layout

A sequence diagram has two sections:

- ♦ **Names pane**, which is the top portion of the diagram. The Names pane is a control to identify instance lines when the role names are not visible.
- ♦ **Message pane** shows the messages passed between instance lines in the diagram.



Names pane

The names pane contains the name of each instance line or *classifier role*. In a sequence diagram, a *classifier role* represents an instance of a classifier. It describes a specific role played by the classifier instance to perform a particular task. A classifier role is shown as an instance line with a text header (name) with a box drawn around it. A classifier role can realize a classifier (class or actor) of the static model object.

Changing names

Names that are too long to fit in the pane continue past the divider, running down behind the lower pane. To change the size of the names pane, click the dividing line and drag it up or down.

You can change the font and edit the names in the names pane using the menu for text items.

There are three ways to describe the name:

```
Classifier Role Name: Classifier Name
                    : Classifier Name
Classifier Role Name
```

In the first two cases, if the classifier name does not exist in the metamodel, Rational Rhapsody asks if you want to add a new classifier to the project. The third case tells Rational Rhapsody that you want to use an <Unspecified> classifier role, which means that the classifier role is not a realization of an existing classifier or actor.

Renaming classifier roles

If you change the name of a classifier role to a role name that already exists in the model, the classifier role is automatically realized to that classifier. For example, if you change the role name of classifier B to “Alarm” and there is a class Alarm in the model, this role becomes a realization of class Alarm and its name changes to B: :Alarm.

If you change the name to a class that does not yet exist in the model, Rational Rhapsody asks if you want to create that class. For example, if you type x:X, Rational Rhapsody asks if you want to create the class X.

Message pane

The message pane contains the elements that make up the interaction. In the object pane, system borders and instances are displayed as instance line, which are vertical lines with a box containing the role name at the top. Messages, such as events, operations, and timeouts are generally shown as horizontal and slanted arrows.

The messages appear in sequence as time advances down the diagram. The vertical distance between points in time indicates only the sequence in time and not any time scale.

Analysis versus design mode

Three properties (under `SequenceDiagram::General`) to support the SD operation modes:

- ♦ `ClassCentricMode` specifies whether classes are realized when you draw instance lines. The possible values are as follows:
 - `Checked` means instance names of the form `<xxx>` are treated as class names, not instance names. For example, if you create a new instance line named `c`, Rational Rhapsody creates a class named `c` and displays it in the sequence diagram as `:c`.
 - `Cleared` means when you create an instance line, it is named `role_n` by default, which represents an anonymous instance. This is the default value.
- ♦ `RealizeMessages` specifies whether messages are realized when you create them. The possible values are as follows:
 - `Checked` where in Design mode, when you type in a message name, Rational Rhapsody asks if you want to realize the message. If you answer no, the message is unspecified. For example, you could use an unrealized message to describe a message that is part of the framework (such as `takeEvent()`), without actually adding it to the model. (In analysis mode, the confirmation is controlled by the property `SequenceDiagram::General::ConfirmCreation`.)
 - `Cleared` where you can draw message lines freely, without messages from Rational Rhapsody about realization. This is the default value.
- ♦ `CleanupRealized` specifies whether to delete messages in the sequence diagram if the corresponding operation is deleted. The possible values are as follows:
 - `Checked` means to delete the messages when the operation is deleted.
 - `Cleared` means to not delete the messages when the operation is deleted. This is the default value.

For sequence diagrams produced in Rational Rhapsody 4.0 or earlier, all three of these properties are `Cleared`.

Showing unrealized messages

To show a message that has not been realized, select **Edit > Select > Select Un-Realized**. The unrealized message is selected in the sequence diagram.

Realizing a selected element

To realize a selected element:

1. Select the element in the sequence diagram.
2. Select **Edit > Auto Realize**.

When you realize a message, Rational Rhapsody creates a new message in just the manner as if you selected <New> in the **Realization** field in the Features window. If you realize a classifier role, Rational Rhapsody creates a class with the same name as designated in the role name, with a leading colon. For example, Dishwasher becomes :Dishwasher.

Creating sequence diagram elements

The following sections describe how to use the sequence diagram tools to draw the parts of a sequence diagram. For basic information on sequence diagrams, including how to create, open, and delete them, see [Graphic editors](#).

Sequence diagram drawing tools

The **Diagram Tools** for a sequence diagram includes the following tools.

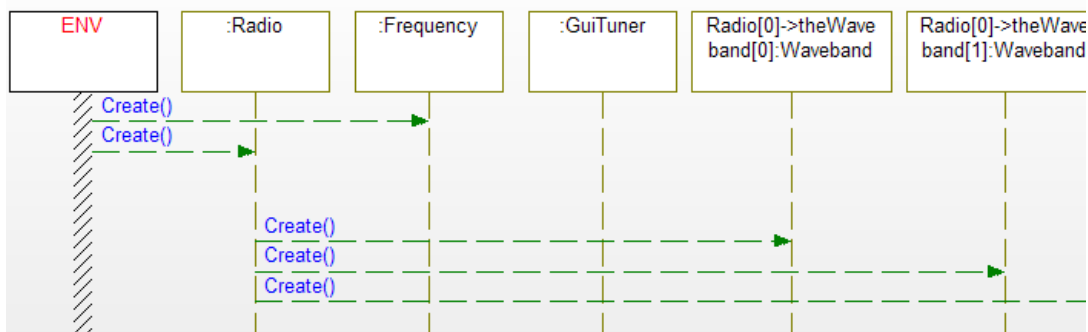
Drawing Tool	Name	Description
	Instance line	Shows how an actors participates in the scenario. For more information, see Creating an instance line .
	System border	Represents the environment. Events or operations that do not come from instance lines shown in the chart are drawn coming from the system border. For more information, see Creating a system border .
	Message	Represents an interaction between parts, or between a part and the environment. A message can be an event, a triggered operation, or a primitive operation. For more information, see Creating a message .
	Reply message	Represents the response from a message sent to a part or the environment. For more information, see Creating a reply message .
	Create arrow	Marks when an instance is created. It can originate from the system border or another instance. It is a horizontal, dotted line from the creator object to the new object. An object can have only one "create arrow." You can label the create arrow with construction parameters.
	Destroy arrow	Marks the destruction of an object. It is a dotted line from the destroying object to the object being destroyed. It can be either a horizontal line or a message-to-self. For more information, see Creating a destroy arrow .
	Timeout	Indicates when an event stops and might include a parameter indicating the length of the time the event is stopped. This is a type of message that is always communicating with itself. For more information, see Creating a timeout .
	Cancelled timeout	indicates the condition when an event that has timed out should restart. For more information, see Creating a cancelled timeout .
	Time interval	Can be used to create a waiting state with an event stopping for this predefined interval and then automatically restarting. For more information, see Specifying a time interval .
	DataFlow	Indicates the flow of data between two objects. You can use the Features window to select the flowport to which it is connected on the receiving object and change the value being sent. This connection is also automatically added to the sequence diagram during animation. For more information, see Creating a dataflow .
	Partition line	Separates phases of a scenario represented in the sequence diagram. For more information, see Creating a partition line .

Drawing Tool	Name	Description
	Condition mark	Indicates that the object is in a certain condition or state at this point in the sequence. For more information, see Creating a condition mark .
	Execution Occurrence	Shows the beginning and end of the unit of behavior (the actions performed by an operation or event) that is triggered by a specific message. For more information, see Creating execution occurrences .
	Interaction Occurrence	Refers to another sequence from within a sequence diagram. This allows complex scenarios to be divided into smaller, reusable scenarios. For more information, see Creating an interaction occurrence .
	Interaction Operator	Groups related elements and define specific conditions under which each group of elements occurs. For more information, see Creating interaction operators .
	Interaction Operand Separator	Create two subgroups of elements within the sequence diagram. This might be used to create two paths that are supposed to be carried out in parallel or to define two possible paths and a condition that determines which is to be followed. For more information, see Adding an interaction operand separator to an interaction operator .
	Lost Message	Indicates a message sent from an instance that never arrives to its destination. This item is not supported in code generation or animation.
	Found Message	Indicates a message that arrives at an instance, but its target is unknown. This item is not supported in code generation or animation.
	Destruction Event	Indicates the destruction of the instance, such as the destroy arrow, has happened. This item is not supported in code generation or animation.

Creating a system border

A *system border* represents the environment. Events or operations that do not come from instance lines shown in the chart are drawn coming from the system border.

A system border is a column of diagonal lines, labeled ENV. You can place a system border anywhere an instance line can be placed, but the typical locations are the far left and right edges of the chart.



To create a system border:

1. Click the System border button .
2. Move the cursor into the drawing area, then click to place the system border. At this point, the system border is anchored in place.
3. Because the system border represents the environment, it is named ENV by default. If wanted, rename the system border.

Creating an instance line

An instance line (or *classifier role*) is a vertical timeline labeled with the name of an instance. It represents a typical instance or class in the scenario being described. It can receive messages from or send messages to other instance lines.

In addition to creating, deleting, and modifying the name of an instance line, you can realize the instance line to a class or actor in the static model.

1. Click the Instance line button .
2. Move the cursor over the diagram.
3. Move the line to a suitable location, then click to dock the line into place.
4. Type the name of a class or an instance to replace the default name.
5. If you specified design mode, Rational Rhapsody names the instance line `:class_n` by default. If the specified class does not exist, Rational Rhapsody asks if you want to create it. Click **OK**.
6. You can continue creating instance lines, or return to select mode by clicking the **Select** tool.

If you prefer, you can place several lines and rename them later. Rational Rhapsody gives them default names until you rename them. For information on renaming instance lines, see [Message line menu](#). Note that the sequence diagram automatically expands the diagram as necessary as you add more instance lines.

Note

To shift one or more messages to different instance lines, select the relevant messages and press **Ctrl+Right arrow**, or **Ctrl+Left arrow**. The messages “jump” to the new source and destination instance lines. This replaces the cut and paste (or drag) functionality of messages between instance lines in the same diagram.

Modifying the features of a classifier role

The Features window enables you to change the features of a classifier role, including its realization. A classifier role has the following features:

- ♦ **Name** specifies the name of the classifier role.
- ♦ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ **Stereotype** specifies the stereotype of the element, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).
Note: The COM stereotypes are constructive; that is, they affect code generation.
- ♦ **Realization** specifies the class being realized by the instance line.
- ♦ **Decomposed** specifies the referenced sequence diagram for the instance line, if you are using part decomposition. For more information, see [Part decomposition](#).
- ♦ **Description** describes the classifier role. This field can include a hyperlink. For more information, see [Hyperlinks](#).

Names of classifier roles

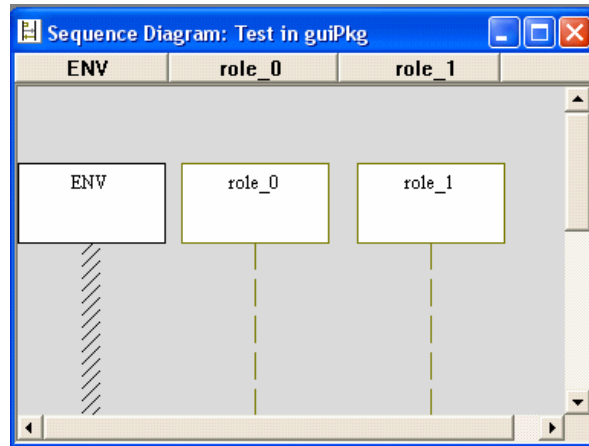
A classifier role (instance line) with a class name is a view of the class in the model. An instance line with an instance name (of the form `instance: class`) is also a view of the class in the model.

Instance lines reference classes in the model. If you rename an instance line to another class name that exists in the model, the line acts as a view to the other class in the model. If the class does not exist, Rational Rhapsody will create it.

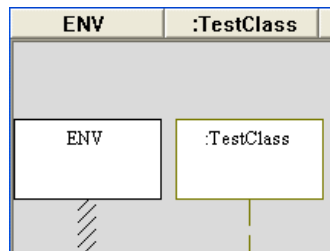
Note

The names for instance lines are resizable text frames. Text wraps to match the contour of the bounding box.

If you specified analysis mode, Rational Rhapsody names the instance line `role_n` by default and does not prompt you for class information. The following figure shows a new instance line in an analysis SD.



If you specified design mode, Rational Rhapsody names the instance line `:class_n` by default. If the specified class does not exist, Rational Rhapsody asks if you want to create it. The following figure shows a new instance line in a design SD.



Note that Classifier role names are animated when their expression can be mapped to an existing object.

Examples:

We have a class A and an object `a:A` who is the only object of A. All the following names will be mapped to it:

- ◆ `:A`
- ◆ `a:A`
- ◆ `A[0]:A`
- ◆ `A[#0]:A`

Suppose that instead of `a:A`, we have a single instance of A as a part `itsA` of another object `b:B`.

The instance line can be named as:

- ◆ :A
- ◆ b->itsA:A
- ◆ B[0]->itsA:A
- ◆ A[#0]:A
- ◆ B[#0]->itsA->a:A

The same instance mappings apply as described in [Instance Names](#).

Instance line menu

- ◆ **Class** displays a submenu of commands for classes.
- ◆ **Open Reference Sequence Diagram** opens the reference sequence diagram associated with the classifier role. This option is unavailable if a reference sequence diagram does not exist for this classifier role. For more information, see [Creating an interaction occurrence](#).
- ◆ **Display Options** specifies how the element should be displayed.

Creating a message

A *message* represents an interaction between objects, or between an object and the environment. A message can be an event, a triggered operation, or a primitive operation. In the metamodel, a message defines a specific kind of communication. The communication could be raising a signal, invoking an operation, or creating or destroying an instance.

The recipient of a message is either a class or a reactive class. Reactive classes have statecharts, whereas nonreactive classes do not. Reactive classes can receive events, triggered operations, and primitive operations. Non-reactive classes can receive only messages that are calls to primitive operations. Events are usually shown with slanted arrows to imply that they are *asynchronous* (delivery takes time). Triggered operations are shown with straight arrows to imply that they are *synchronous* (happen immediately).

To create a message:

1. Click the Message (event) button .
2. Move the cursor over the instance lines.
Note: A plus sign displays on each instance line as you move the cursor from one to the next. This symbol indicates a potential origination point for the wanted message.
3. Left-click to anchor the start of the message at the intended location, then move the cursor. A dashed line displays as a guide for the message.
4. Move the cursor lower the start of the message to create a downward-slanted diagonal line. Click to anchor the end of the message on the target object once the diagonal line has extended itself to that point.
5. If you specified design mode and the specified message is not realized in the model, Rational Rhapsody asks if you want to realize it. Click **OK**.

Rational Rhapsody creates a message with the default name `message_n()`, where *n* is an incremental integer starting with 0. Sequence diagrams automatically expand in length to accommodate new messages.

To specify the type of operation and its access level, select the **Features** option from the menu. By default, Rational Rhapsody creates a primitive operation with public access. For more information, see [The Features window](#).

Message names

The naming convention for messages is as follows:

```
message(arguments)
```

The names for messages can be event or operation names. They can include actual parameters in parentheses, which would be expressions in the scope of the sender/caller. Message names are resizable, movable text frames. Text wraps to match the contour of the bounding box.

Note: In Rational Rhapsody versions 4.0 and earlier, if you changed the message name, Rational Rhapsody asked if you wanted to create a new operation with the given name.

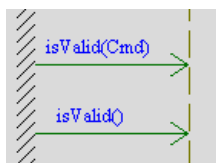
If you modify the name of an operation that exists in the model, the message is automatically realized to that operation, and Rational Rhapsody changes its type to the type of that operation (constructor, event, and so on).

If you change the message name to a message that does not belong to the classifier, it becomes unspecified (in analysis mode).

Displaying message arguments

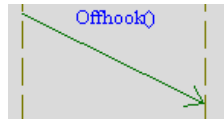
The `SequenceDiagram::General::ShowArguments` property displays or hides message arguments. By default, the `ShowArguments` property is activated. It applies to all messages in a sequence diagram, not to individual messages.

Note that any changes you make to property settings apply only to new elements you draw after making the change, not to existing elements. For example, to have arguments displayed on one message but not on another message of the same type, set the `ShowArguments` property before drawing one message, then reset it before drawing the next message, as shown in this example.



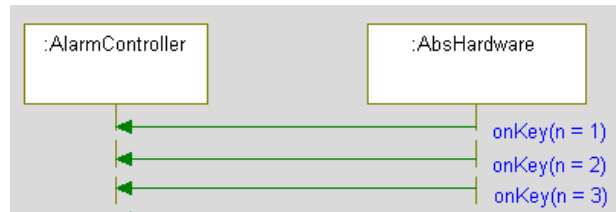
Slanted messages

A message drawn on a slant is interpreted as an event if the target is a reactive class, and as a primitive operation if the target is a nonreactive class. A slanted message emphasizes that time passes between the sending and receiving of the message. Slanted messages can cross each other.



Horizontal messages

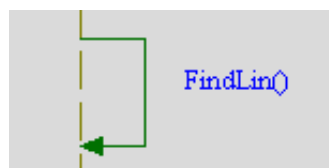
A message drawn as a horizontal line is interpreted as a triggered operation, if the target is a reactive class, and a primitive operation if the target is a nonreactive class. The horizontal line indicates that operations are synchronous.



Message-to-self

A message-to-self is interpreted as an event if the instance is a reactive class. If the instance is a nonreactive class, a message-to-self is interpreted as a primitive operation.

A message-to-self is shown by an arrow that bends back to the sending instance. The arrow can be on either side of the instance line. If the message-to-self is a primitive operation, the arrow folds back immediately. If the message-to-self is an event, the arrow might fold back sometime later.



Message line menu

- ♦ **Select Message** enables you to select a message. For more information, see [Selecting a message or trigger](#).
- ♦ **Add Execution Occurrences** enables you to add execution occurrences.
- ♦ **Display Options** specifies how the element should be displayed.

Modifying the features of a message

The Features window enables you to change the features of a message, including its type or return value. A message has the following features:

- ♦ **Name** specifies the name of the message. The default name is Message_n, where *n* is an incremental integer starting with 0.
- ♦ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ **Stereotype** specifies the stereotype of the message, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).

Note: The COM stereotypes are constructive; that is, they affect code generation.

- ♦ **Message Type** specifies the type of message (event, primitive operation, and so on).

Note that you cannot change the message type once the message has been realized.

- ♦ **Sequence** specifies the order number of a message. Make sure any numbering sequence you use, such as 1a., 1b., 1.2.1., 1.2.2., and so on ends with a period (.). Rational Rhapsody needs the ending period to continue the numbering sequence automatically.

Note: In collaboration diagrams, you can modify both the format of the numbering and the starting point for the numbering. In sequence diagrams, this field is read-only. You cannot modify the numbering format or the starting point of the numbering.

- ♦ **Arguments** specifies the arguments for the message.
- ♦ **Return Value** specifies the return value of the message.
- ♦ **Realization** specifies the class that will be realized. This list contains the following options:
 - Existing classes in project.
 - <Unspecified> where if this is an analysis SD, this is the realization setting for the instance line.
 - <New> which opens the Class window so you can set up the new class. For more information, see [Class features](#).

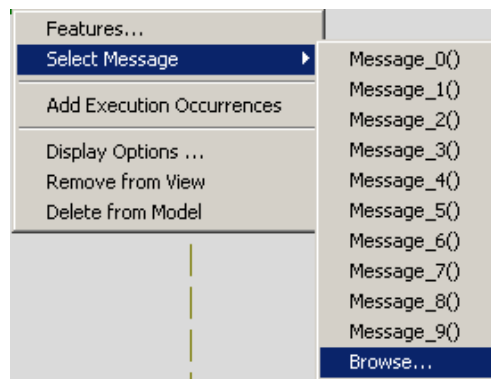
Click the **Features** button to open the Features window for the class specified in the list.

In addition, this section lists the sender and receiver objects for the message.

- ♦ **Description** describes the message. This field can include a hyperlink. For more information, see [Hyperlinks](#).

Selecting a message or trigger

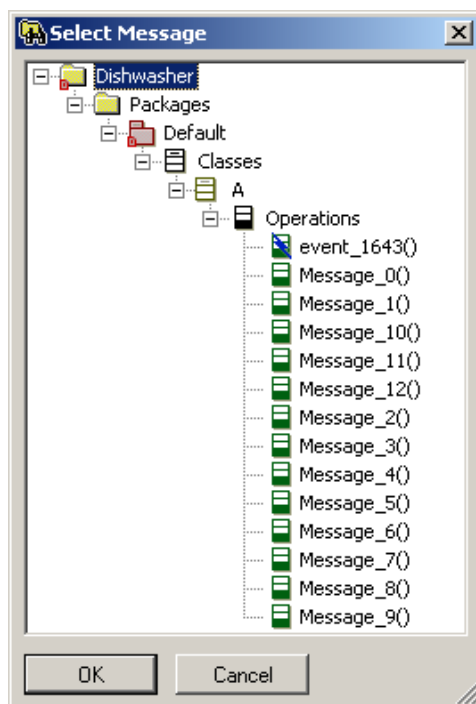
When you choose **Select Message** or **Select Trigger**, Rational Rhapsody displays a list of the messages or triggers provided by the target object, as shown in the following figure for messages. Notice that if there are more messages (or triggers) that can appear on the pop-up menu, a **Browse** command displays. For triggers, see also [Selecting a trigger transition](#).



For messages, if the target is an instance of a derived class, the list also includes those message inherited from its superclass. The target class provides the message, whereas the source class requires it. You can also think of provided messages as those to which the class responds. At the implementation level, an event is generated into statechart code that checks for the existence of that event.

Browsing for messages

The following figure shows the Select Message window that opens when you choose **Select Message > Browse** for messages. It shows you all the messages/events that are available.



Cutting, copying, and pasting messages

You can cut, copy, and paste messages in sequence diagrams using the standard **Ctrl+X**, **Ctrl+C**, and **Ctrl+V** keyboard shortcuts, respectively.

Moving messages

To move a message line:

1. Select the message you want to move.
2. Press **Ctrl+Left arrow** to move the message to the left, or **Ctrl+Right arrow** to move it to the right.

Message types

If you open the Features window for the message, you can select the message type: primitive operation, or triggered operation, or event. Once defined, these messages are displayed in the browser, denoted by unique icons. In the browser, you can access modify a message by right-clicking on it and selecting the appropriate option from the menu.

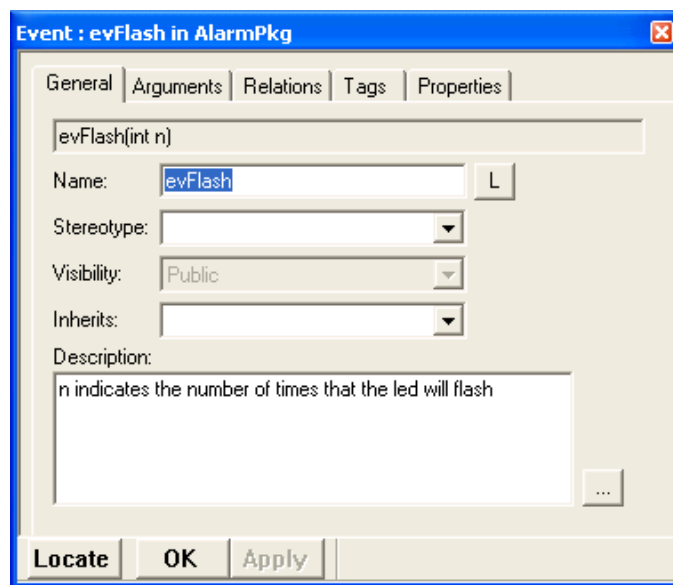
Note

Once a message has been realized, you cannot change its type.

Events

An *event* is an instantaneous occurrence that can trigger a state transition in the class that receives it. Because events are objects, they can carry information about the occurrence, such as the time at which it happened. The browser icon for an event is a lightning bolt.

The following figure shows the Features window for an event.



Note

If an event argument is of type *& (pointer reference), Rational Rhapsody does not generate code for it.

Triggered operations

A *triggered operation* can trigger a state transition in a statechart, just like an event. The body of the triggered operation is executed as a result of the transition being taken. The browser icon for a triggered operation is a green operation box overlaid with a lightning bolt.

Note

If an argument of a triggered operation is of type *& (pointer reference), Rational Rhapsody will not generate code for that argument.

Operations

By default, operations are primitive operations. *Primitive operations* are those whose bodies you define yourself instead of letting Rational Rhapsody generate them for you from a statechart.

The browser icon for operations is a three-compartment class box with the bottom compartment filled in:



The icon for the Operations category is black.



The icon for an individual operation is green.



The icon for a protected operation is overlaid with a key.



The icon for a private operation is overlaid with a padlock.

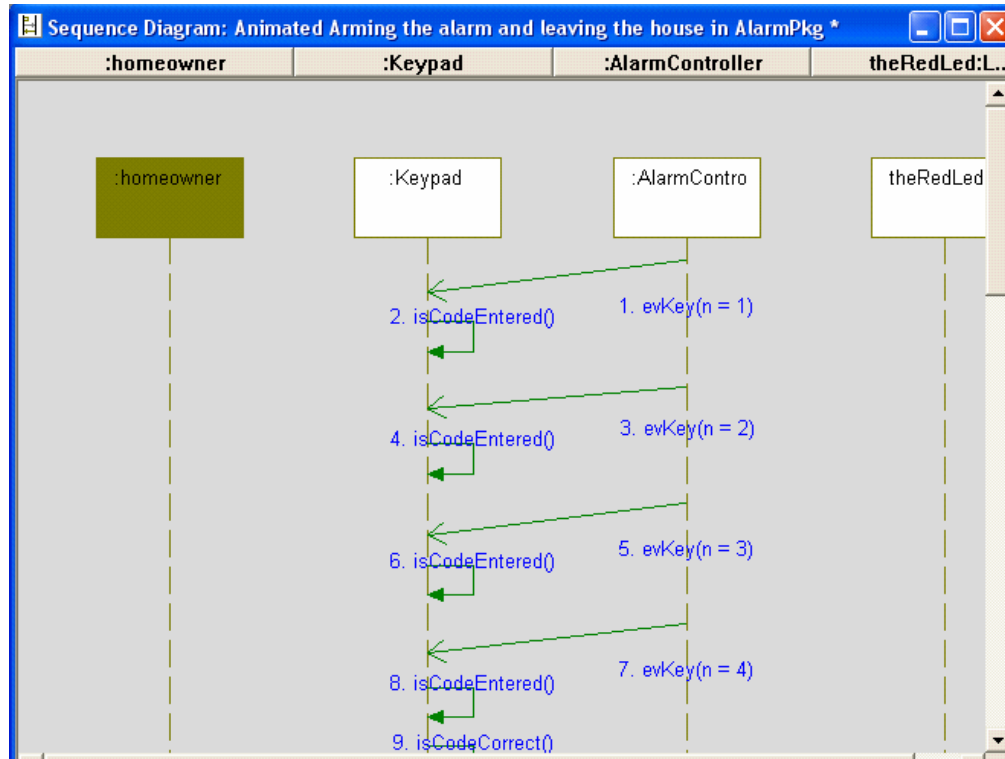
Deleting operations

You delete an operation like any other model element in Rational Rhapsody. However, if you delete an operation that is realized by a message in a sequence diagram, the message becomes unspecified (in both design and analysis modes).

If you delete an operation, class, or event, the corresponding message lines are not deleted automatically from the diagram. To have Rational Rhapsody perform this cleanup automatically, set the `SequenceDiagram::General::CleanupRealized` property to `Checked`.

Viewing sequence numbers

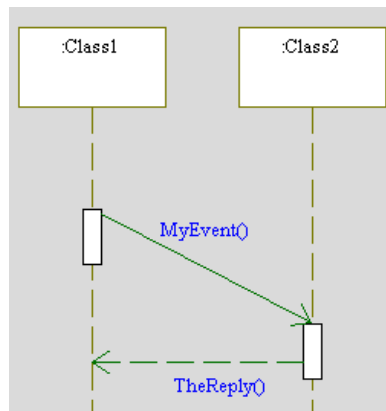
If wanted, you can display the sequence number with the message name. The following figure shows a sequence diagram that includes the sequence number of each operation.



To display the sequence numbers, set the `SequenceDiagram::General::ShowSequenceNumber` property to `Checked`. By default, this property is set to `Cleared` (sequence numbers are not displayed).

Creating a reply message

A *reply message* can realize an operation or event reception at the *source* (unlike other messages, which realize operations at the *target*). By default, reply messages are drawn as unnamed, dashed lines



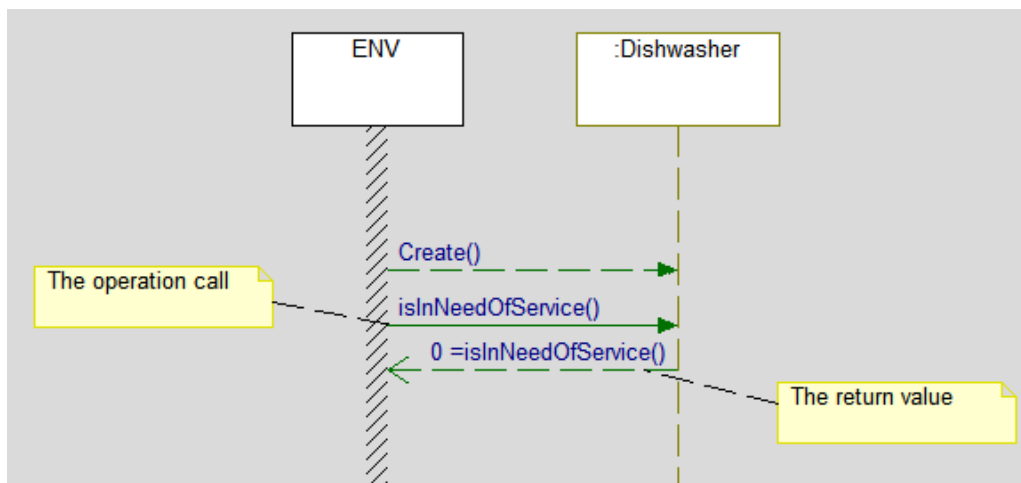
Animation of the return value for an operation

Note

This feature is not supported in Rational Rhapsody in J.

To show the return value of a function as a reply message in an animated sequence diagram, you can use one of a number of predefined macros within the code of your function. This means that the return value for your function visually displays as a reply message on your sequence diagram. The same is true for a trace of a function.

The following figure shows an animated sequence diagram that draws a return value.



You can use any of the following macros depending on your situation:

- ◆ OM_RETURN. Use this macro in the body of an operation instead of the regular “return” statement:

Examples:

- `Int Test(int& x) {x = 5; OM_RETURN(10);}`
- `A* Test() {OM_RETURN(newA());}`

- ◆ **CALL**. Use this macro if you cannot change the operation code or if you want to animate return values only on specific calls to the operation. Note that this macro can handle only primitive types.

Example:

```
Int test(int n) {return n*5;}
void callingFunction()
{
    int v;
    CALL (v, foo(10));
    // after the call v equals 50
}
```

- ◆ **CALL_INST**. Same as **CALL**, but use **CALL_INST** when the return value is of a complex type, such as a class or a union.

Example:

```
A* test() {return new A();}
void callingFunction()
{
    A *a;
    CALL_INST(a, test());
    // after the call a equals new A[0]
}
```

- ◆ **CALL_SER**. Use this macro when the type has a user-defined serialization function.

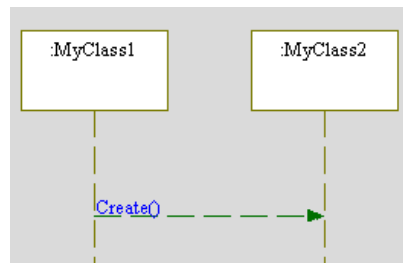
Examples:

```
- char* serializeMe(A*) {...}
- A* test() {return new A();}
void callingFunction()
{
    A *a;
    CALL_SER(a, test(), serializeME);}
    // after the call v equals <string that serializeMe returns>
}
```

Note that even if you choose not to embed these macros in your application, you can still see animated return values by explicitly calling an operation through the Operations window. To call an operation, click the **Call operations** tool in the **Animation** toolbar.

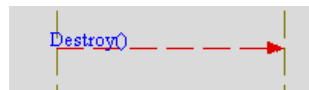
Drawing an arrow

An *arrow* marks when an instance is created. It can originate from the system border or another instance. It is a horizontal, dotted line from the creator object to the new object. Every object can have at most one create arrow. You can label the create arrow with construction parameters.



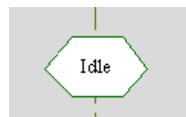
Creating a destroy arrow

A *destroy arrow* marks the destruction of an object. It is a dotted line from the destroying object to the object being destroyed. The destroy arrow is red and is not labeled. It can be either a horizontal line or a message-to-self.



Creating a condition mark

A *condition mark* (or state mark) is displayed on an instance line. A condition mark shows that the object has reached a certain condition or is in a certain state. Often, the name corresponds to a state name in the statechart for an object.



1. Select the condition mark using the Select arrow.
2. Click-and-drag a selection handle to resize the condition mark.

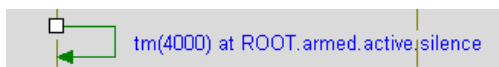
The condition mark remains centered over the instance line. If necessary, other elements on the sequence diagram are adjusted to accommodate the new size.

Creating a timeout

The notation for timeouts is similar to the notation for events sent by an object to itself. There are two differences:

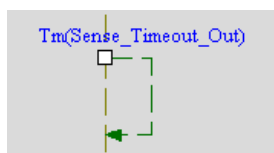
- ♦ A timeout starts with a small box.
- ♦ The name is a `tm(x)`.

The label on a timeout arrow is a parameter specifying the length of the timeout. Timeouts are always messages-to-self.



Creating a cancelled timeout

When designing a software system, you can establish waiting states, during which your program waits for something to occur. If the event occurs, the timeout is canceled. The sequence diagram shows this with a canceled timeout symbol. If it does not happen, the timeout wakes up the instance and resumes with some sort of error recovery process. Canceled timeouts are always messages-to-self.

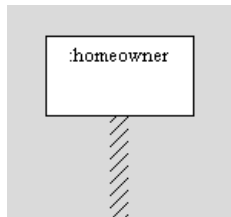


For example, on a telephone, a dial tone waits for you to dial. The telephone has a timeout set so if you do not dial, the dial tone changes to a repeating beep. If you do dial, the timeout is canceled.

A canceled timeout occurs automatically once the state on which the timeout is defined is exited. As a designer, you do not need to do anything to cancel a timeout. The framework has a call to cancel a timeout, but you do not need to use it because the code generator inserts it automatically.

Creating an actor line

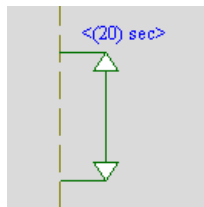
An *actor line* shows where actors affect the sequence diagram. To draw an actor line, drag-and-drop an actor from the browser to the sequence diagram. An actor is represented as an instance line with hatching.



When you want to realize a classifier, the list contains all the available classes and actors.

Specifying a time interval

A time interval is a vertical annotation that shows how much (real) time has passed between two points in the scenario. The name is free text; it is not constrained to be a number or unit of any kind. Time intervals can only be messages to self.

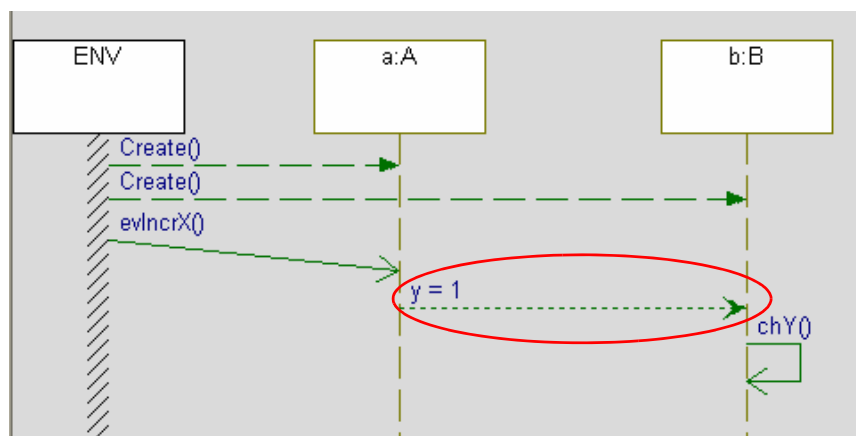


Creating a dataflow

A dataflow in Rational Rhapsody indicates the flow of data between two instances on a sequence diagram, as shown in the following figure. Rational Rhapsody animation uses this notation to represent data flow between flow ports. For information on flow ports, see [Flow ports](#).

Note

This feature is not supported in Rational Rhapsody in J.

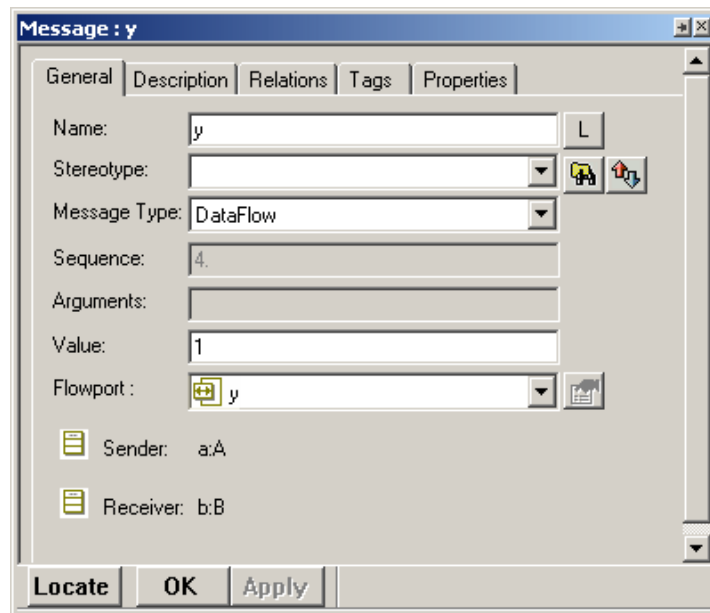


The dataflow will be realized to the flow port on the receiving instance. The name of the dataflow is the name of its realized flow port and the data that has been received. For example, for the dataflow `y = 1`, flow port `y` has received the data `1`.

Note that the dataflow arrow can be created through the following ways:

- ♦ Automatically during the animation of the sequence diagram
- ♦ Manually by drawing the dataflow on the sequence diagram (that is, click the DataFlow button  and then click the sender and receiver instances).

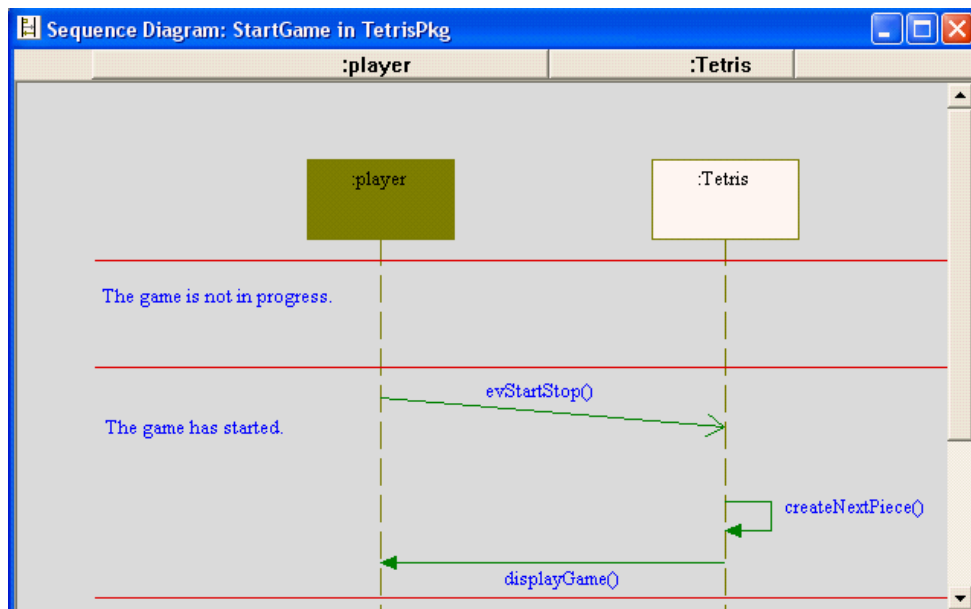
You can double-click the dataflow arrow to open its Features window, as shown in the following figure, from which you can, for example, choose which flow port to connect to and change the value to send.



Creating a partition line

Partition lines separate phases of a scenario. They are red lines drawn across the chart and are usually used to keep parts of the scenario grouped together.

Each partition line includes a note positioned at its left end by default. You can move or resize the note, but a note is always attached to its partition line. If you move the line, the note follows. Notes contain the default text “note” until you enter text for them.



Creating an interaction occurrence

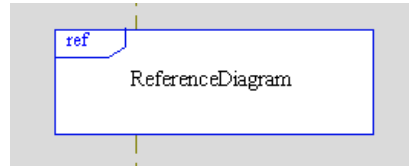
An *interaction occurrence* (or reference sequence diagram) enables you to refer to another sequence diagram from within a sequence diagram. This functionality enables you to break down complex scenarios into smaller scenarios that can be reused. Each scenario is an “interaction.”

To create an interaction occurrence:

1. Click the Interaction Occurrence button.

Alternatively, you can use the **Add Interaction Occurrence** option in the menu.

2. Place the reference diagram on one or more instance lines to signify that those classes interact with the referenced sequence diagram. The interaction occurrence displays as a box with the “ref” label in the top corner, as shown in this example.



By default, when you first create the interaction occurrence (and have not yet specified which diagram it refers to), Rational Rhapsody names it using the convention `interaction_n`, where *n* is greater than or equal to 0.

3. Right-click the interaction occurrence and then select **Features**.
4. Use the **Realization** list to specify the sequence diagram being referenced. When you select the referenced diagram, the name of the interaction occurrence is updated automatically to reflect the name of the referenced SD.
5. Click **OK**.

You can move, rename, and delete reference sequence diagrams just like regular sequence diagrams. However, if you delete a sequence diagram that references an interaction occurrence, the interaction occurrence itself is not deleted, but becomes unassociated.

To change the default appearance of interaction occurrences, use the `SequenceDiagram::InteractionOccurrence` properties. See the definition displayed in the **Properties** tab for this property.

Navigating to a reference sequence diagram

To navigate to a reference sequence diagram from the current sequence diagram, right-click the interaction occurrence and select **Open Reference Sequence Diagram**. The referenced SD is displayed in the drawing area.

Interaction occurrence menu

- ♦ **Create Reference Sequence Diagram** enables you to specify the reference diagram for the interaction occurrence, if you have not yet specified one
- ♦ **Open Reference Sequence Diagram** opens the sequence diagram referred to by the interaction occurrence
- ♦ **Display Options** specifies whether labels are displayed

Part decomposition

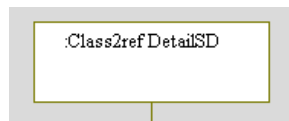
Instance line decomposition enables you to easily decompose a instance line on a sequence diagram into a series of parts. For example, if you have a composite class view in one sequence diagram and want to navigate to its parts, you can click the composite class and open a collaboration, which shows how its internal parts communicate for a particular scenario.

Part decomposition is a specialization of an interaction occurrence.

To create a part decomposition:

1. Open the Features window for the instance line. The Classifier Role window opens.
2. Specify the reference sequence diagram using the **Decomposed** list.
3. Click **OK**.

In the sequence diagram, the name of the reference sequence diagram is added to the classifier role label (after the word “ref”), as shown in this example.



As with other interaction occurrences, navigate to the reference SD by right-clicking the instance line and selecting **Open Reference Sequence Diagram**.

Limitations

Note the following limitations for decomposition:

- ♦ UML gates are not supported: use instance lines instead.
- ♦ Animation is not supported.

Creating interaction operators

Interaction operators, which are included in UML 2.0, are used to group related elements in a sequence diagram. This includes the option of defining specific conditions under which each group of elements will occur.

Characteristics of interaction operators

Each interaction operator has a type, which determines its behavior, for example Opt, Par, or Loop.

In addition, interaction operators can include a guard to specify specific conditions under which the path will be taken.

Interaction operators can be nested where necessary.

Adding an interaction operator to a diagram

To add an interaction operator to a diagram:

1. Click the Interaction Operator button on the **Diagram Tools**.
2. Click the canvas for a default-size interaction operator, or click and drag to draw an interaction operator of a specific size.

Setting the type of an interaction operator

To set the type of an interaction operator, you can do any of the following actions:

- ♦ With the interaction operator selected in the diagram, click the type text in the top left corner and enter the appropriate type.
- ♦ With the interaction operator selected in the diagram, click the type text and then press **Ctrl+Space** and select a type from the list that is displayed.
- ♦ Open the Features window for the interaction operator, and select a type from the Type list.

Setting the guard of an interaction operator

To set the guard for an interaction operator, do one of the following actions:

- ♦ Click **[condition]** and enter the appropriate expression inside the brackets
- ♦ Open the Features window for the interaction operator, and type in the expression under Constraints.

Note

Using **Display Options** from the context menu, the guards for an interaction operator can be hidden or shown.

Adding an interaction operand separator to an interaction operator

For certain types of interaction types, you might want to create two subgroups of elements, for example, if you have two paths that are supposed to be carried out in parallel, or you want to define two possible paths and a condition that determines which will be followed.

To create an interaction operand separator:

1. Select the Interaction Operand Separator button on the **Diagram Tools**.
2. Click inside the interaction operator where you would like the division to be.

If necessary, more than one separator can be added to a single interaction operator.

Note

Interaction operators (and interaction operand separators) only appear in the diagram itself. They do not appear as independent model elements in the browser, nor do they influence code generation. For this reason, when you display the context menu for an interaction operator, there is an option to **Delete From View** but no option for removing from the model.

Interaction operator types

These are some of the common types of interaction operators:

- ♦ Alt for (Alternative) multiple fragments; only the one whose condition is true will execute
- ♦ Opt for (Optional) fragment executes only if the specified condition is true
- ♦ Par for (Parallel) each of the contained fragments is run in parallel
- ♦ Loop for the fragment might execute multiple times; guard indicates the basis for iteration

Creating execution occurrences

Execution occurrences show the beginning and end of the unit of behavior (the actions performed by an operation or event) that is triggered by a specific message.

Note

To animate a sequence diagram automatically, set the `SequenceDiagram::General::AutoLaunchAnimation` property to one of these options:

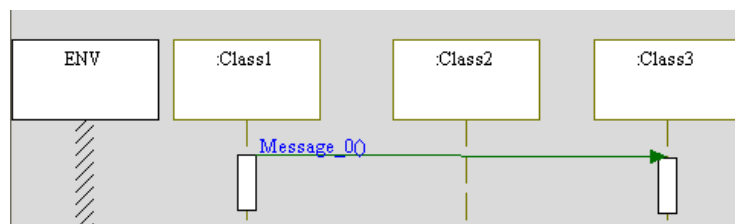
- ♦ **Always** launches the sequence diagram automatically.
- ♦ **If In Scope** launches animation only if the sequence diagram is in the active component scope.

The **Never** option for this property prevents automatic animation and is the default.

There are two ways to draw interaction occurrences:

1. Select the message in the sequence diagram, right-click, and select **Add Execution Occurrences**.
2. Click the Execution Occurrences button in the **Diagram Tools**, then select the appropriate message in the sequence diagram.

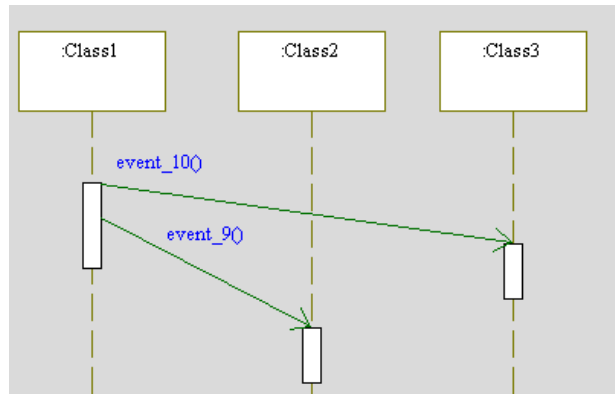
You can have Rational Rhapsody create execution occurrences automatically when you create messages by setting the property `SequenceDiagram::General::AutoCreateExecutionOccurrence` to `Checked`. Execution occurrences are drawn at the beginning and end of the message.



Note the following information:

- ♦ If you move a message, its execution occurrences move with it.
- ♦ You can resize execution occurrences (lengthwise), but cannot move them.

- ◆ If you move a message with execution occurrences or resize them so they overlap other execution occurrences, they are “merged,” as shown in this example.



Deleting execution occurrences

You can delete execution occurrences in two ways:

- ◆ Select the execution occurrence and then click **Delete** (or use the **Delete from Model** option in the menu)
- ◆ Delete the start message (the message assigned to the execution occurrences). This automatically deletes the execution occurrences “owned” by that message.

Shifting diagram elements with the mouse

You can use the following mouse actions to shift all or some of the elements in a sequence diagram.

To shift the entire diagram upward or downward:

1. Verify that an instance line is not currently selected.
2. Position the mouse in the area above the classifier role names.
3. Hold down the Shift key and drag the mouse up to shift the entire diagram upward, or drag down to shift the entire diagram downward.

To shift groups of elements upward or downward.

1. Verify that an instance line is not currently selected.
2. Position the mouse above the highest element that you want to shift.
3. Hold down the Shift key and drag the mouse up to shift the element and all the elements below it upward, or drag down to shift the element and all the elements below it downward

To shift groups of instance lines to the left or right:

1. Select any of the instance lines in the diagram.
2. Position the mouse to the left of the instance lines that you want to shift.
3. Hold down the Shift key and drag the mouse to the right to move all the elements on the right side of the mouse cursor to the right. Hold down the Shift key and drag the mouse to the left to move all the elements on the right side of the mouse cursor to the left.

Note: If you place the mouse on an instance line and drag while the Shift key is held down, Rational Rhapsody will shift the instance lines on the right of that instance line but not that instance line itself. “On an instance line” includes the entire width below the box that contains the name of the classifier role.

Display options

Most of the elements that can be added to a sequence diagram have options that you can set to affect how they are displayed in the diagram. While these options vary from element to element, the following options are common to the System Border, Instance Line, Message, and Reply Message elements:

- ♦ Select whether to display the name or the label of the element
- ♦ Show/hide any applied stereotypes

Sequence diagrams in the browser

The browser icon for sequence diagrams consists of two instance lines with messages passing between them. If the diagram is a unit, the icon has a small gray file overlay.

To open the menu for a sequence diagram, right-click the name of the diagram. The menu contains the following options:

- ♦ **Open Sequence Diagram** opens the selected sequence diagram in the drawing area.
- ♦ **Features** opens the Features window for the sequence diagram.
- ♦ **Features in New Window** opens the Features window for the sequence diagram in a separate window.
- ♦ **Add New** enables you to add a new dependency (see [Dependencies](#)), annotation (see [Annotations for diagrams](#)), hyperlink (see [Hyperlinks](#)), or tag (see [Use tags to add element information](#)).
- ♦ **References** enables you to search for references to the diagram in the model (see [Finding element references](#)).
- ♦ **Unit** enables you to either make the sequence diagram a unit that you can add to a CM archive (**Save**) or modify an existing unit (**Edit Unit**).
- ♦ **Configuration Management** provides access to common CM operations for the sequence diagram, including Add to archive, Check In, Check Out, Lock, and Unlock.
- ♦ **Format** changes the format used to draw the element (color, line style, and so on). For more information, see [Change the format of a single element](#).
- ♦ **Delete from Model** deletes the sequence diagram from the entire model.

Animation for selected classes

As part of the reverse engineering workflow, you might want to animate selected classes without the defining configurations. To accomplish this, after reverse engineering create one or more sequence diagrams, drag reverse engineered classes onto these diagrams, and select the ones to animate. The resulting animation shows the communication between the selected classes.

Sequence diagram comparison

During the development process, sequence diagrams (SDs) are used for the following primary purposes:

- ◆ In the early system requirements phase, they are used for use case description.
- ◆ In the implementation phase, they verify that all conditions are met in terms of communication between classes.
- ◆ In the testing phase, they capture the actual system trace.

Therefore, there is a need to facilitate comparison between SDs because, in principle at least, they should be identical. The execution message sequence should match the specification message sequence. The Sequence Diagram Comparison tool enables you to perform comparisons, for example between hypothetical and actual message sequences. You could also use this tool to compare two runs for regression testing.

If all execution SDs are identical to their corresponding specification (nonanimated) SDs, the system satisfies the requirements as captured in the use cases. However, if there are differences, you need to determine whether the specification was inaccurate or an error exists in the implementation. In both cases, you should correct the modeling error (either in the statechart or the SD) and then repeat the testing cycle to determine whether you have fixed the problem.

Sequence comparison algorithm

When comparing sequences, the following message parameters are used to determine whether the messages in the two SDs are identical:

- ◆ Departure time
- ◆ Arrival time
- ◆ Arguments

One simple approach involves comparing the exact position of every message and stopping at the first difference. However, this is probably too naive a comparison. For example, if there is a time offset in one SD, this kind of comparison would stop at the first message.

A more useful approach, therefore, is to take all events (message departures and arrivals) in order, and compare them without using the exact time. This kind of comparison, although simple, still shows when two SDs are essentially identical.

Because some messages can be “noise,” the comparison algorithm should also be able to decide whether a message is legitimate, and if not, mark it and continue with the comparison starting with the next message.

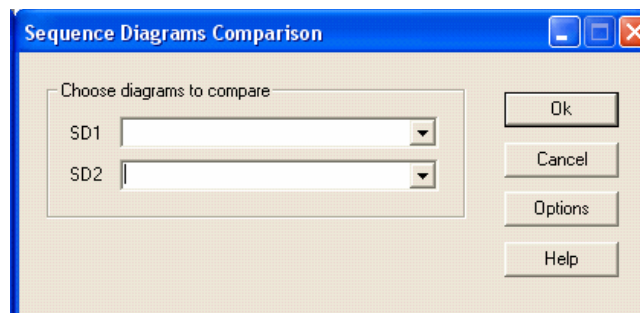
The point in comparing two SDs is not to show when one sequence is identical to another, but rather where and why they are different. Therefore, yes/no answers are not sufficient. Proper results must detail precisely what is identical and what is different. This is the approach that Rational Rhapsody takes when comparing message sequences.

Comparing sequence diagrams

Once you have saved two SDs illustrating the same scenario, for example, a specification and an execution version or two subsequent runs to test for regression, you are ready to start the sequence comparison.

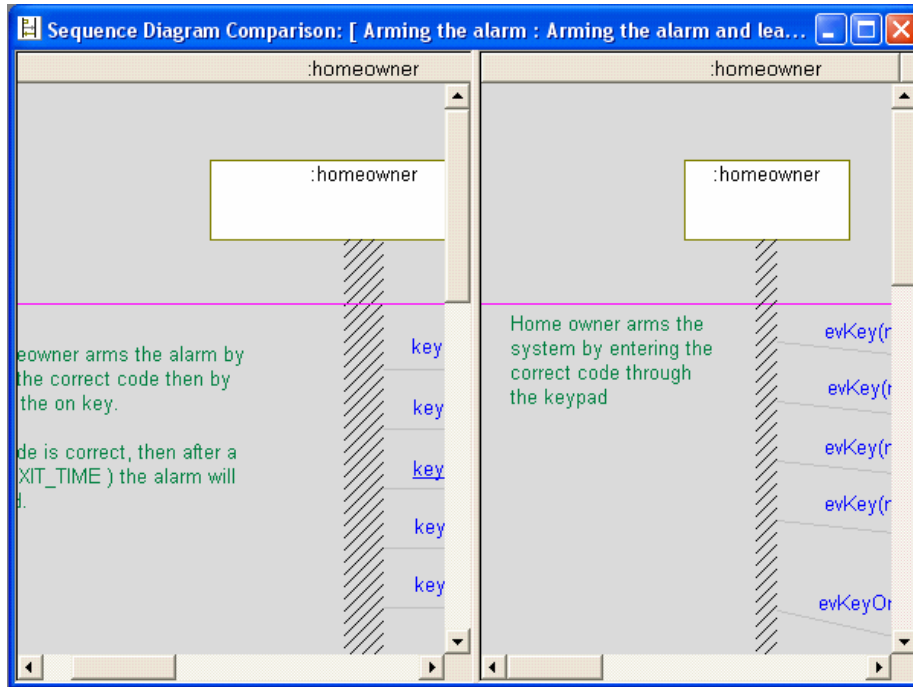
To start the comparison:

1. Select **Tools > Sequence Diagram Compare**. The Sequence Diagrams Comparison window opens, as shown in the following figure.



2. Using the **SD1** and **SD2** list controls, select two sequence diagrams to compare.
3. Set options for the sequence comparison as wanted. See [Sequence comparison options](#).
4. When all options are set and you are ready to start the comparison, click **OK**.

The result of the comparison displays as a dual-pane window with the diagram selected for SD1 on the left, and the diagram selected for SD2 on the right. Both panes are read-only. The following figure shows sample results.



The messages displayed in both panes are color-coded based on the comparison results. The following table lists the color conventions used in the comparison.

Arrow Color	Name Color	Description
Green	Blue	Message matches in both SDs
Pink	Pink	Message is missing in the other SD
Green	Pink	Message has different arguments in the other SD
Orange	Orange	Message arrives at a different time in the other SD
Gray	Gray	Message was excluded from comparison

Sequence comparison options

Specification Sequence Diagrams show only one specific thread from the mind of the designer. Therefore, certain instances and messages will be missing. On the other hand, execution (animated) Sequence Diagrams reflect the full collaboration between objects. This is why the simple comparison between specification and execution Sequence Diagrams always fails. Rational Rhapsody provides various options that enable you to compensate for some of the necessary differences between the two kinds of diagrams when doing a sequence comparison.

Select **Options** in the Sequence Diagrams Comparison window to open the Sequence Diagram Comparison options window. This window contains the following tabs:

- ♦ [The General tab for the sequence comparison](#)
- ♦ [The Message Selection tab](#)
- ♦ [The Instance Groups tab](#)
- ♦ [The Message Groups tab](#)

The following sections describe how to use these tabs in detail.

The General tab for the sequence comparison

The **General** tab allows you to specify whether to use synchronization and to save or upload your option settings. The tab contains the following fields:

- ♦ **Synchronization** specifies whether to ignore the arrival times of messages. For more information, see [Ignoring message arrival times](#).
- ♦ **Save** saves your option settings to a file that you can reuse. For more information, see [Saving and loading options settings](#).
- ♦ **Load** loads your option file. For more information, see [Saving and loading options settings](#).

Ignoring message arrival times

Sometimes the order of arriving messages is insignificant. The **Synchronization** option enables you to ignore the arrival times of messages and consider only the order in which they are sent.

In the resulting comparison display, equivalent messages are vertically synchronized in the adjacent window panes. This helps you to locate corresponding messages in both diagrams.

To enable or disable the synchronization option, select or clear the **Synchronization** box in the **General** tab.

Saving and loading options settings

You can save your options settings to a file and then reload them for subsequent message comparisons.

To save the settings:

1. Click the **Save** button on the **General** tab.
2. The Save As window opens. The default name for the options file is composed of the first words of the titles of each of the diagrams being compared separated by an underscore:

`<SD2>_<SD1>.sdo`

The file extension `.sdo` stands for Sequence Diagram Options. If wanted, edit the path and default name for the options file.

3. Click **OK**.

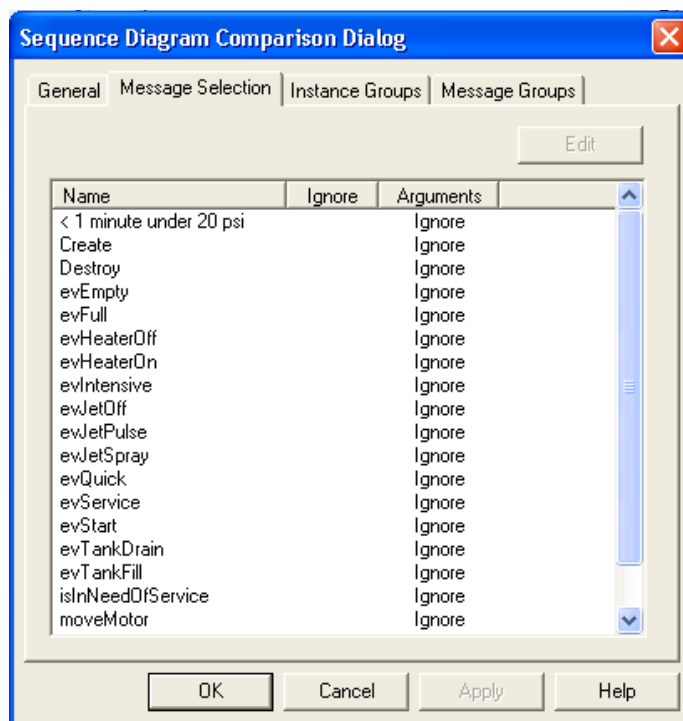
To reload your option settings:

1. In the **General** tab, click **Load**. The Open window is displayed.
2. Select the `.sdo` file that contains your option settings.
3. Click **Open**.

The sequence comparison options are restored to the settings last saved in the file.

The Message Selection tab

The **Message Selection** tab, shown in the following figure, enables you to select which messages to include and whether to include arguments in the comparison.



On this **Message Selection** tab, the word “Ignore” is the default setting for the Arguments column for all messages. This means that, by default, argument comparison is ignored for messages.

Using this tab, you can:

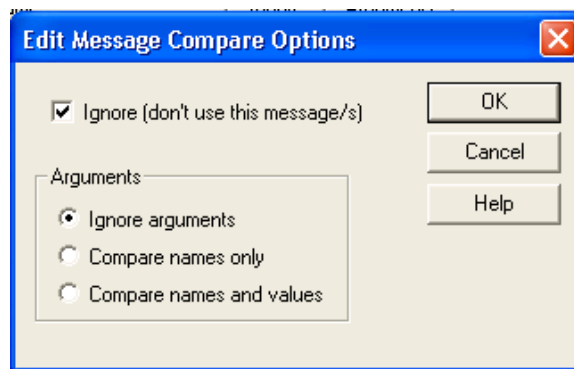
- ◆ Exclude a message from the comparison (see [Excluding a message in the comparison](#))
- ◆ Compare arguments (see [Comparing arguments](#))

Excluding a message in the comparison

Specification SDs typically include information that is essential to a particular use case or scenario. In many cases, they exclude the initialization phase messages, whereas execution SDs include all messages. Therefore, it might be necessary ignore certain messages when doing a comparison, such as constructors. Ignored messages are inaccessible in the resulting comparison window.

To exclude a message from the comparison:

1. Select the message to exclude.
2. Click **Edit**. The Edit Message Compare Options window opens.



3. Click the **Ignore** box to exclude the message from the comparison.
4. The three radio buttons, allow you to specify the way to treat **Arguments** associated with the selected message.
5. Click **OK**.

Comparing arguments

There are two options for determining whether messages are identical: the first is to compare the message names and all arguments, the second is to compare only the message names. The latter option is more useful because SDs show four different kinds of arguments:

- ◆ Unspecified arguments
- ◆ Actual values
- ◆ Formal names
- ◆ Both names and values

In specification SDs, you might not always provide complete information about message arguments. Because execution SDs record what the system actually does, they always show both

argument values and names. Therefore, the message comparison ideally should not use arguments but rather focus primarily on message names.

When two messages are named identically, you can compare their arguments.

For example, consider messages called `evDigitDialed(Digit)`. They would be equivalent if you compared only their argument names (`Digit`). However, if you compared their values (`EvDigitDialed(Digit=0)`, `EvDigitDialed(Digit=1)`, and so on), their argument values would not be equivalent.

Argument comparison occurs in the following steps:

1. Find each argument.
2. Find the argument name and value.
3. Determine whether to use the name, the value, or both for the comparison.

To specify whether to use argument names or values:

1. In the **Message Selection** tab, select a message and click **Edit**. The Edit Message Compare Options window opens.
2. Select one of the following options:
 - ♦ **Compare Names Only** compares argument names, but ignore their values.
 - ♦ **Compare Names and Values** compares both argument names and values.

These are commonly used settings:

Specification SD	Execution SD	Value
Message()	Message(Arg = 1)	Ignore Arguments
Message(Arg)	Message(Arg = 1)	Compare Names Only
Message(1)	Message(Arg = 1)	Compare Names and Values

3. Click **OK**.

Depending on your selections, the following labels are displayed in the Arguments column on the **Message Selection** tab:

- ♦ **Disable** means messages for which arguments should be ignored
- ♦ **Name** means messages for which argument names should be compared, but not the argument values
- ♦ **Value** means messages for which both argument names and values should be compared

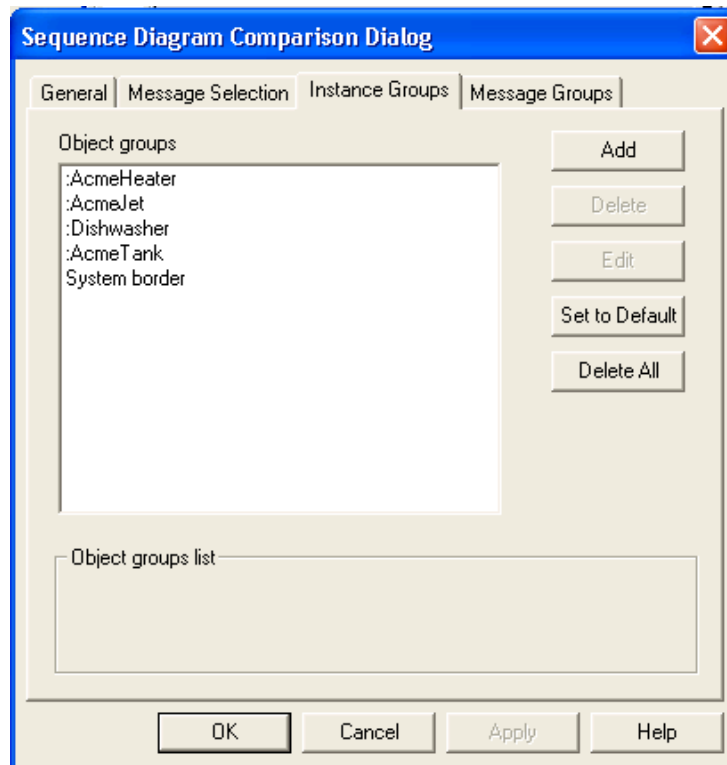
The Instance Groups tab

In specification SDs, all messages sent by the environment come from specific objects. In execution SDs, however, these messages could potentially come from you interacting with the animation. This difference can impair the comparison.

In general, requiring a complete object match between execution and specification SDs is too rigorous a requirement. The solution is to associate objects in one SD with other objects in the other SD. Messages can then match if their source and target objects are associated in both SDs.

To associate objects with each other you create *object groups*. Object groups are, in essence, instance abstractions that bridge the gap between high-level use cases and actual implementation, or between black-box and white-box scenarios. Using object groups, you can then compare objects that do not have the same name, or compare one object to several other objects.

To view object groups, select the **Instance Groups** tab in the Sequence Diagram Comparison options window. The **Instance Groups** tab, shown in the following figure, displays a list of the existing object groups in the model. There is one object group for each object, which is, by default, the only member of its own group.



The objects that belong to the group are displayed in the **Objects groups list** at the bottom of the window. This means that the objects listed for SD1 are considered logically the same as those listed for SD2.

The **Instance Groups** tab enables you to perform the following operations:

- ♦ **Add** creates a new object group (see [Creating object groups](#))
- ♦ **Delete** deletes an object group (see [Deleting object groups](#))
- ♦ **Edit** modifies an existing object group (see [Modifying object groups](#))
- ♦ **Set to Default** resets an object group (see [Resetting object groups](#))
- ♦ **Delete All** deletes all object groups (see [Deleting object groups](#))

Creating object groups

To create a new instance group:

1. On the **Instance Groups** tab, click **Add**. The Edit Object Group window opens. The default name for new object groups is `ClassBufn`, where n is an integer starting with 1.
2. If wanted, edit the name of the new object group.
3. To add objects to the group, move one or more unused objects from either of the boxes on the right to the corresponding box on the left.
4. Click **OK**.

Deleting object groups

To delete an existing object group:

1. On the **Instance Groups** tab, select the object group you want to delete.
2. Click **Delete**.
3. Click **OK**.

Any objects that belonged to the deleted group are now unused and available to be assigned to another object group.

To delete all instance groups:

1. On the **Instance Groups** tab, click **Delete All**.
2. Click **OK**.

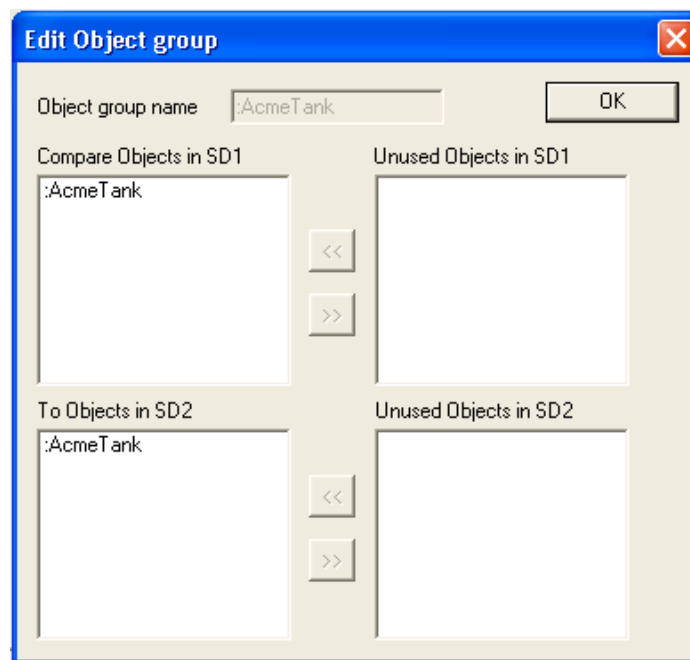
All objects are now unused and available to be assigned to a new object group.

Modifying object groups

If you want to associate different objects than the ones shown, either move one or more of the objects to a different object group or create a new group. In either case, you first need to remove the object you are moving from the group it is currently in, because an object can only belong to one group at a time.

To remove an object from a group:

1. On the **Instance Groups** tab, select an object group.
2. Click **Edit**. The Edit Object group window opens.



The name of the selected object group is displayed at the top of the window. The name box is unavailable because you cannot edit it here.

The Edit Object Group window contains four boxes. The two on the left show which objects in SD1 will be associated with which objects in SD2. The two boxes on the right show which objects in each diagram are currently not assigned to any group, and are therefore available to be assigned to a group.

3. Select an object in one of the boxes on the left and click the right arrows button.

To add an object to the group:

1. Select an object in one of the boxes on the right, and move it with the left arrows button.
2. Click **OK**. The selected object in one diagram is now available to be added to another group.
3. On the **Instance Groups** tab, select the new group for the object and click **Edit**.
4. Select the object in the **Unused Objects in SD<number>** box in the lower, right corner and click the left arrows key to add it to the group.
5. Select the object in the **To Objects in SD<number>** box in the lower, right corner and click the right arrows button to remove it from the group.
6. Click **OK**.

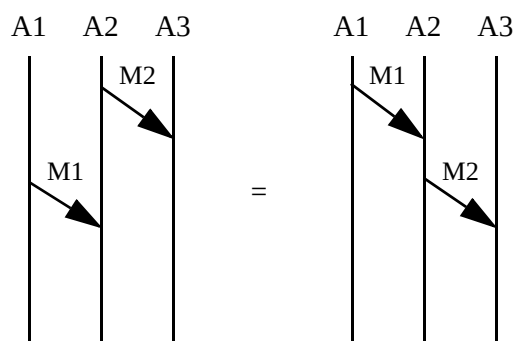
Resetting object groups

To set all object groups back to the default of one group per object, click **Set to Default** on the **Instance Groups** tab. An object group is added for each object with the same object in SD1 and SD2 belonging to the group.

The Message Groups tab

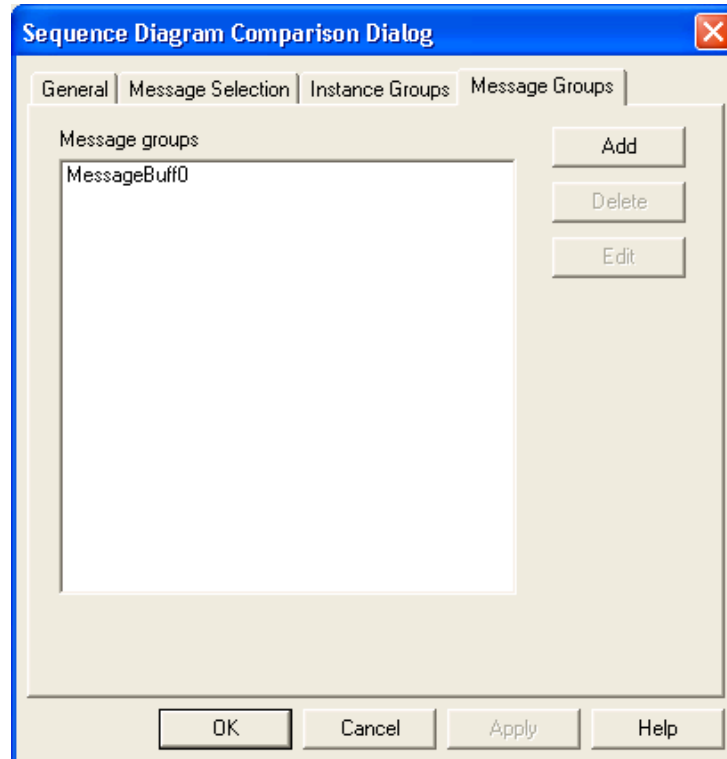
In specification SDs, you often must assume how the message queue works to determine the sequence of messages. It is highly likely that in specification SDs the order of messages will be different than the actual one specified in the statechart. An incorrect ordering assumption can result in large mismatch.

To avoid this problem, the comparison must be able to ignore the timing of messages. For example, a message M1 sent by an instance A1 after a message M2 sent by an instance A2 could match the same message sent before M2:



There can also be cases where two or more messages should be sent at the same time, but the order is not important. *Message groups* enable you to specify groups of messages for which ordering is not important. There is a match if any message in the group occurs in any order.

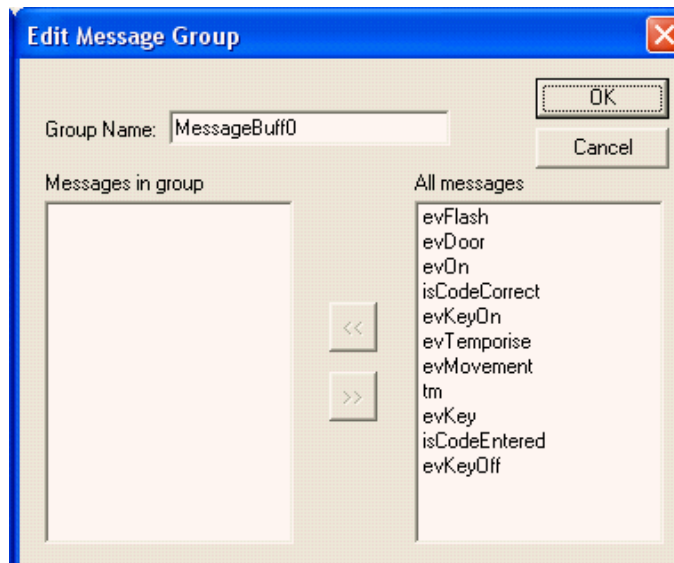
The **Message Groups** tab, shown in the following example, enables you to create, modify, and delete message groups.



Creating message groups

To create a message group:

1. On the **Message Groups** tab, click **Add**. The Edit Message Group window opens.



The default name for new message groups is MessageBuff n , where n is an integer starting with 0.

2. If wanted, edit the name of the new message group.

The Edit Message Group window contains two list boxes: the left one shows the messages that currently belong to message group, whereas the right one shows all messages in the two SDs being compared. Messages can belong to more than one message group.

Adding a message to a message group

To add a message to the message group:

1. On the **Message Groups** tab, click **Add**.
2. Select a message from the **All Messages** list and click the left arrows button to move it to the **Messages in group** list. For multiple selections, use **Shift+Click** or **Ctrl+Click**.
3. Click **OK**.

Removing a message from a message group

To remove a message from the message group:

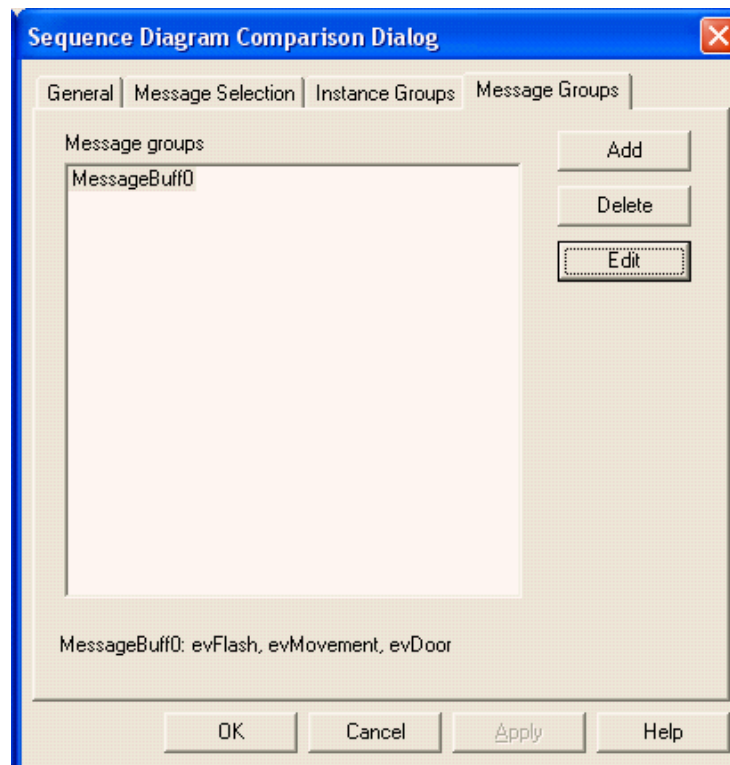
1. On the **Message Groups** tab, click **Add**.
2. Select a message from the **Messages in group** list and click the right arrows button to move it to the **All Messages** list. For multiple selections, use **Shift+Click** or **Ctrl+Click**.
3. Click **OK**.

Determining the message group members

To see which messages belong to a message group:

1. On the **Message Groups** tab, select a message group from the list.

The messages that belong to that group are listed at the bottom of the window, as shown in this example.



2. Click **OK**.

Modifying message groups

To modify an existing message group:

1. On the **Message Groups** tab, select a message group from the list and click **Edit**. The Edit Message Group window opens.
2. Select a message in the **Messages in group** list and click the right arrows button to remove it from the group.

Select a message in the **All Messages** list and click the left arrows button to add it to the group.

3. Click **OK**.

Deleting message groups

To delete an existing message group:

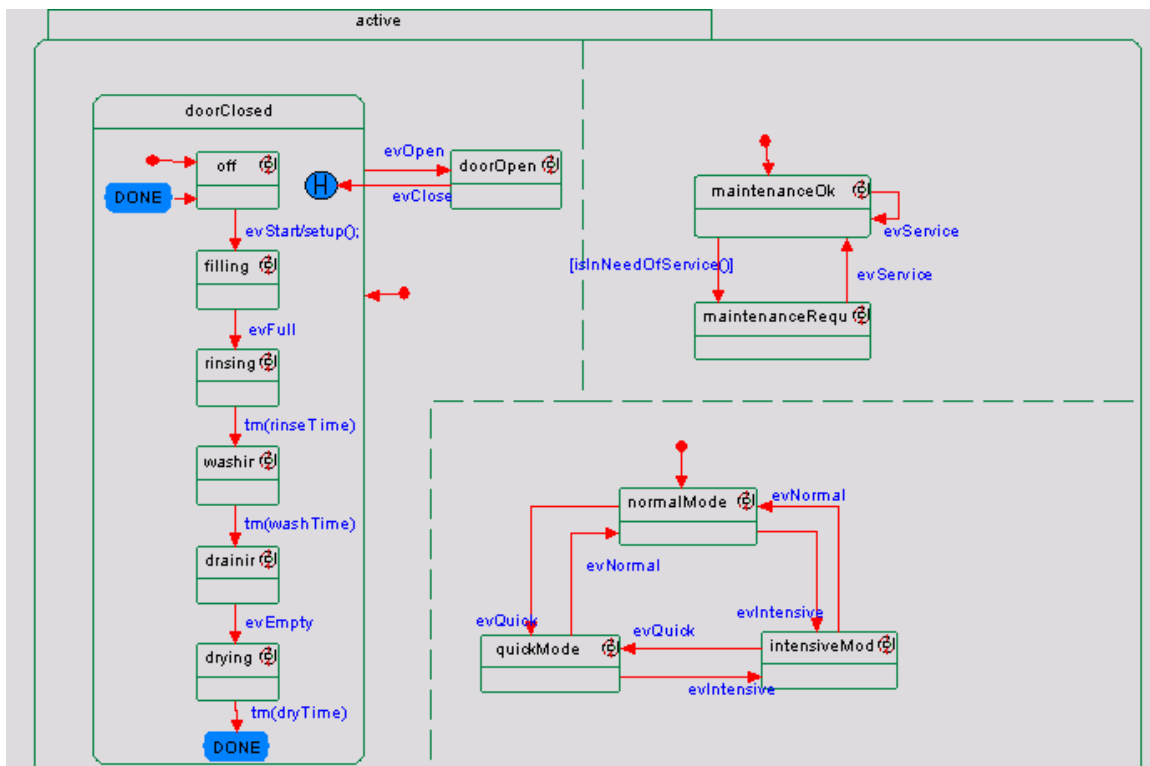
1. In the Sequence Diagram Comparison options window, select the message group to delete.
2. Click **Delete**.
3. Answer **OK** to the confirmation prompt.

Statecharts

Statecharts define the behavior of objects by specifying how they react to events or operations. The reaction can be to perform a transition between states and possibly to execute some actions. When running in animation mode, Rational Rhapsody highlights the transitions between states.

Statecharts define the run-time behavior of instances of a class. A state in a statechart is an abstraction of the mode in which the object finds itself. A message triggers a transition from one state to another. A message can be either an event or a triggered operation. An object can receive both kinds of messages when sent from other objects. An object can always receive events it sends to itself (self-messages). In Rational Rhapsody, statecharts are part of the object-oriented paradigm. The more complicated classes can have statecharts; simpler classes do not require them.

You can use operations and attributes in classes with statecharts to define guards and actions, as in the following example.

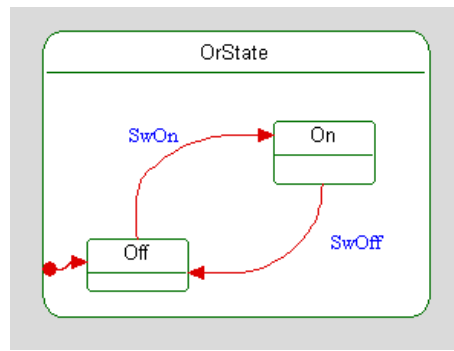


States

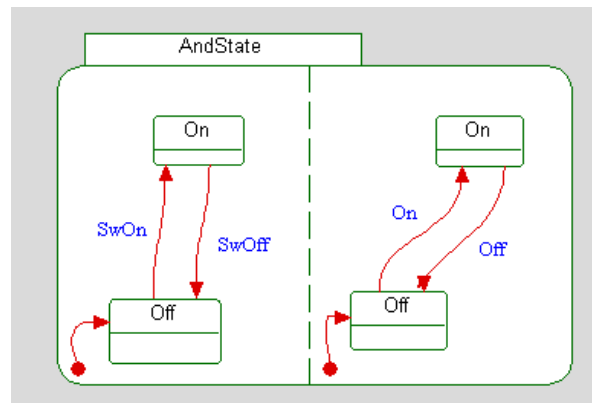
A *state* is a graphical representation of the status of an object. It typically reflects a certain set of its internal data (attributes) and relations. In statecharts, states can be broken down hierarchically as follows:

- ♦ **Basic (leaf) state** is a state that does not have any substates.
- ♦ **Or state** is a state that can be broken down into exclusive substates. This means that the object is exclusively in one or the other of its substates.

In the following example, there are possible two states: On and Off.



- ♦ **And state** An object is in each of its substates concurrently. Each of the concurrent substates is called an *orthogonal component*. You can convert an Or state to an And state by dividing it with an And line. For more information, see [And lines](#).



You set the statechart implementation in the **Settings** tab of the Configuration window in the browser.

Opening an existing statechart

To open an existing statechart in the drawing area:

1. Click the appropriate button in the **Diagrams** toolbar to the Open Statechart window opens for you to select the diagram.
2. For statecharts, select the class that the diagram describes from the list of available diagrams. For all other diagrams, select the diagram you want to open.
3. Click **OK**. The diagram opens in the drawing area.

As with other Rational Rhapsody elements, use the Features window for the diagram to edit its features, including the name, stereotype, and description. For more information, see [The Features window](#).

Statechart drawing tools

The **Diagram Tools** for a statechart includes the following tools:

Drawing Tool	Name	Description
	State	Indicates the current condition of an object, such as On or Off. For more information, see States .
	Transition	Represents a message or event that cause an object to transition from one state to another. For more information, see Transitions .
	Initial connector	Shows the default state of an object when first instantiated. For more information, see Initial connectors .
	Add line	Separates the orthogonal components of an And state. There can be two or more orthogonal components in a given And state and each behaves independently of the others. For more information, see And lines .
	Decision Node	Shows the branches on transitions, based on Boolean conditions called guards. For more information, see Decision nodes .
	History connector	Stores the most recent active configuration of a state. An transition to a history connector restores this configuration. For more information, see History connectors .
	Termination connector	Ends the life of the object. For more information, see Termination connectors .
	Merge Node	Joins multiple transitions into a single, outgoing transition. For more information, see Merge nodes .

Drawing Tool	Name	Description
	Diagram connector	Joins physically distant transition segments. Matching names on the source and target diagram connectors define the jump from one segment to the next. For more information, see Diagram connectors .
	EnterExit point	Represents the entry to / exit from sub-statecharts. For more information, see EnterExit points .
	Join node	Merges multiple incoming transitions into a single outgoing transition. For more information, see Activity diagrams .
	Fork Node	Separates a single incoming transition into multiple outgoing transitions. For more information, see Activity diagrams .
	Transition Label	Add or modify a text describing an transition.
	Termination State	Signifies either local or global termination, depending on where they are placed in the diagram.
	Dependency	Indicates a dependent relationship between two items in the diagram. For more information, see Activity diagrams .
	Send Action	Represents the sending of events to external entities. For more information, see Send action elements .

The following sections describe how to use these tools to draw the parts of a statechart. For basic information on diagrams, including how to create, open, and delete them, see [Graphic editors](#).

Drawing a state

To draw a state:

1. Click the **State** button  in the **Diagram Tools**.
2. Click-and-drag or click in the drawing area to create a state with a default name of state_n, where *n* is an incremental integer starting with 0.
3. If wanted, change the state name, then press **Enter**.

States include a standard name compartment.

State name guidelines

When naming states, follow these guidelines:

- ♦ Must be identifiers.

- ◆ Do not include spaces, “My House” is not valid. Use “MyHouse.”
- ◆ Must be unique among sibling states.
- ◆ Should not be the same as the names of any events or classes in the model.

Features of states

The **Features window** allows you to add and change the features for a state. A state has the following **General** tab features:

- ◆ **Name** specifies the name of the state.
- ◆ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ◆ **Stereotype** specifies the stereotype of the state, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).

Note: The COM stereotypes are constructive; that is, they affect code generation.

Action on entry

The Action on Entry specifies the action that should be executed whenever the system enters this state, regardless of how the system arrived here.

If the action on entry value is overridden, the **Overridden** check box is checked. The **Overridden** check box is available in the Features windows for textual information in statecharts (state entry and exit actions, and guards and actions for transitions and static reactions). By enabling or disabling this check box, you can easily override and unoverride statechart inheritance without actually changing the model. As you toggle the check box on and off, you can view the inherited information in each of the window fields, and can decide whether to apply the information or revert back to the currently overridden information. For more information, see [Overriding textual information](#).

Action on exit

The Action on Exit specifies the action that should be executed whenever the system exits this state, regardless of how the system exits.

If the action on entry value is overridden, the **Overridden** check box is checked.

Reactions in state

The Reaction in State specifies the trigger, guard, and actions identified in an transition label. If the trigger occurs and the guard is true, the action is executed. For more information, see [Adding or modifying activity flow labels](#). Use the appropriate window button:

- ♦ **New** creates a new reaction in state. If you select this option, the Reaction Features window opens so you can specify the Trigger, Guard, and Action for the new reaction.
- ♦ **Edit** modifies an existing reaction.
- ♦ **Delete** deletes a reaction.

Note: If you specify action on entry or exit behavior for a state, this icon is added to the state display.

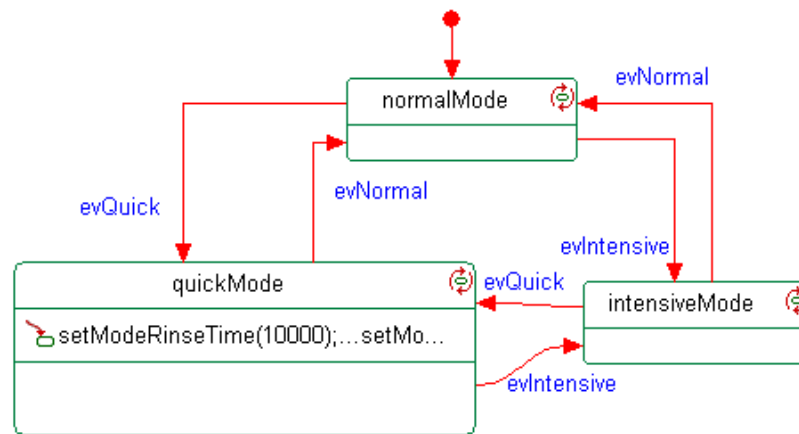


Display options for states

Using the **Display Options** option in the menu, you can specify whether:

- ♦ States are displayed with a name or label.
- ♦ Stereotypes are displayed with a name, label, or icon.
- ♦ The exit and entry actions are displayed.

The display for the quickMode state includes its entry and exit actions.



Termination states

A *termination state* provides local termination semantics. Local termination implies the completion of a composite state without the destruction of the context instance.

There are two different modes for local termination:

- ♦ **Statechart mode** where local termination applies only to composite states with final activities inside them.
- ♦ **Activity diagram mode** where local termination applies to every block and composite state, even those that do not have internal final activities. (This is the UML statechart/activity diagram-supported mode.)

The `CG::Statechart::LocalTerminationSemantics` property specifies whether activity diagram mode of local termination is available. The default value of `Cleared` means that the activity diagram mode of local termination is unavailable (statechart mode is active).

Local termination code with the reusable statechart implementation

The following sections describe how code is generated to support local termination when you use the reusable statechart implementation.

Or states in reusable statechart

Code is generated for local termination of Or states with the reusable statechart implementation as follows:

- ♦ For each final activity, a `Concept` class data member of type `FinalState` is generated. A new class is not created as for other state types.
- ♦ The following local termination guard is added to each outgoing null transition from an Or state that needs local termination semantics:

```
&& IS_COMPLETED(state)
```

- ♦ If an Or state has several final activities, an outgoing null transition is activated when any one of them is reached. However, the specific connector is instrumented.
- ♦ The `isCompleted()` function is overridden for an Or state that has a final activity, returning `True` when the final activity is reached. The function is also overridden for an Or state without a final activity in activity diagram mode, always returning `False`.
- ♦ An instance of a `FinalState` is created by a line similar to the following example:

```
FinalA = new FinalState(this, OrState, rootState,  
    "ROOT.OrState.FinalA");
```

And states in reusable statechart

Code is generated for local termination of And states with the reusable statechart implementation as follows:

- ◆ The following local termination guard is added to each outgoing null transition from an And state if one of the components has a final activity:

```
&& IS_COMPLETED(AndState)
```

In this case, the `isCompleted()` function of the `AndState` framework class is called.

- ◆ The following local termination guard is added to a join transition for each Or state that is a source of the transition:

```
&& IS_COMPLETED(state)
```

- ◆ If a source state of a join transition is a simple state (leaf state), its guard is as follows:

```
(IS_IN(state))
```

The following example shows the code generated for a join transition with a real guard and local termination guards, where `C1` and `C2` are Or states with final activities and `C3` is a leaf state:

```
if(RealGuard() && IS_COMPLETED(C1) && IS_COMPLETED(C2) && IS_IN(C3))
```

Local termination code with flat statechart implementation

The following sections describe how code is generated to support local termination when the flat statechart implementation is used.

Or states in flat statechart

Code is generated for local termination of Or states with the flat statechart implementation as follows:

- ◆ For each final activity, the new state enumeration value is generated (as for a regular state).
- ◆ For each Or state with a final activity, a `<StateName>_isCompleted()` operation is generated. This operation returns an `OMBoolean` value of `True` when the state is completed. If the `CG::Class::IsCompletedForAllStates` property is checked, the operation is generated for all states.

The following example shows the code generated for a `<StateName>_isCompleted()` operation where `FinalA` and `FinalB` are final activities in the Or state:

```
inline OMBoolean
class_0::OrState_isCompleted() {
    return (FinalA_IN() || FinalB_IN());
}
```

- ◆ The following local termination guard is added to each outgoing null transition from an Or state that needs local termination semantics:

```
&& IS_COMPLETED(state)
```

- ◆ Instrumentation information for `FinalState` is generated in the transition code (as for normal states).

And states in flat statechart

Code is generated for local termination of And states with the flat statechart implementation as follows:

- ◆ The following local termination guard is added to each outgoing null transition from an And state if one of the components has a final activity.

```
&& IS_COMPLETED(AndState)
```

In this case, the `isCompleted()` function of the `AndState` framework class is called:

- ◆ The `isCompleted()` operation of `AndState` calls the `IS_COMPLETED()` macro for all components that have a final activity. This operation returns `TRUE` only when all components are completed. If an And state does not have components with a final activity, the operation returns `TRUE` in statechart mode and `FALSE` in activity diagram mode.

The following example shows the `<StateName>_isCompleted()` function generated for an And state named `AndState`, with two components, `Component1` and `Component2`, each of which has a final activity:

```
OMBoolean class_0::AndState_isCompleted()
{
    if(IS_COMPLETED(Component1) == FALSE)
        return FALSE;
    if(IS_COMPLETED(Component2) == FALSE)
        return FALSE;
    return TRUE;
}
```

- ◆ Implementation of join transitions with the flat statechart implementation is the same as for the reusable statechart implementation (see [And states in reusable statechart](#)).

Transitions

A *basic transition* is composed of a single arrow between a source and a destination. Transitions represent the response to a message in a given state. They show what the next state will be, given a certain trigger. A transition can have a trigger, guard, and actions.

The transition context is the scope in which the message data (parameters) are visible. Any guard and action inherit the context of an transition determining the parameters that can be referenced within it.

The source of an transition can be one of the following items:

- ◆ State
- ◆ Initial connector
- ◆ History connector

The destination of an transition can be one of the following items:

- ◆ State
- ◆ Final activity
- ◆ History connector

Creating a statechart transition

To draw a statechart transition:

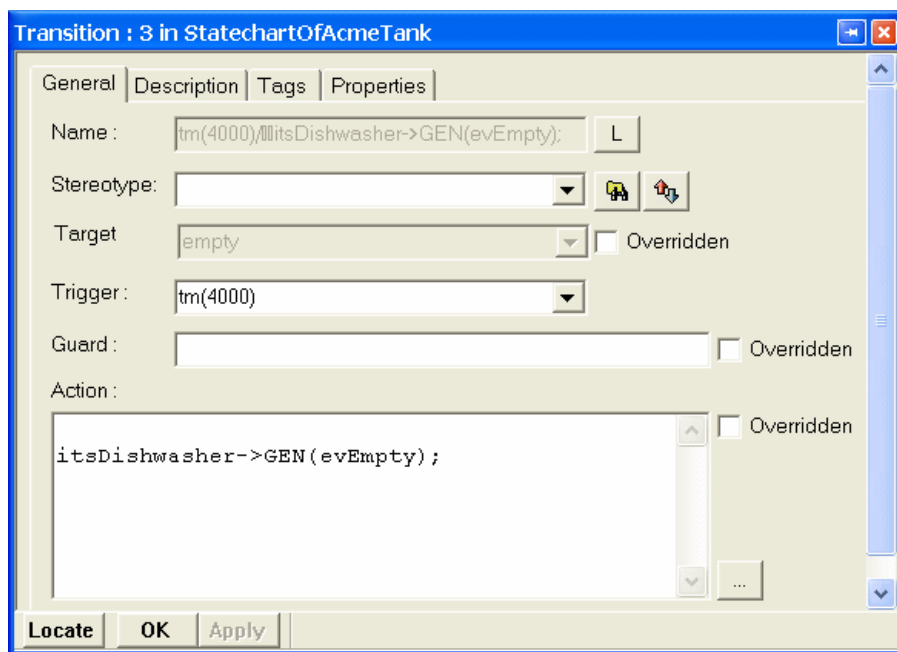
1. Click the **Transition** button  in **Diagram Tools**.
2. Click the bottom edge of the state to anchor the start of the transition.
3. Move the cursor to the top edge of the state and click to anchor the transition line.
4. In the label box, type the name of the event. Press **Ctrl+Enter** to terminate.

Note

Pressing Enter in an transition name, without simultaneously pressing Ctrl, simply adds a new line.

Features of transitions

Use the Features window to add and change the features for a transition, as shown in this example.



An transition has the following **General** tab features:

- ♦ **Name** specifies the name of the transition.
- ♦ **L** specifies the label for the element, if any. For information on creating labels, see [Descriptive labels for elements](#).
- ♦ **Stereotype** specifies the stereotype of the element, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. For information on creating stereotypes, see [Stereotypes](#).
Note: The COM stereotypes are constructive; that is, they affect code generation.
- ♦ **Target** specifies the target of the transition. This field is read-only.
- ♦ **Trigger** specifies the trigger for the transition. See [Triggers](#).

- ♦ **Guard** specifies the guard for an transition.

The **Overridden** check box is available in the Features windows for textual information in statecharts (state entry and exit actions, and guards and actions for transitions and static reactions). By enabling or disabling this check box, you can easily override and unoverride statechart inheritance without actually changing the model. As you toggle the check box on and off, you can view the inherited information in each of the window fields, and can decide whether to apply the information or revert back to the currently overridden information. For more information, see [Overriding textual information](#).

- ♦ **Action** specifies the transition action.

Types of transitions

You can create the following types of transitions:

- ◆ Compound transitions
- ◆ Forks
- ◆ Joins

Compound flows

A *compound flow* is composed of several transition segments connected by intermediate nodes. A transition segment is a subpart of a transition between any source, destination, or intermediate nodes.

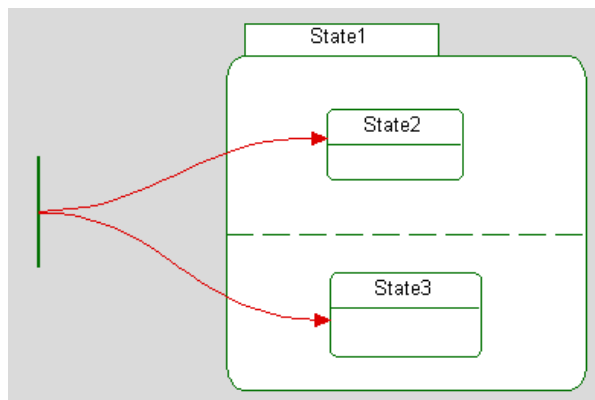
The intermediate nodes can be any of the following items:

- ◆ [Forks](#)
- ◆ [Joins](#)
- ◆ [Merge nodes](#)
- ◆ [Decision nodes](#)
- ◆ [Diagram connectors](#)

In semantic terms, forks and joins can both be nodes.

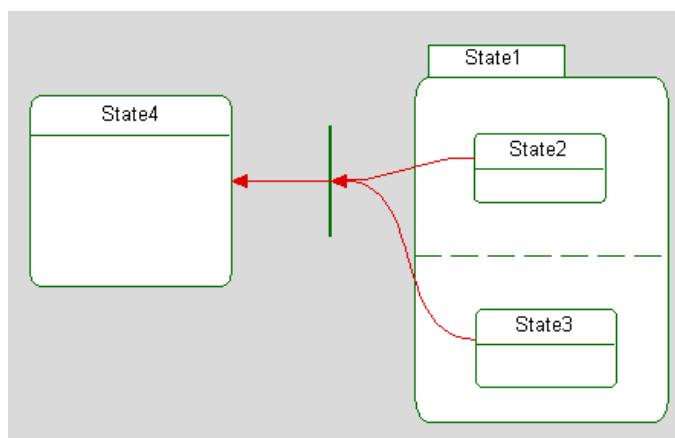
Forks

A *fork* is a compound transition that connects a transition to more than one orthogonal destination. The destination of a fork segment can be a state, final activity, or connector. However, it cannot have a label, as shown in the this example.



Joins

A *join* is a compound transition that merges segments originating from states, final activities, or connectors in different orthogonal components. Rational Rhapsody automatically generates a join when you combine source segments. The segments entering a join connector should not have labels, as shown in the following figure.



Note

If the model contains joins from more than two concurrent states, this generates an error. This is a Rational Rhapsody limitation in that the error is a violation of MISRA rule 33.

Selecting a trigger transition

For a list of transitions defined in the diagram, right-click a transition line in the diagram and select **Select Trigger**. The pop-up menu that displays lets you select one of the available triggers, including inherited triggers, that have already been defined for the class.

Notice that if there are more triggers that can appear on the pop-up menu, a **Browse** command displays. Click **Browse** to open the Select Trigger window that shows you all the triggers that are available.

See also [Selecting a message or trigger](#).

Transition labels

Transition labels can contain the following parts:

- ♦ [Triggers](#)
- ♦ [Guards](#)
- ♦ [Actions](#)

The syntax for transitions is as follows:

```
trigger [guard] /action
```

The following example shows a transition label consisting of a timeout trigger (see [Timeouts](#)), a guard, and an action:

```
tm(500)[isOk()]/printf("a 0.5 second timeout occurred\n")
```

In this example, the trigger is the timeout `tm(500)` and the guard is `[isOk()]`. The action to be performed if the trigger occurs and the guard is true is `printf("a 0.5 second timeout occurred\n")`.

All three parts of the transition are optional. For example, you can have a transition with only a trigger and an action, or only a guard. The following example shows a transition label consisting of only a trigger and an action:

```
clockw /itsEngine->GEN(start)
```

When typing a multiline transition label (for example, one that has several actions separated by semicolons), you can press **Ctrl+Enter** to advance the cursor to the next line and continue the label.

Triggers

Every transition is associated with a designated message, which is the *trigger* of the transition. In other words, the trigger of the transition is waiting for its event. A transition cannot be associated with more than one message. Triggers can be events, triggered operations, or timeouts.

Events are asynchronous; time can pass between the sending of the event and when it actually affects the statechart of the destination. Triggered operations are synchronous; their effect on the statechart of the destination is immediate.

This not a valid transition label:

```
e1 or e2
```

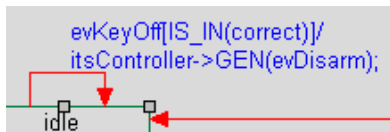
The trigger part of a transition label cannot use conditional expressions; however, guards can.

Events

Events originate in the analysis process as “happenings” within the system to which objects respond. Their concrete realizations are information entities carrying data about the occurrence they describe.

For designers, an event is a one-way, asynchronous communication between objects or between some external interface and the system (see [Events and operations](#)).

The following figure shows an event.



You can specify an event using the following methods:

- ◆ An event name
- ◆ The name of a triggered operation
- ◆ tm(time expression)

For timeout events, the time expression must be an integer expression, in milliseconds.

Event usage

Classes and their statecharts can receive events defined in any package in the model. Cross-package inheritance of statecharts for reactive classes is not allowed.

Event class hierarchy

An event is an instance of a particular event-class. Events can be subclassed to add attributes (event parameters). The base class for all events is `OMEvent`.

For example, windowing systems define several event classes: `MouseEvent` is a subclass of `InputEvent`, and `MouseClickedEvent` and `MouseMotionEvent` are subclasses of `MouseEvent`.

To make an event a subclass of another event:

1. Open the Features window for the event.
2. From the *Inherits:* list, select the event that is to serve as the base class.
3. Click **Apply** or **OK**.

Generating events

You generate an event by applying a `gen` method on the destination object, as follows:

```
client->gen(new event(p1,p2,...,pn))
```

The `generate` method queues the event on the proper event queue.

The framework provides a `GEN` macro, which saves you from having to use the `new` operator to generate an event. For example:

```
client->GEN(event(p1, p2, pN))
```

Event semantics

An event is created when it is sent by one object to another, then queued on the queue of the target object thread (thread partitioning is not covered in this guide). An event that gets to the head of the queue is dispatched to the target object. Once dispatched to an object, it is processed by the object according to the semantics of event propagation in statecharts and the run-to-completion semantics. After the event is processed, it no longer exists and the execution framework implicitly deletes it.

Internal events

An internal event occurs when an object sends a message to itself. To create an internal event, omit the destination object from the `send` operation, as follows:

```
GEN(warmEngine(95))
```

Private events

You can control which classes can generate events to which classes using friendship. In this way, you can ensure that events come from trusted classes only. The event request and queueing function is controlled by the `gen()` methods, which are public by default in the framework base class `OMReactive`. If you want to control the generation of events using friendship, make the first `gen()` method in `Share\oxf\OMReactive.h` protected. This is a one-time effort. Do not change the second `gen()` method, which is used for instrumentation.

Inside each application class, grant friendship to the classes that need to generate events for it. If you do not grant friendship, your program will no longer compile.

Adding operations to an event

Rational Rhapsody allows you can add operations to events you have defined. This allows you to add additional behavior to your events.

To add an operation to an event, right-click the event in the browser and select **Add New > Operation**.

The new operation displays below the event in the browser, and when code is generated, the operation displays in the class that represents the event.

Note

Roundtripping does not bring into the model any new operations that you have added to event classes in your code, nor does it bring into the model changes that were made directly to the body of operations that were previously created for events.

Events as attribute types

Events can be used as types for attributes.

Triggered operations

Triggered operations are services provided by a class to serve other classes. They provide synchronous communication between a client and a server object. Because its activation is synchronous, a triggered operation can return a value to the client object.

Unlike events, operations are not independent entities; they are part of a class definition. This means that operations are not organized in hierarchies.

The usage of operations corresponds to invocation of class methods in C++. There are three reasons why operations have been integrated with the statechart framework:

- ◆ They allow use of statecharts in architectures that are not event-driven to specify behaviors of objects in the programming sense of operations and object state.
- ◆ They provide for late design decisions to optimize execution time and sequencing by converting event communication into direct operation invocations.
- ◆ They allow the description of behaviors of (primitive) “passive” classes using statecharts.

Applying a triggered operation

A triggered operation is started in the same way as a primitive operation:

```
server->operation(p1, p2, ..., pn)
```

Or:

```
result = server->operation(p1, p2, ..., pn)
```

Operation replies

Operations can return a value. The return value for an operation *m* must be determined within the transition whose context is the message *m*, using the reply operation:

```
m/reply(7);
```

Note

A triggered operation might not result in another triggered operation on the same instance.

Making sure a triggered operation is called

There might be a problem with the reply from triggered operations if the receiver object is not in a state in which it is able to react to a triggered operation. If a triggered operation is called when not expected, incorrect return values might result.

Rather than use the `IS_IN` macro to determine what state the receiver is in, you can design your statechart so the triggered operation is never ignored. To do this, create a superstate encompassing the substates in the object, and in the superstate create a static reaction with a trigger to return the

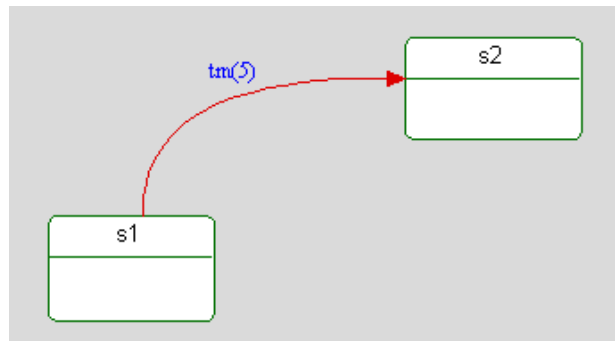
proper value. For example, to make sure that a sensor is always read regardless of what state an object is in, create a static reaction in the superstate with the following transition:

```
opRead/reply(getStatus())
```

This way, no matter what substate the object is in, it will always return the proper value. Although both the trigger to the superstate and that to a substate are active when in a substate, the trigger on the transition to the superstate is taken because it is higher priority. See [Transition selection](#).

Timeouts

Timeout triggers have the syntax `tm(<expression>)`, where *<expression>* is the number of time units. The default time unit is milliseconds. The time units are set based on the operating system adapter implementation of the tick timer. A timeout is scheduled when the origin state (s1) is entered. If the origin state has exited before the timeout was consumed, the timeout is canceled.



You can use the timeouts mechanism (`tm()`) when the quality of service (QOS) accuracy requirement conforms with the following timeout accuracy. When a timeout occurs, it is inserted to the event queue related to the reactive instance. The time on which the timeout is consumed depends on the actual system state. The timeout occurrence depends on three factors:

1. The timeout request time (T)
2. The tick-timer resolution (R)

The *resolution* specifies how often the system checks if there are expired timeouts.

3. Timeout latency (L)

The tick timer implementation for some operating system adapters is synchronous (using a call to `sleep(interval)`). This means that there is a built-in *latency* (the time spent processing the expired timeouts). This latency can be significant when the timeout is very long (involving many timer ticks).

The following formula determines when a timeout will expire:

$$[(T+L) - R, (T+L)+R]$$

Note: If you use triggered operations and events (including timeouts) in the same statechart and the triggered operation can be called from an object running in a thread other than the event consumption thread, it might lead to a race situation. To prevent the race, make the triggered operations guarded (which will also prevent the race with timeouts).

Null transitions

In some cases, it is useful to use a transition to leave a state without using a trigger. These are examples of such cases:

- ◆ When a state tries to allocate a resource that might not be available
- ◆ When you want to branch according to some entry action
- ◆ When you have a join transition

You can accomplish this with a null transition. A *null transition* is any transition without a trigger (event or timeout). Null transitions can have guards (for example, `[x == 5]`). The run-to-completion semantics of the Rational Rhapsody framework checks for an infinite (run-time) loop of null transitions, which might otherwise be difficult to detect.

You can modify the `maxNullSteps` number and recompile the framework if you need to change the number.

Guards

A *guard* is a conditional expression that is evaluated based on object attributes and event data. Rational Rhapsody does not interpret guards. They are host-language expressions, or simply code chunks, that must resolve to either a Boolean or an integer value that can be tested. Otherwise, the statechart code generated for guards will not compile.

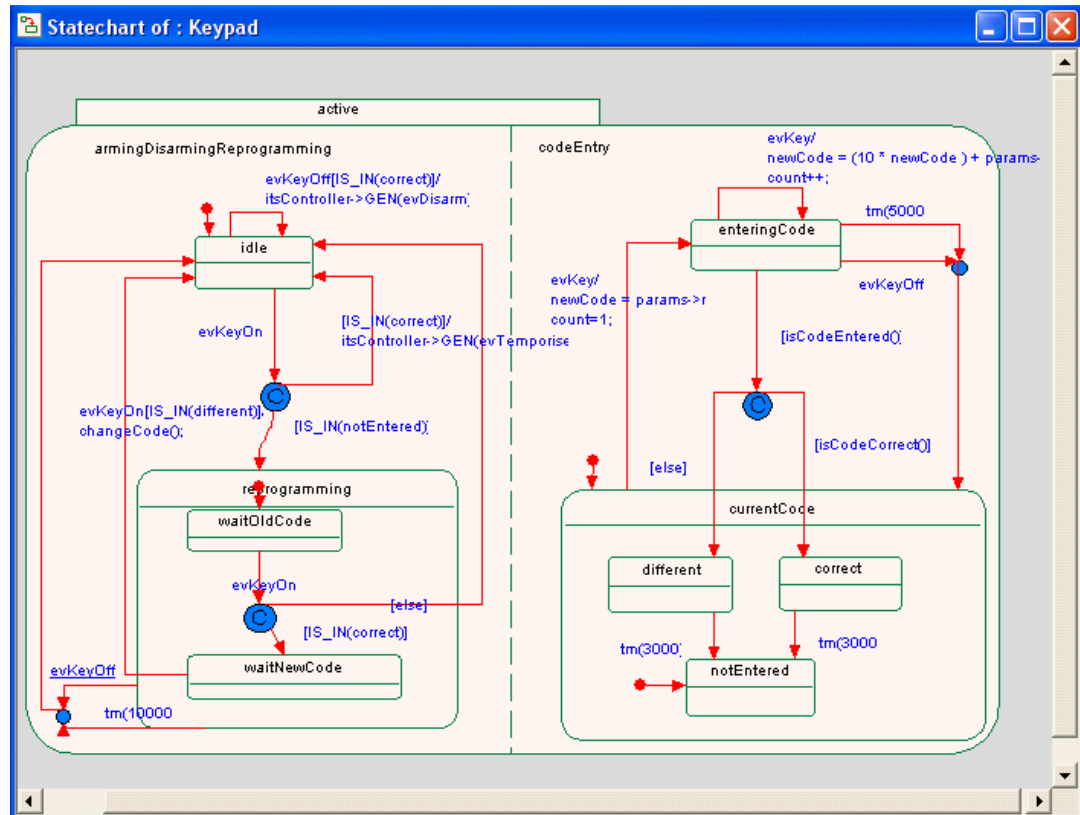
The following example shows a transition label that consists of a guard and an action that uses the `GEN` macro to generate an event:

```
[x > 7]/controller->GEN(A7Failures)
```

A transition can consist of only a guard. The low-to-high transition of the condition (or Boolean value) is considered to be the triggering event. For example, the following guard is a valid transition label:

```
[x > 7]
```

During animation, all guards without triggers are tested every time an event happens. The following statechart uses several guards without transitions.



This statechart is for the keypad of a home alarm system. When the keypad of the alarm system is in the idle state, you can enter a code to arm the alarm before leaving the house. After entering the code, you press the On button to turn the alarm on. Pressing the On button issues an `evKeyOn` event. Each time this event occurs, the state machine evaluates the two guards that come after the decision node, `[IS_IN(correct)]` or `[IS_IN(notEntered)]`, and follows the path of the one that evaluates to true.

By using an animated sequence diagram, you can see when a guard is tested. If you want to test a condition more frequently or at a more regular interval than whenever an event occurs, you can create a polling mechanism. To do this, create a short timeout transition from the state to itself so the guard is evaluated on at least these occasions. Alternatively, you can poll using another object and replace the guard in the current statechart with an event signaled from the polling object.

Note

Using guards that cause side-effects is not typical, because it might cause problems in the application.

Actions

An *action expression* is a sequence of statements. Like guards, actions are uninterpreted code chunks based on object attributes and event data.

There is no need to add a semicolon to the last statement; Rational Rhapsody adds one for you. Therefore, there is no need for any semicolon if there is only one statement.

The following example shows a transition label consisting of a trigger and an action sequence with more than one step:

```
e1/x=1;y=2 // comments are allowed
```

Action expressions must be syntactically legal in the programming language. This means they must be written within the context of the owner class (the one that owns the statechart being described). Therefore, all identifiers must be one of the following items:

- ◆ Class (or superclass) attributes or operations
- ◆ Role names
- ◆ Global variables known to the class

Any other identifier will cause failures at compile time.

Actions can reference the parameters of an event or operation as defined by the transition context, using the pseudo-variable `params->`. See also [Message parameters](#).

Initial connectors

An *initial connector* leads into the state (or substate of an Or state, or component of an And state) that should be entered by default. Each Or state must designate one of its substates as the default for that state. The default state is indicated using an initial connector, of which there can be only one per Or state. The initial connector target should be a substate of the Or state to which it belongs.

The initial connector cannot have a trigger or a guard, although it can have an action. It might connect to a decision node following which there might be guards.

Each state can have the following properties:

- ♦ **Entry action** where an expression executed upon entrance to the state (uninterpreted by Rational Rhapsody). Note that an uninterpreted expression is resolved by the compiler, not by the tool.
- ♦ **Exit action** where an expression executed upon exit from the state (uninterpreted by Rational Rhapsody).
- ♦ **Static reactions** where actions performed inside a state in response to a triggering event/operation. The reaction can be guarded by a condition. A static reaction does not change the active state configuration.

The state executes static reactions if:

- The state is part of the active configuration.
 - The trigger and guard are satisfied.
 - A lower-level state has not already responded to the trigger.
 - There is no active transition that causes the state to be exited.
- ♦ **Default entry**
 - ♦ **History**

Note

When a state contains an entry action, exit action, or static reaction, an icon  displays in the top-right corner of the state. This icon can be used to toggle the display of these actions in the state.

Events and operations

Events inherit from the `OMEvent` abstract class defined in the Rational Rhapsody framework. They are abstract entities that do not exist in C++ or other object-oriented programming languages. They are framework-based, and you can implement them in various ways.

In Rational Rhapsody, both events and messages create operations for a class. You can edit the operations created as a result of messages, but you cannot modify any event handlers.

Events and operations relate statecharts to the rest of the model by triggering transitions. Operations specified by a statechart are called triggered operations (as opposed to operations specified in OMDs, called primitive operations).

Events facilitate asynchronous collaborations and operations facilitate synchronous collaborations. Triggered operations have a return type and reply. Triggered operations have a higher priority than events.

In the rest of this guide, the term *message* means either an event or an operation.

Statecharts can react to operations and events that are part of the interface of a reactive class. Using a message as a trigger in a statechart to transition from state S1 to state S2 means that if the object is in S1 when it receives the message, it transitions to S2.

Events that do not trigger an active transition are ignored and discarded. If the object happens to be in state S3 when it receives the message and S3 does not reference the message, it ignores the message.

See [Events](#) for more information.

Sending events across address spaces

Rational Rhapsody allows you to send events to reactive instances in different address spaces.

This feature applies to multiple address spaces on the same computer. It is not possible to send events to reactive instances on a different computer.

This feature can be used with the following target environments:

- ♦ INTEGRITY5
- ♦ VxWorks6.2diab_RTP
- ♦ VxWorks6.2gnu_RTP

Note

Currently, the multiple address space feature applies only to Rational Rhapsody in C.

Use of this feature requires:

- ♦ Setting a number of properties
- ♦ Calling a different function than that used for sending events within the same address space

Properties for sending events across address spaces

To allow use of the multiple address space feature, different code generation settings are required. These settings are controlled by the following property:

- ♦ `C.CG::Configuration::MultipleAddressSpaces`
When this boolean property is set to Checked, Rational Rhapsody uses the code generation settings required for use of the multiple address space feature. *The default value of this property is Cleared, so you must change the value to enable this feature.*

In order to be able to receive events from other address spaces, the reactive object must publish the name by which it will be identified. The following two properties, set at the class level, are used for this purpose:

- ♦ `C.CG::Class::PublishedName`
This is the name that will be used to identify the reactive object in order to send a distributed event to it.
If there is only one reactive instance of the class, the value of this property is used to identify the object.
If there is more than one reactive instance of the class, each named explicitly, the name used to identify the reactive object will be the name that you have given to the object, and not the property value.
In the case of multiplicity, where the objects are not named explicitly, the name used to

identify the reactive object will be the published name + the index of the object, for example, if the value of the property `PublishedName` is `truck`, then the objects would be identified by `truck[0]`, `truck[1]`...

- ◆ `C.CG::Class::PublishInstance`
This boolean property indicates whether or not the object should be published as a reactive instance that is capable of receiving distributed events.

In addition, the following property, which is set at the configuration level, allows you to specify a specific target address space when sending events, as described in [API for sending events across address spaces](#):

- ◆ `C.CG::Configuration::AddressSpaceName`
When you want to send an event to a reactive object in a specific address space, you specify the address space by using the value of this property as a prefix, using the format *addressSpaceName::publishedNameOfReactiveObject*. The default value of this property is the name of the relevant component.

If the events to be sent across address spaces have no arguments or only primitive types as arguments, such as integers or chars, it is sufficient to just set the above properties. However, if the events to be sent include objects as arguments, you must also set the following properties at the event level:

- ◆ `C.CG::Event::SerializationFunction`
Name of user-provided serialization function to use
- ◆ `C.CG::Event::UnserializationFunction`
Name of user-provided unserialization function to use

For details regarding the required structure for these two user-provided functions, see [Functions for serialization/unserialization](#).

API for sending events across address spaces

When sending events to reactive objects in different address spaces, the function `RidSendRemoteEvent` must be used (and not the standard event generation macro `RiCGEN`):

```
RiCBoolean RidSendRemoteEvent (const RhpString strReactiveName, struct
RiCEvent* const ev, const RhpPositive eventSize);
```

`strReactiveName` - the published name of the destination reactive object

`ev` - pointer to the event to send

`eventSize` - the size of the event to send

Note

When providing the `strReactiveName` parameter for the function `RidSendRemoteEvent`, you can indicate which address space contains the target object,

using the format `addressSpaceName::publishedNameOfReactiveObject`. This allows you to have objects with the same name in multiple address spaces and still have the event sent to the appropriate object.

When using this option, the name you use for the address space is the value of the property `C.CG::Configuration::AddressSpaceName`, described in [Properties for sending events across address spaces](#).

For convenience, Rational Rhapsody includes a macro named `RIcGENREMOTE`, which calls the function `RidSendRemoteEvent`:

`RIcGENREMOTE` ([string - the published name of the destination reactive object], [type of event with parameters in parentheses])

For example:

```
RIcGENREMOTE("destinationObject", Fstarted());
```

Functions for serialization/unserialization

If the events to be sent across address spaces have no arguments or only primitive types as arguments, such as integers or chars, you just have to call the function `RidSendRemoteEvent`. However, if the events to be sent include objects as arguments, you must also provide two functions - one for serializing and one for unserializing the event arguments:

Serialization function

```
RhpAddress evStartSerialize(struct RiCEvent_t* const ev, const RhpAddress
buffer, RhpPositive bufferSize, RhpPositive* resultBufferSize);
```

return value - pointer to the serialized event

ev - pointer of the event to be serialized

buffer - a local buffer that can be used for storing the serialized event (the user can allocate their own buffer instead)

bufferSize - the size in bytes of the parameter buffer

resultBufferSize - pointer for storing the size of the returned serialized event

Unserialization function

```
RiCEvent_t* evStartUnserialize(RhpAddress const serializedBuffer, RhpPositive
serializedBufferSize);
```

return value - pointer to the unserialized event

serializedBuffer - pointer to the serialized buffer

serializedBufferSize - the size of the parameter serializedBuffer

Example of serialization/unserialization functions

The example refers to the event evStart, which is defined as follows:

```
struct evStart {
    RiCEvent ric_event;
    /**    User explicit entries    */
    char* msg;
};

RhpAddress evStartSerialize(struct RiCEvent* const ev, const RhpAddress
buffer, RhpPositive bufferSize, RhpPositive* resultBufferSize)
{
    evStart* castedEv = (evStart*)ev;
    RhpPositive msgLength = strlen(castedEv->msg);
    /* Testing the size of the message parameter against the size of local
buffer */
    if (bufferSize <= msgLength)
    {
        /* buffer too small - serialization is aborted */
        return NULL;
    }
    /* copy the message string + the null terminating */
    memcpy(buffer, castedEv->msg, msgLength + 1);
    *resultBufferSize = msgLength + 1;
    return buffer;
}
```

The function below uses a local buffer called receivedBuffer to store the string of the event evStart which was passed as a parameter.

```
RiCEvent* evStartUnserialize(RhpAddress const serializedBuffer,
RhpPositive serializedBufferSize) {
    /* copy the message to a local buffer */
    memcpy(receivedBuffer, serializedBuffer, serializedBufferSize);
    return (RiCEvent*)RiC_Create_evStart(receivedBuffer);
}
```

Send action elements

The Send Action button  can be used in statecharts, activity diagrams, and flow charts to represent the sending events to external entities.

The Send Action element can be used to specify the following actions:

- ◆ Event to send
- ◆ Event target
- ◆ Values for event arguments

This is a language-independent element, which is translated into the relevant implementation language during code generation.

Note

Code can be generated for Send Action elements in C, C++, and Java.

Defining send action elements

To define the element, provide the following information in the Features window:

- ◆ Using the *Target* list, select the object that is to receive the event.
- ◆ Using the *Event* list, select the event that should be sent.
- ◆ Provide values for the event arguments by selecting the argument in the argument list and clicking the *Value* column.

Note

In cases where there are a number of objects based on the same class, you need to provide additional information after selecting the target from the list. For cases of simple multiplicity, you must provide the array index to specify the object that receives the event. In the case of qualified associations, you need to provide the qualifier value for the object that is to receive the event.

The *Preview* text box displays the text that is displayed on the element if you select full notation as the display option to use.

The target list includes all objects known to the class for the statechart. You can choose the name of the target object, or the name of a port on the target object.

You can click the button next to the *Target* list to open the Features window of the relationship with the target object. Similarly, you can click the button next to the *Event* list to open the Features window for the selected event.

Display options for send actions

The display options for the Send Action element allow you to display a full notation, such as “Reset (false) to p1” or a short notation, such as “Reset.” Full notation includes the event name, the values for the event arguments, and the name of the target. Short notation includes only the event name.

Graphical behavior of send actions

In terms of its behavior in the graphic editors, Send Action elements are connected to states in the statechart with transitions.

While the graphical behavior of Send Action elements is similar to that of states, it should be remembered that semantically these elements are not states. For example, you cannot put a condition on the transition out of a Send Action element (it is an automatic transition).

Code generation for send actions

Code can be generated for Send Action elements in C, C++, and Java.

For each language, code generation for this element is determined by the following properties

- ◆ `CG::Framework::EventGenerationPattern` - general format
- ◆ `CG::Framework::EventToPortGenerationPattern` - used when sending event to a port

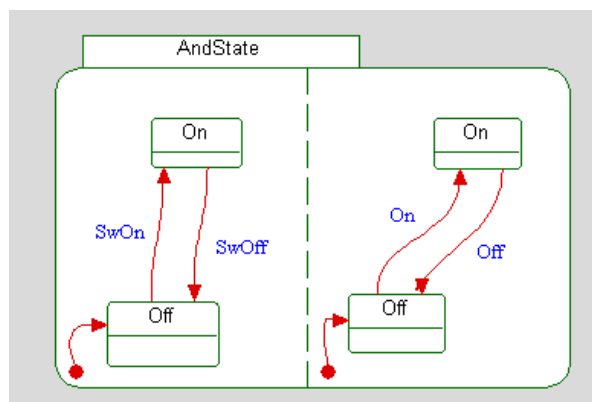
Note

Rational Rhapsody does not support roundtripping for Send Action elements.

And lines

An And line is a dotted line that separates the orthogonal components of an And state. There can be two or more orthogonal components in a given And state and each behaves independently of the others. If the system is in an And state, it is also simultaneously in a substate of each orthogonal component.

The following figure shows an And line.



Drawing And lines

To draw an And line to divide a state into substates:

1. Click the **And line** button  in the **Diagram Tools**.
2. Click in the middle of the upper edge of the state to anchor the start of the And line.
3. Move the cursor down to the bottom edge of the state and click to anchor the end of the And line. Rational Rhapsody draws a dotted line that divides the state into two halves (orthogonal states), as shown in the following figure.

Note that the state label, which used to be inside the state, has moved outside into a tab-like rectangle.

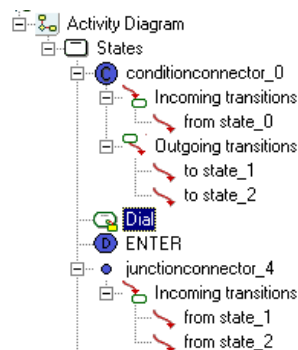
Connectors

Rational Rhapsody supports the following connectors:

- ◆ [Decision nodes](#)
- ◆ [History connectors](#)
- ◆ [Merge nodes](#)
- ◆ [Diagram connectors](#)
- ◆ [Termination connectors](#)
- ◆ [EnterExit points](#)

Rational Rhapsody includes connector information for diagram, condition, and EnterExit points in its repository (core). This means that:

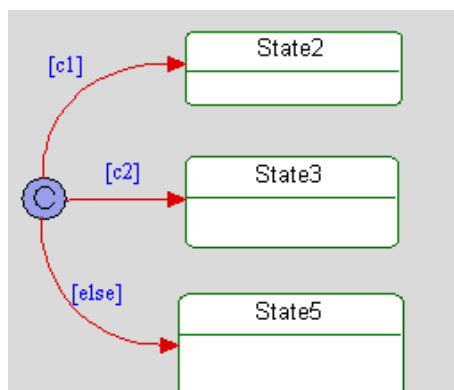
- ◆ Semantic checks are done by the standard core functions.
- ◆ Statechart inheritance is core-oriented, not graphics-oriented.
- ◆ Code is generated for these connectors.
- ◆ The Undo operation supports all connector actions.
- ◆ Rational Rhapsody EnterExit points are now UML-compliant.
- ◆ Reports include information on diagram connectors, decision nodes, and EnterExit points.
- ◆ Diagram connectors, decision nodes, and EnterExit points are displayed in the browser, as shown in the following figure.



Decision nodes

Decision Nodes split a single segment into several branches. Branches are labeled with guards that determine which branch is active.

The following figure shows a decision node.



The following rules apply to decision nodes and branches:

- ◆ Branches cannot contain triggers.
- ◆ You can nest branching segments. This means that a branching segment can enter another decision node.
- ◆ A decision node can have only one entering transition.
- ◆ The branching tree should not have cycles.

Else branches

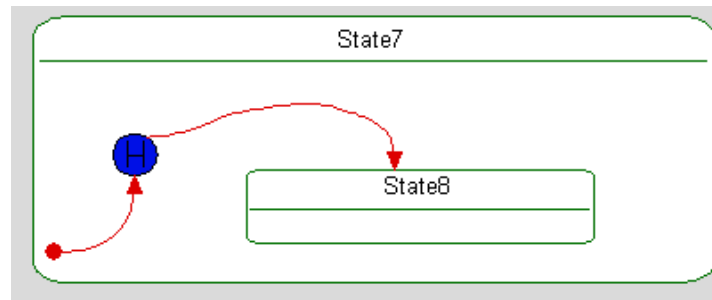
A guard called `[else]` is active if all the guards on the other branches are false. Each decision node can have only one else branch.

The semantics of an else branch are similar to a structured if-then-else statement.

History connectors

History connectors store the most recent active configuration of a state and its substates. Once an object is created, it is associated with a configuration for an active state, starting in the initial configuration, and evolving as the statechart responds to messages.

The following figure shows a history connector.



When a transition is attached to a history connector and that transition is triggered, the state containing the history connector recalls its last active configuration. A state can have a single history connector.

Transitions from a history connector are constrained to a destination on the same level as the history connector.

Note

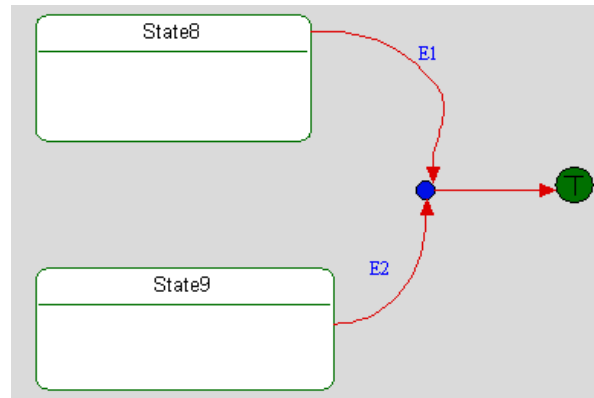
Do not put more than one history connector in a state. Rational Rhapsody allows you to draw more than one history connector in a state; however, the code generator does not support this.

A state might have a history property used for recalling the recent active configuration of the state and its substates. Transitioning into a history connector associated with the state recalls the last active configuration.

A transition originating from the history connector designates the history default state. The default history state is taken if no history existed prior to the history enter.

Merge nodes

A *merge node* combines several segments into one outgoing segment, as shown in the following figure.



This means that segments share the same line and a common transition suffix. The segments end up sharing the same transition line.

Diagram connectors

A *diagram connector* functions similarly to a merge node in that it joins several segments in the same statechart. Diagram connectors enable you to jump to different parts of a diagram without drawing spaghetti transitions. This helps avoid cluttering the statechart. The jump is defined by matching names on the source and target diagram connectors.

Note

You can rename diagram connectors, and the checks are performed during code generation.

Diagram connectors should either have input transitions or a single outgoing transition. A statechart can have at most one target diagram connector of each label, but it can have several source diagram connectors with the same label.

During code generation, Rational Rhapsody flattens all junctions and diagram connectors by merging the common suffix to each segment entering the connector.

In both diagram and merge nodes, a label that belongs to an incoming segment is shared and duplicated during code generation among outgoing segments of that connector. Rational Rhapsody merges the guards (conjunction), then concatenates the actions.

Note

Both incoming and outgoing transitions cannot have labels. If you label the incoming transitions, do not label the outgoing transition because its label will override the label of the incoming transition and negate any action or trigger associated with the incoming transition.

Diagram connectors connect different sections of the same statechart, whereas EnterExit points connect different statecharts. See [EnterExit points](#).

Termination connectors

The *termination connector* is the suicide or self-destruct connector. If a transition to a termination connector is taken, the instance deletes itself. A termination connector cannot have an outgoing transition.

EnterExit points

EnterExit points are used to represent the entry to / exit from sub-statecharts.

At the level of the parent state, these points represent entry to / exit from the various contained substates without revealing any information about the specific substate that the transition connects to.

At the level of the sub-statechart, these points represent the entries to / exits from the parent state vis-a-vis the other elements in the statechart.

When you create a sub-statechart from a parent that contains deep transitions (that is, transitions entering one of the substates), EnterExit points are automatically created on the borders of the parent state in both the sub-statechart and the original statechart.

Once the sub-statechart has been created, you can add additional deep transitions as follows:

1. Add an EnterExit point to the parent state.
2. Add a corresponding EnterExit point in the sub-statechart (manually or using the Update feature - see [Updating EnterExit points](#)).
3. Draw a transition from the EnterExit point to the relevant substate.

Updating EnterExit points

If you would like to add additional EnterExit points after creating a sub-statechart, you can add them manually to both the parent state in the original statechart and the parent state in the sub-statechart.

Alternatively, you can have Rational Rhapsody automatically update the EnterExit points:

1. Add one or more EnterExit points to the parent state in either the original statechart or the sub-statechart.
2. Go to the second statechart, right-click and select **Update EnterExit Points**.

Submachines

Submachines enable you to manage the complexity of large statecharts by decomposition. The original statechart is called the *parent*, whereas the decomposed part is called the *submachine*.

Creating a submachine

You can create a submachine from a complex state using either the Edit menu or the menu for the state.

To create a submachine, on a statechart, right-click a state and then select **Create Sub-Statechart**. Rational Rhapsody creates a submachine called `<class>.<state>`, which is a new statechart consisting of the submachine state and its contents.

If you decompose the `doorClosed` state into a submachine, Rational Rhapsody creates a new submachine.

Actions and reactions move into the top state of the submachine if the transition goes to the submachine state, and inside the submachine if the transition goes into the nested part.

Note

You cannot create submachines of inherited states. The workaround is to add a dummy state as a child of the inherited state and make that the submachine state.

Opening a submachine or parent statechart

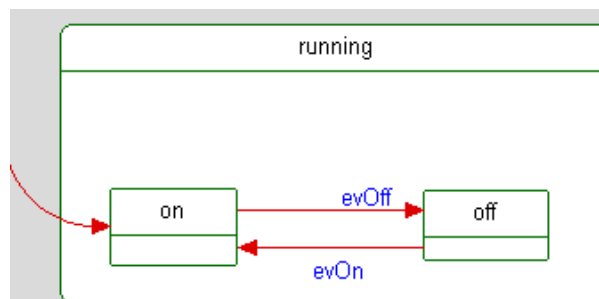
To open a submachine, right-click the submachine state in the parent statechart and select **Open Sub-Statechart**.

Similarly, to open a parent statechart from a submachine, right-click the top state and select **Open Parent Statechart**.

Deep transitions

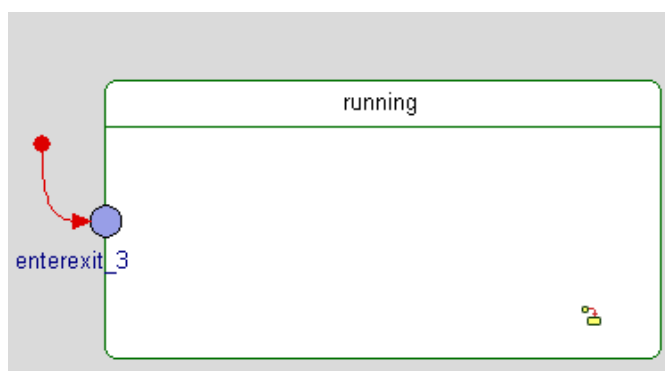
A *deep transition* is a cross-chart transition, for example, from a parent statechart into a submachine, or vice versa. When you create a submachine, deep transitions are automatically split via substates.

Consider the following example:



This statechart has a deep transition that crosses the edge of a parent state (running) and leads into a nested state (on).

If you make a submachine of the running state, the deep transition is automatically split via matching EnterExit points created in the parent statechart and submachine, as shown in this example.



Merging a sub-statechart into its parent statechart

To merge the contents of a sub-statechart into its parent statechart:

1. Select the parent state in the original statechart.
2. Right-click the state and select **Merge Back Sub-Statechart**.

Statechart semantics

The following sections describe the object-oriented interpretation of statecharts.

Single message run-to-completion processing

Rational Rhapsody assumes that statecharts react to a single message applied by some external actor to the statechart. The external actor can be either the system event queue or another object.

Message processing by a statechart is partitioned into steps. In each step, a message is dispatched to the statechart for processing.

Once a message is dispatched, it might enable transitions triggered by the message. Each orthogonal component can fire one transition at most as a result of the message dispatch. Conflicting transitions will not fire in the same step.

The order in which selected transitions fire is not defined. It is based on an arbitrary traversal that is not explicitly defined by the statechart.

Each component can execute one transition as a result of the message. Once all components complete executing the transition, the message is said to be consumed, and the step terminates.

After reacting to a message, the statechart might reach a state configuration in which some of the states have outgoing, active null transitions (transient configurations). In this case, further steps need to be taken until the statechart reaches a stable state configuration (no more transitions are active). Null transitions are triggered by null events, which are dispatched to the statechart whenever a transient-configuration is encountered. Null events are dispatched in a series of steps until a stable configuration is reached. Once a stable configuration is reached, the reaction to the message is completed, control returns to the dispatcher, and new messages can be dispatched.

Note

Theoretically, it is possible that the statechart will never reach a stable configuration. The practical solution is to set a limit to the maximum number of steps allowed for a statechart to reach a stable configuration. In the current implementation, reaching the maximum number of steps is treated as if the message processing has been completed.

Active transitions

A transition is active if:

- ♦ The trigger matches the message posted to the statechart. (Null triggers match the null event.)
- ♦ There is a path from the source to the target states where all the guards are satisfied (evaluate to true).

Note

Guards are evaluated before invoking any action related to the transition.

Because guards are not interpreted, their evaluation might include expressions that cause side effects. Avoid creating guards that might cause side effects. Guard evaluation strategy is intentionally undefined as to when guards are evaluated and in which order.

Transition selection

Transition selection specifies which subset of active transitions should fire. Two factors are considered:

- ♦ Conflicts
- ♦ Priorities

Conflicts

Two transitions are said to *conflict* if both cause the same state to exit. Only orthogonal or independent transitions fire simultaneously. This means that interleaved execution causes equivalent results. Disjoint exit states are a satisfactory condition for equivalent results.

Note: With regard to conflicts, static reactions are treated as transitions that exit and enter the state on which they are defined.

Priorities

Priorities resolve some, but not all, transition conflicts. Rational Rhapsody uses state hierarchies to define priorities among conflicting transitions. However, lower-level (nested) states can override behaviors, thus implying higher priority.

The priority for a transition is based on its source state. Priorities are assigned to join transitions based on their lower source state.

For example, if transition t1 has a source state of s1 and transition t2 has a source state of s2,:

- ♦ If state s1 is a descendant of state s2, t1 has a higher priority than t2.

- ♦ If states s_1 and s_2 are not hierarchically related, relative priorities between t_1 and t_2 are undefined.

Rational Rhapsody does not define a priority with regard to events and transitions other than arrival order. If two transitions within the same orthogonal component are both active (ready to fire), as can happen with non-orthogonal guards, only one of them will actually fire, but statecharts do not specify which one it will be.

Transition selection algorithm

The set of transitions to fire satisfies the following conditions:

- ♦ All transitions must be active.
- ♦ Any transition without conflicts will fire.
- ♦ If a priority is defined between transitions, the transitions with lower priority will not fire.
- ♦ In any set of conflicting transitions, one transition is selected to fire. In cases where conflicts are not resolved by priorities, the selected transition is arbitrary.

The above definition of the selection set is not imperative, but implementing a selection algorithm is done by a straightforward traversal of the active state configuration.

Active states are traversed bottom-up where transitions related to each are evaluated. This traversal guarantees that the priority principle is not violated. The only issue is resolving transition conflicts across orthogonal states. This is solved by “locking” each And state once a transition is fired inside one of its components. The bottom-up traversal and the And state locking together guarantee a proper selection set.

Transition execution

Once a transition is active and selected to fire, there is an implied action sequencing:

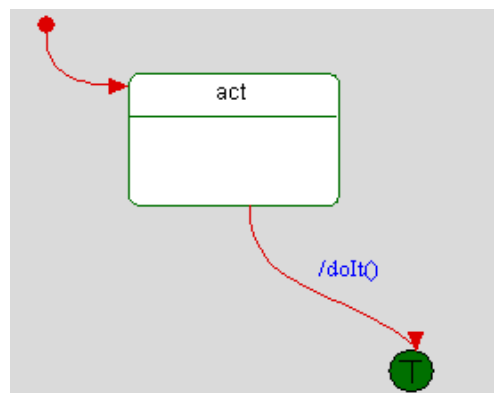
- ♦ States are exited and their exit actions are executed, where deeper states are exited before their parent states. In case of orthogonal components, the order among orthogonal siblings is undetermined.
- ♦ Actions sequencing follow the direction of the transition. The closer the action to the source state, the earlier it is evaluated.
- ♦ Target states are entered and their entry actions are executed, where parent states are entered before substates. In the case of orthogonal components, the entry order is undetermined.

Active classes without statecharts

Normally, active classes (threads) must also be reactive (have statecharts). However, you might have tasks that have no state memory. The workaround of defining a dummy (empty) statechart is not entirely acceptable because such an active object uses statechart behavior to process events. It is, however, possible to achieve the same effect by setting the class to active, defining an empty statechart, then overriding the default behavior by defining an operation named `takeEvent()` for the class and adding the wanted behavior to this operation. This method allows you to benefit from visual debugging, using the event queue, and so on.

Single-action statecharts

Rational Rhapsody cannot interpret simple statecharts that execute a single action and then terminate. For example, if you represent a task as an active class with a simple statechart that essentially executes a single action and terminates, you might be tempted to draw your statechart, as shown in this example.



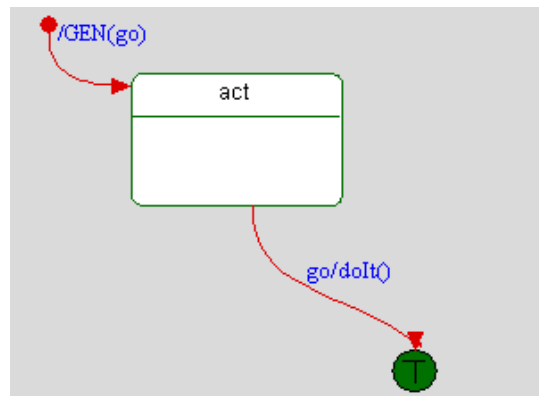
In this diagram, `doIt()` represents the action that needs to be spawned.

This statechart has two problems:

- ♦ The Rational Rhapsody framework does not allow an active instance to terminate on the initial run-to-completion (RTC) step. In other words, the `startBehavior()` call cannot end with a destroyed object.
- ♦ The `startBehavior()` call executes the initial step on the creator thread, not as part of the instance thread. The instance thread processes events following the initial step. In this statechart, the `doIt()` operation is executed on the creator thread, which is probably not what was expected.

The workaround is to create a dummy action on the initial connector that leads into a transition. This action can run on the instance thread and thus terminate normally.

For example, the following statechart postpones the execution of the action until the thread is ready to process it.



Inherited statecharts

Statechart inheritance begins when a class derives from a superclass that has a statechart. The statechart of the subclass, the inherited statechart, is initially a clone of that of the superclass. With the exception of the items listed below, you can add things to an inherited statechart to override behavior in the inherited class.

You cannot make the following changes to items in the statechart of a subclass:

- ♦ Change the source of a transition.
- ♦ Change the triggers (events or triggered operations).
- ♦ Delete or rename a state.
- ♦ Draw a state around an existing state.

You can make the following changes to items in the statechart of a subclass:

- ♦ Change anything that does not affect the model, such as moving things in the diagram without actually editing.
- ♦ Add objects to a state.
- ♦ Add more states, but not re-parent states.
- ♦ Attach a transition to a different target.

An inherited statechart consists of all the items inherited from the superclass, as well as modified and added elements.

Note

It is possible to inherit statecharts across packages.

If you edit a base statechart, the derived statechart is redrawn only on demand at checks, code generation, report generation, or the opening of a derived statechart.

Types of inheritance

Each item in the derived statechart can be:

- ♦ **Inherited** where any modifications to an item in the superclass is applied to the item in the subclass.
- ♦ **Overridden** where any modifications to an item in the superclass do not apply to the subclass. However, deleting an item from the superclass also deletes the item from the subclass. This is different from C++, for example, where deleting an overridden behavior in the superclass causes the overridden behavior to become a regular item.
- ♦ **Regular** where regular items are owned by the subclass. The item is not related to the superclass and is not affected by the superclass.

Noting the status of items as inherited, overridden, or regular is crucial both for Rational Rhapsody and the user.

Note

The current implementation of statechart inheritance is restricted to single inheritance. A reactive class can have at most one reactive superclass.

Inheritance color coding

Inheritance status is indicated by the following color coding:

- ♦ Inherited items are gray.
- ♦ Regular and overridden items are colored in the usual drawing colors.

Inheritance rules

Classes with inherited statecharts can reside in different packages. A class with a statechart (reactive class) can inherit from a class without a statechart. Multiple inheritance of reactive classes (with statecharts) is not supported. Derived classes can inherit from multiple primitive classes. Rearranging inheritance hierarchies of reactive classes is not supported.

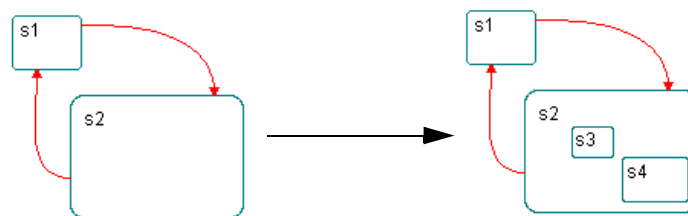
There are different inheritance rules for states, transitions, triggers, guards, and actions.

Rules for states

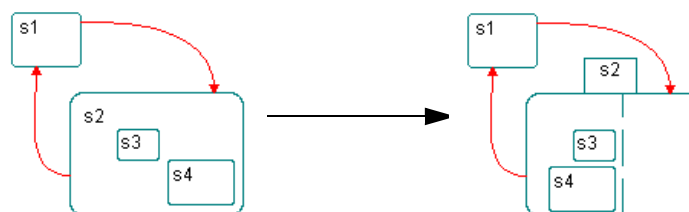
The structure of a state in a subclass should be a refinement of the same state of the superclass. Because of this, state inheritance is strict. All states and their hierarchy are inherited by the statechart of the subclass.

You can add states to the derived statechart, as long as they do not violate the hierarchy in the statechart of the superclass. In practice, this means that a regular state cannot contain inherited substates.

In the following example, the leaf state s2 was refined and became an Or state. The states s1 and s2 on the right are the inherited states.



You can add And lines to inherited states (adding components). If you convert an inherited Or state into an And state, the Or state becomes an And state, and one of the components contains its substates. This is an exception to the previous rule, where the state hierarchy is modified by introducing an orthogonal component. The component that re-parents the substates is designated as “main.” In the following example, s2 becomes an And state. The component containing s3 and s4 is the main component. The name of a component is the same as the name for the And state.



Note the following actions:

- ◆ You cannot rename inherited states in the derived statechart.
- ◆ You cannot delete inherited states from a derived statechart.
- ◆ A state is either inherited or regular. It cannot be overridden.
- ◆ You cannot change state topology by re-parenting.

Rules for transition labels

You can modify the labels of derived segments according to the following rules:

- ◆ You cannot modify triggers. They are inherited from the superclass.
- ◆ You can modify actions and guards.
- ◆ You can override a guard, but still get changes on the action.

Modifications to the label of the corresponding segment in the superclass no longer affect the subclass.

Note

The inheritance color coding of the label and the segment are independent. The label can be overridden while the segment is still inherited, and vice versa.

Rules for entry and exit actions

Both entry and exit actions can be overridden by an inherited state. Once they have been modified (modifying the text of the action) by a derived state, they reach an overridden state.

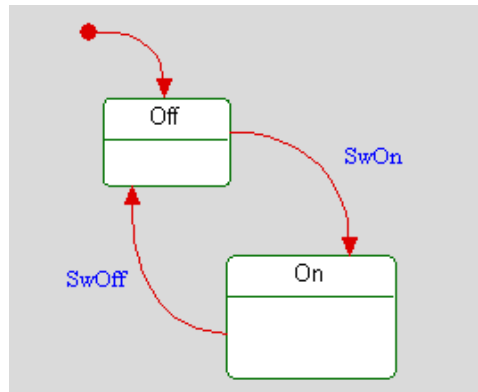
Rules for static reactions

Static reactions can be overridden by a derived statechart. Static reactions are designated by their triggers, which cannot be modified in a derived statechart. Therefore, only the guard and action can be modified. A static reaction cannot be deleted by a subclass.

Currently, there is no inheritance color coding for static reactions. In addition, tracing inherited actions between the superclass and the derived statechart is done implicitly by Rational Rhapsody and is not visible.

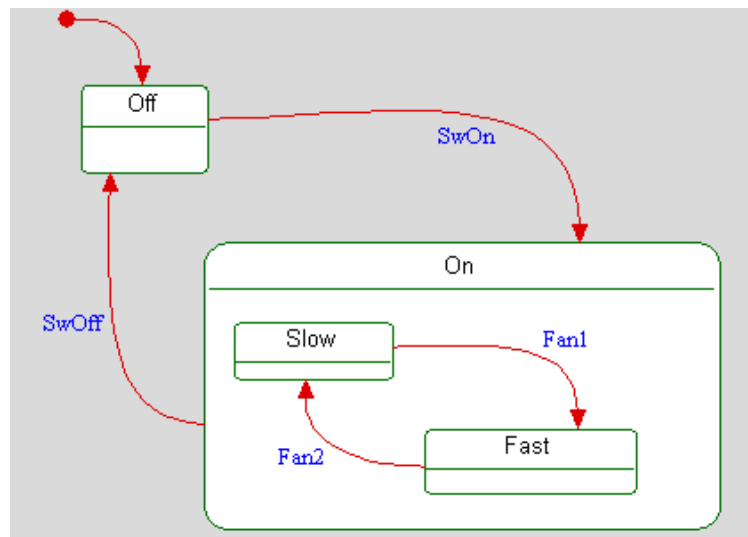
Rules for connectors

Connectors are always inherited. You cannot modify them or delete them. The following example illustrates statechart inheritance.

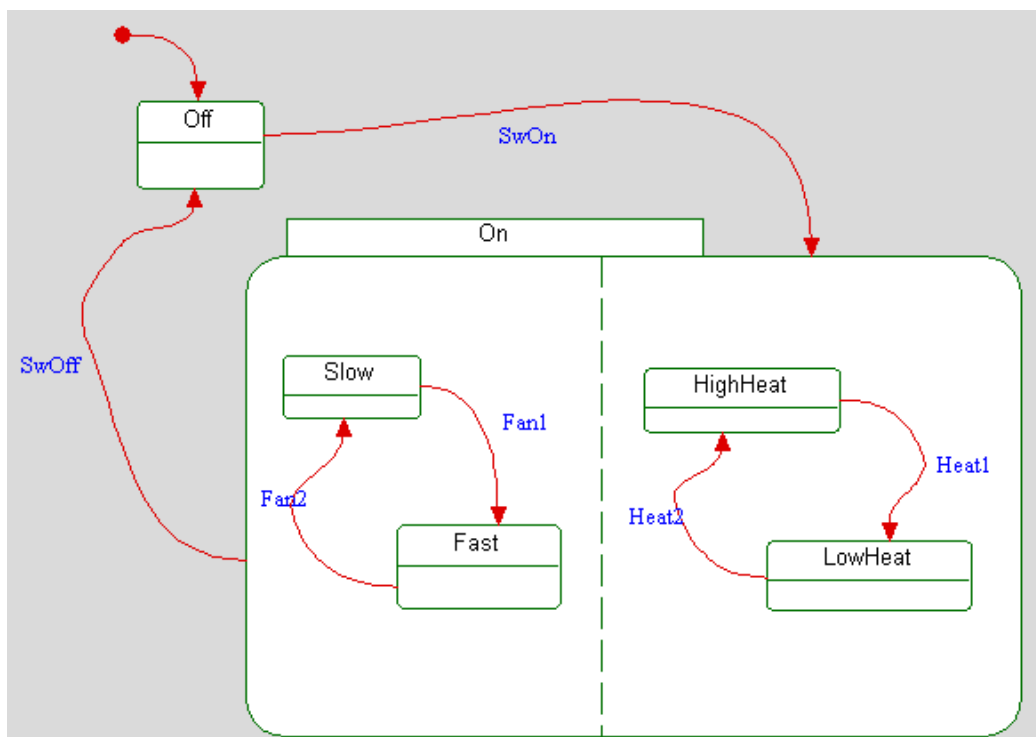


As shown, a basic blower has only On and Off modes.

In a dual-speed blower, the On state is refined to include Fast and Slow modes, as shown in the following figure.



If you make the On state into an And state, you can add a heat mode, as shown.



Overriding inheritance rules

To override the inheritance rules of statecharts, right-click in the statechart and select **Override Inheritance**.

Once you have overridden inheritance, the derived statechart becomes independent from its parent and you can modify it without constraint. In addition, colors are no longer gray. They are the usual statechart colors.

To undo the inheritance override, right-click in the statechart and select **Unoverride Inheritance**.

Overriding textual information

The Features window for textual information in statecharts (state entry and exit actions, and guards and actions for transitions and static reactions) include the **Overridden** check box. By enabling or disabling this check box, you can easily override and unoverride statechart inheritance without actually changing the model. As you toggle the check box on and off, you can view the inherited information in each of the window fields, and can decide whether to apply the information or revert back to the currently overridden information.

To apply the change, click **OK**. The transition changes to `doServe(params->rounds)` and it is displayed in blue text instead of gray because it is no longer overridden.

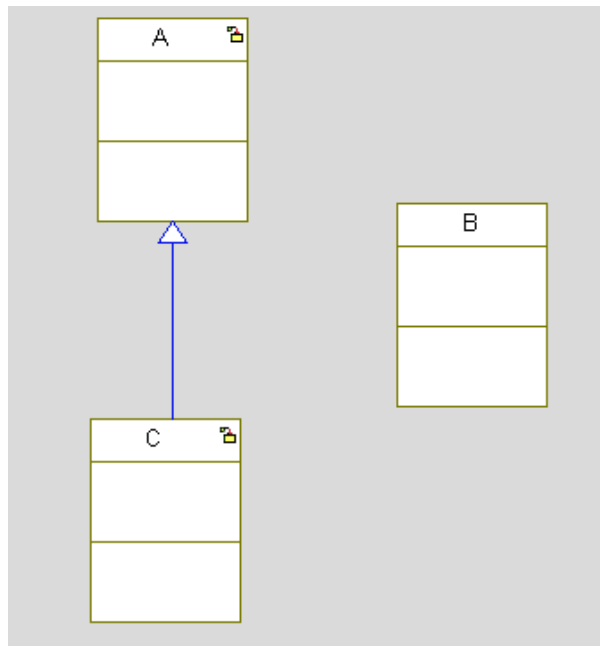
Note that if you override the textual information, the display colors and statechart change as follows:

- ♦ If you unoverride the textual information of a transition or state, the label color reverts to gray.
- ♦ If you unoverride a transition target, the transition color reverts to gray and the graphics are synchronized to the new target.

Refining the hierarchy of reactive classes

You can refine the hierarchy of reactive classes without using overrides and unoverrides (without losing any information).

For example, suppose you have class C inheriting from class A, as shown in the following OMD.



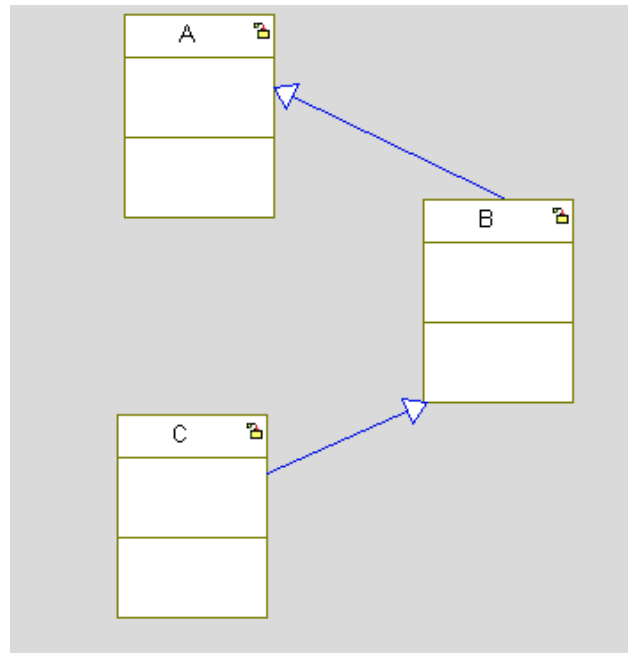
Suppose you want to change the hierarchy so C inherits from B, which in turn inherits from A. Therefore:

- ♦ The statechart that C inherited from A will now be inherited from B; B will inherit its statechart from A.
- ♦ The inheritance between A and C will be deleted.
- ♦ C will not lose any information, because its inherited elements will reference new GUIs.

To make these changes:

1. Using either the browser or the **Diagram Tools**, create inheritance between B and A.
2. Create inheritance between C and B.
3. Rational Rhapsody displays a window that informs you that you are adding a level of inheritance, and asks you to confirm the deletion of the inheritance between C and A. Click **Yes**.

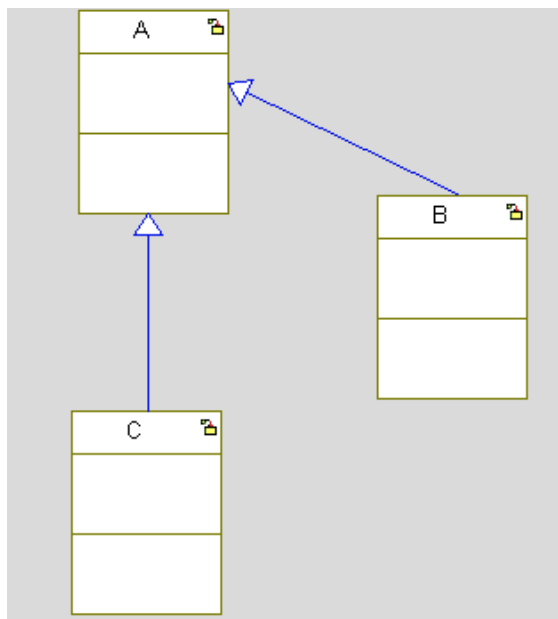
The following figure shows the revised OMD.



Removing a level of inheritance

Suppose you now want C to inherit from A instead of B, thereby removing a level of inheritance. When you draw the inheritance between C and A, Rational Rhapsody notifies you that a level of inheritance will be removed and asks for confirmation.

Click **Yes**. The following figure shows the revised OMD.



Inheritance between two reactive classes

If you try to establish inheritance between two distinct reactive classes (for example, B inherits from A), Rational Rhapsody displays a message stating that the action will result in overriding statechart inheritance. Click **Yes** to override the statechart inheritance.

IS_IN Query

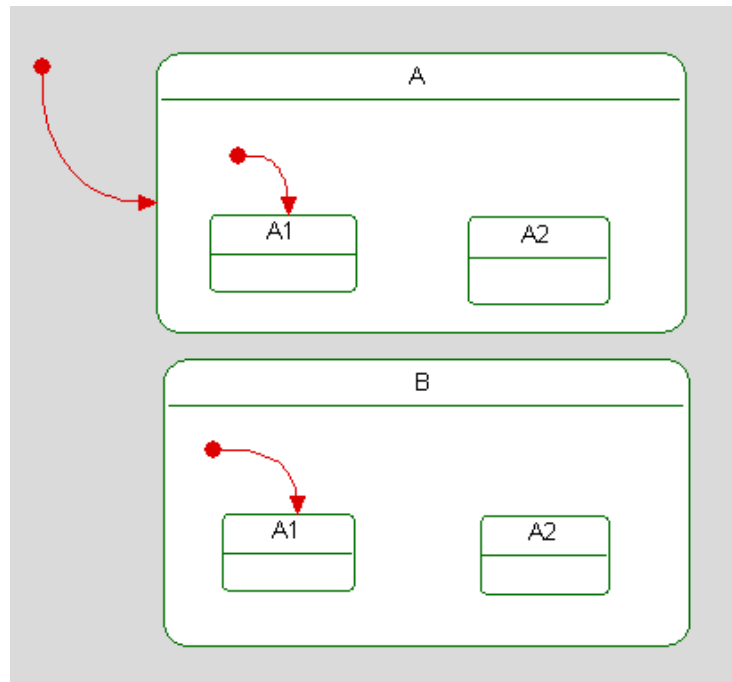
The Rational Rhapsody framework provides the query function `IS_IN(state)`, which returns a true value if the state is in the current active configuration.

Note the following information:

- ♦ `IS_IN(state)` returns true if the state is in the active configuration at the beginning of the step. For states entered in a step, `IS_IN(state)` returns false, unless the states are being re-entered.
- ♦ The state name passed to `IS_IN()` is the implementation name, which might be different from the state name if it is not unique.

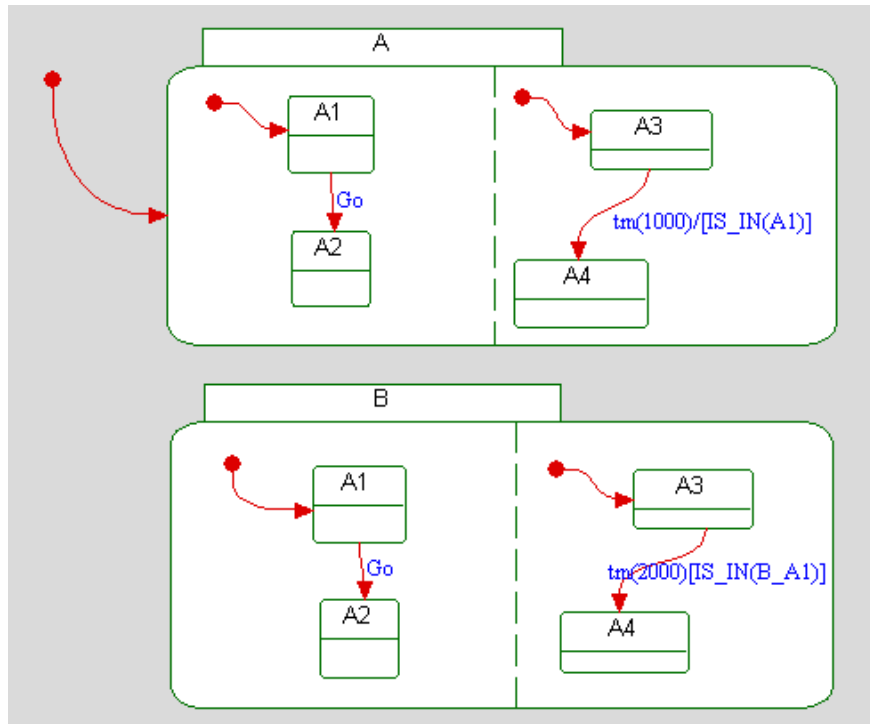
For example, the following state names are generated for the statechart shown in the figure:

```
State* B; State* B_A2; State* B_A1;  
State* A; State* A2; State* A1;
```



The implementation name of A1 in state A is simply A1 because it was drawn first. The implementation name of A1 in state B is B_A1, because it is a duplicate.

In the following statechart, the transition to substate A4 in state A is taken only if the object is still in substate A1 in A after one second. The transition to substate A4 in state B is taken only if the object is still in substate A1 in B after two seconds.



In these two cases, the `IS_IN()` query requires the use of the implementation names `A1` and `B_A1` to differentiate between like-named substates of two different states.

The `IS_IN` macro is called as if it were a member operation of a class. If you want to test the state of another class (for example, a relation), you must use the relation name. For example, if you have a relation to a class `A` called `itsA` and you want to see if `A` is in the `idle` state, you would use `itsA->IS_IN(idle)` rather than `A->IS_IN(idle)`.

Message parameters

Message data are formal parameters used within the transition context. By default, if the message is an event, the names of message parameters are the same as the arguments (data members) of the event class.

You reference event arguments in a statechart using the pseudo-variable `params->` with the following syntax:

```
event/params->event_arg1, params->event_arg2
```

Consider a class `Firecracker` that processes an event `discharge`, which has an argument `color`. The argument `color` is of an enumerated type `Colors`, with the possible values `red` (0), `green` (1), or `blue` (2). In the statechart, you would indicate that you want to pass a color to the event when it occurs using the following label on the transition:

```
discharge/params->color
```

When you run the application with animation, you can generate a `discharge` event and pass it the value `red` by typing the following command in the animation command field:

```
Firecracker[0]->GEN(discharge(red))
```

The application understands `red` as a value being passed to the argument `color` of event `discharge` because of the notation `params->color`. The color `red` is translated to its integer value (0), and the event is entered on the event queue of the main thread as follows:

```
Firecracker[0]->discharge((int)color = 0)
```

Finally, the event queue processes the event `discharge` with the value `red` passed via `params->`. The `Firecracker` explodes in red and transitions from the `ready` to the `discharged` state.

The way the `params->` mechanism works is as follows: When you create an event and give it arguments, Rational Rhapsody generates an event class (derived from `OMEvent`) with the arguments as its attributes. Code for events is generated in the package file.

The following sample code shows the event discharge, which has one argument called color. The code was generated in the header file for the Default package:

```
//-----
// Default.h
//-----
class discharge;
class Firecracker;
enum Colors {red, green, blue};
class discharge : public OMEvent {
    DECLARE_META_EVENT
    //// User explicit entries    ////
public :
    Colors color;
    //// User implicit entries    ////
public :
    // Constructors and destructors:
    discharge();
    //// Framework entries    ////
public :
    discharge(Colors p_color);
    // This constructor is need in code instrumentation
    discharge(int p_color);
};
```

When the Firecracker event queue is ready to take the event discharge, it calls SETPARAMS(discharge). SETPARAMS is a macro defined in oxf\state.h as follows:

```
#define SETPARAMS(type) type *params; params=(type*)event
```

Calling SETPARAMS(discharge) allocates a variable params of type pointer to an event of type discharge. This enables you to use params->color in the action part of the transition as a short-hand notation for discharge->color.

Modeling of continuous time behavior

There are three types of behaviors typical of embedded systems:

- ♦ **Simple** means implemented in functions and operations
- ♦ **Continuous** means expressed by items such as PID control loops or digital filters
- ♦ **Reactive** means state-based behavior

Although the Rational Rhapsody GUI directly supports only simple and reactive behaviors, you can implement all three types. To address the continuous time aspects, you can reference or include any code you are currently using to express continuous behavior in any of the operations defined within Rational Rhapsody.

This means that if you are defining your continuous behavior elements manually, you can continue to do so. If you are using another tool to define your continuous behavior and that tool generates code, you can include that code.

Interrupt handlers

The ability to add an interrupt handler depends on operating system support. Typically, a static function without parameters is added by passing its address to an operating system operation like `InstallIntHdlr` (operating system-dependent). The static function can be either a special singleton object or a function defined within a package. This operation must use compiler-specific utilities to get to the registers. Eventually, it must return and execute a return from the interrupt instruction.

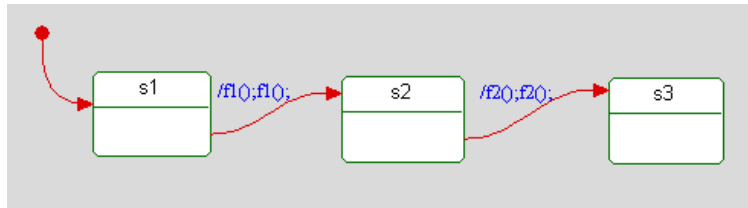
You can pass the data from the interrupt handler to the CPU (assuming that the interrupt handler needs to), in the following ways:

- ♦ Generate an event (using the `GEN()` macro), which then goes via the operating system to the reactive object (which should be in a different thread).
- ♦ Use a rendezvous object with a read/write toggle lock. The interrupt handler checks whether the lock is in the write state, then updates the data and puts the lock in the read state. The reader (in another thread) periodically checks the lock and only reads when it is in the read state. If it is in that state, the reader reads the data and updates the lock to its write state. This can easily be extended to a queue structure.
- ♦ Write the interrupt handler manually outside Rational Rhapsody and send it to the operating system message queue for the target thread. Typically, if the operating system does not support interrupt handlers directly, you store the current handler in your function and write the address of the function in the appropriate place in the interrupt vector table. When the interrupt goes off, either the new function replaces the old interrupt handler (meaning when it is done it simply returns) or it chains to it (calls the original). In any

event, when the interrupt handler is decommissioned, you replace the vector in the vector table with the original address.

Inlining of statechart code

Consider the following statechart:



When you implement this statechart using the flat scheme, you should expect the following methods to be added to the class and called through a series of a few function calls from `rootState_dispatchEvent()`:

```
int MyClass::s2TakeNull() {
    int res = eventNotConsumed;
    s2_exit();
    //#[ transition 2
    f2();
    f2();
    //#]
    s3_entDef();
    res = eventConsumed;
    return res;
};

int MyClass::s1TakeNull() {
    int res = eventNotConsumed;
    s1_exit();
    //#[ transition 1
    f1();
    f1();
    //#]
    s2_entDef();
    res = eventConsumed;
    return res;
};
```

In fact, what happens is that the transition code (between the `//#[` and `//#]` brackets) is immediately inlined (embedded) inside `rootState_dispatchEvent()`, as follows:

```
int MyClass::rootState_dispatchEvent(short id) {
    int res = eventNotConsumed;
    switch(rootState_active) {
        case s1:
        {
            if(id == Null_id)
            {
                popNullConfig();
            }
        }
    }
}
```

```
        //#[ transition 1
        f1();
        f1();
        //#]
        pushNullConfig();
        rootState_subState = s2;
        rootState_active = s2;
        res = eventConsumed;
    }
    break;
};
case s2:
{
    if(id == Null_id)
    {
        popNullConfig();
        //#[ transition 2
        f2();
        f2();
        //#]
        rootState_subState = s3;
        rootState_active = s3;
        res = eventConsumed;
    }
    break;
};
};
return res;
};
```

This code is more efficient because it saves a few function calls via inlining of code. Inlining is available using the `CG::Class::ComplexityForInlining` property of the class, which is set to 3 by default. This means that if the user code (the action part of the transition) is shorter than three lines, it is inlined (or embedded) where the function call used to be instead of the function call. To get the “expected” result (not inlined), set this property to 0.

Tabular statecharts

In addition to viewing statecharts as diagrams, it is possible to view statecharts in a tabular format. You can also make certain types of changes to your statechart when using the tabular view.

The property `StatechartDiagram::StateDiagram::DefaultView` can be used to determine the default view for statecharts - diagram view or tabular view. This property can be set at the level of individual statecharts or higher.

Format of statechart tables

When Rational Rhapsody displays a statechart as a table:

- ◆ The rows of the table represent the various states.
- ◆ The columns of the table represent the triggers that lead to transitions between states.

- ◆ Table cells display the new state the application will enter when the relevant trigger occurs.
- ◆ The table contains a column named *Null* which is used for transitions not associated with a specific trigger.
- ◆ For an And state, each of the substates is listed in its own row, nested below the And state.
- ◆ Diagram connectors do not appear in the table. Rather, the resulting transition between states is shown.
- ◆ The following statechart elements appear as rows in the table: condition connectors, history connectors, junction connectors, join bars and fork bars.
- ◆ Enter/Exit points appear as rows in the table, nested under their owner state.
- ◆ Default connectors (transitions) are depicted as outgoing transitions from a state called *ROOT*.

Modifying statecharts from tabular view

You can make the following types of changes to statecharts when using the tabular view:

- ◆ Add transitions
- ◆ Add events
- ◆ Delete transitions
- ◆ Delete events
- ◆ Delete states

Adding a transition

To add a new transition to the table:

Select the name of the target state and drag it to the table cell where the row of the source state intersects the column of the relevant trigger.

To add a transition not associated with a specific trigger:

Select the name of the target state and drag it to the table cell where the row of the source state intersects the *NULL* column.

To add a default connector (transition) for the statechart:

Select the name of the target state and drag it to the table cell where the *ROOT* row intersects the *Initial* column.

Adding an event

To add an event when in tabular view, click the **Add model element** button on the toolbar. The Add new element window will be displayed, allowing you to create one or more new events.

Deleting a transition, event, or state

To delete a transition, event, or state:

1. Select the relevant transition cell, event header, or row name in the table.
2. Open the context menu and select **Delete from Model**.

Note

You cannot delete the *ROOT* state.

Menu for tabular view

When in tabular view, the menu includes an option of switching to diagram view (and vice versa).

Refreshing the contents of the statechart table

If you use the browser to add/remove elements, click the **Refresh** button in the toolbar to refresh the contents of the statechart table

Locating model elements

If the tabular view is set as the default view for a statechart, then when you try locating a model element included in the statechart, Rational Rhapsody will highlight the relevant cell in the statechart table.

Panel diagrams

A panel diagram provides you with a number of graphical control elements that you can use to create a graphical user interface (GUI) to monitor and regulate an application. Each control element can be bound to a model element (attribute/event/state). During animation, you can use the animated panel diagram to monitor (read) and regulate (change values/send events) your user application. For more information about animation, see [Animation](#).

This feature provides a convenient way to demonstrate the design and, additionally, provides you with an easy way to create a debug interface for the design.

Note

The panel diagram feature is only available for Rational Rhapsody in C and Rational Rhapsody in C++.

You can use a panel diagram to create the following types of panels for design and testing purposes:

- ♦ Hardware control panel designed for operating and monitoring machinery or instruments.
- ♦ Software graphical user interface (GUI) for display on a computer screen allowing the computer application user easier access to the application function than would be required if the user entered commands or other direct operational techniques.

Panel diagram features

You can create a panel diagram to design a graphical interface in Rational Rhapsody in C and Rational Rhapsody C++ projects. Developers can use the diagrams to:

- ◆ Simulate and prototype a panel
- ◆ Imitate hardware devices for users
- ◆ Activate and monitor a user application
- ◆ Test applications by triggering events and change attributes values

Note

Panel diagrams are intended only for simulating and prototyping, and not for use as a production interface for the user application. In addition, panel diagrams can only be “used” on the host and can be “used” only from within Rational Rhapsody.

The following illustration shows an animated panel diagram for a hypothetical coffee maker application. During animation, the developer of the application can test it by doing such things as, for example:

- ◆ Turn on the coffee maker application by clicking the **power** On/Off Switch control.
- ◆ Use the **coffeeContainer** and **milkContainer** Bubble Knob controls to increase/decrease the amount of coffee and milk that is available.
- ◆ Order a coffee by clicking the **evCoin** Push Button control. The following could happen:
 - Messages appear on the Matrix Display control, such as *Filling Coffee* or *Filling Cup*.
 - The **coffeeContainer** and **milkContainer** Level Indicator controls go down as these items are dispensed.
 - The **cup** Level Indicator control rises as coffee fills a cup, until the *Please Take Your Cup* message displays on the Matrix Display control.
 - The **Take cup** LED control turns red.
 - The **cupCounter** Digital Display control keeps a count of each cup of coffee made.
- ◆ Indicate that a cup of coffee has been taken by clicking the **evTakeCup** Push Button control, which could reset the coffee machine.



Creating a panel diagram

The following procedure lists the general steps to create and use a panel diagram. References to more specific procedures are noted when necessary.

1. Open your Rational Rhapsody model with model elements ready for use.
2. Create a panel diagram. Choose **Tools > Diagrams > Panel Diagram**.
For basic information on diagrams, including how to create, open, and delete them, see [Graphic editors](#).
3. Create a control element in your panel diagram; click any of the tools on the panel diagram **Diagram Tools**. See [Panel diagram drawing tools](#).
4. Bind the control to a model element; right-click a control and select **Features**. Use the **Element Binding** tab on the Control Properties tab. See [Bind a control element to a model element](#).
5. Make whatever changes you might want for the control; see:
 - ♦ [Change the settings for a control element](#)
 - ♦ [Change the properties for a control element](#), when applicable
 - ♦ [Setting the value bindings for a button array control](#), when applicable
 - ♦ [Changing the display name for a control element](#)
6. Set your model for animation. See [Animation](#).
7. Generate and make your model. Run your application and the animation for the panel diagram. When animation starts, the control on your panel diagram is initiated with its bound model element value. See [Basic code generation concepts](#).
8. Use your control elements on the animated panel diagram. Note that when animation is running, the Control Properties window and the Display Options window are unavailable.
9. Terminate animation to terminate the use of the control element and exit animation.
10. Use ReporterPLUS to produce a panel diagram report. For more information, see [ReporterPLUS](#).

Create panel diagram elements

The following sections describe the drawing tools available for a panel diagram. For basic information on diagrams, including how to create, open, and delete them, see [Graphic editors](#).

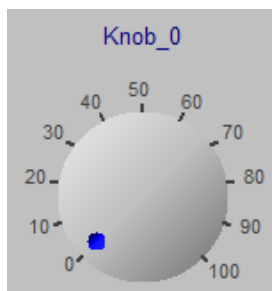
Panel diagram drawing tools

The **Diagram Tools** for a panel diagram contains the following tools.

Drawing Icon		Description
	Select	Lets you select a control on a panel diagram.
	Knob	Represents a Bubble Knob control. See Drawing a bubble knob control .
	Gauge	Represents a Gauge control. See Drawing a gauge control .
	Meter	Represents a Meter control. See Drawing a meter control .
	Level Indicator	Represents a Level Indicator control. See Drawing a level indicator control .
	Matrix Display	Represents Matrix Display control that shows a text string. See Drawing a matrix display control .
	Digital Display	Represents a Digital Display control that shows numbers. See Drawing a digital display control .
	LED	Represents a light-emitting diode control. See Drawing an LED control .
	On/Off	Represents an On/Off Switch control. See Drawing an on/off switch control .
	Push Button	Represents a Push Button control. See Drawing a push button control .
	Button Array	Represents a Button Array control. See Drawing a button array control .
	Text Box	Represents an editable Text Box control. See Drawing a text box control .
	Slider	Represents a Slider control. See Drawing a slider control .

Drawing a bubble knob control

The Bubble Knob control is an input/output control, as shown in the following figure in its default non-animated appearance:



Note the following information:

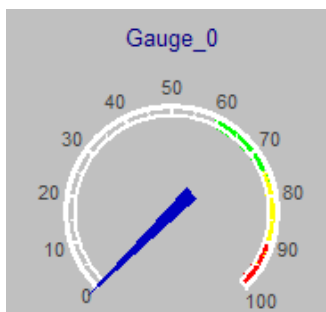
- ◆ You can bind (map) it to an attribute.
- ◆ Its attribute type is a Number.
- ◆ By default its control direction is set to In/Out, though you can change it to In or Out.

To draw a Bubble Knob control on a panel diagram:

1. Click the Knob button  in the **Diagram Tools**.
2. Click the drawing area to create a Bubble Knob control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Bind a control element to a model element](#).
4. Make whatever changes you might want for the control:
 - ◆ To change the settings for the control, see [Change the settings for a control element](#)
 - ◆ To change the properties for the control, see [Change the properties for a control element](#) and [Properties for a bubble knob control](#)
 - ◆ To change the display name for the control, see [Changing the display name for a control element](#)

Drawing a gauge control

The Gauge control is an output control that displays as an analog round dial, as shown in the following figure in its default non-animated appearance:



Note the following information:

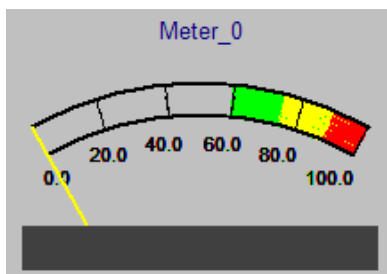
- ◆ You can bind it to an attribute.
- ◆ Its attribute type is a Number.

To draw a Gauge control on a panel diagram:

1. Click the Gauge button  in the **Diagram Tools**.
2. Click the drawing area to create a Gauge control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Bind a control element to a model element](#).
4. Make whatever changes you might want for the control:
 - ◆ To change the settings for the control, see [Change the settings for a control element](#)
 - ◆ To change the properties for the control, see [Change the properties for a control element](#) and [Properties for a gauge control](#)
 - ◆ To change the display name for the control, see [Changing the display name for a control element](#)

Drawing a meter control

The Meter control is an output control that displays as an analog meter, as shown in the following figure in its default non-animated appearance:



Note the following information:

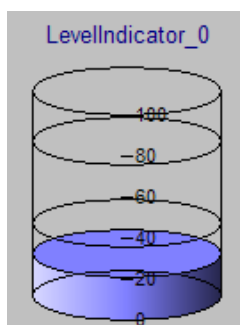
- ◆ You can bind it to an attribute.
- ◆ Its attribute type is a Number.

To draw a Meter control on a panel diagram:

1. Click the Meter button  in the **Diagram Tools**.
2. Click the drawing area to create a Meter control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Bind a control element to a model element](#).
4. Make whatever changes you might want for the control:
 - ◆ To change the settings for the control, see [Change the settings for a control element](#)
 - ◆ To change the properties for the control, see [Change the properties for a control element](#) and [Properties for a meter control](#)
 - ◆ To change the display name for the control, see [Changing the display name for a control element](#)

Drawing a level indicator control

The Level Indicator control is an output control. By default it displays as a vertical 3-dimensional cylindrical level indicator, as shown in the following figure in its default non-animated appearance. However, you can change its appearance (for example to a 3-dimensional square shape) through the **Properties** tab of the Control Properties window.



Note the following information:

- ◆ You can bind it to an attribute.
- ◆ Its attribute type is a Number.

To draw a Level Indicator control on a panel diagram:

1. Click the Level Indicator button  in the **Diagram Tools**.
2. Click the drawing area to create a Level Indicator control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Bind a control element to a model element](#).
4. Make whatever changes you might want for the control:
 - ◆ To change the settings for the control, see [Change the settings for a control element](#)
 - ◆ To change the properties for the control, see [Change the properties for a control element](#) and [Properties for a level indicator control](#)
 - ◆ To change the display name for the control, see [Changing the display name for a control element](#)

Drawing a matrix display control

The Matrix Display control is an output control, as shown in the following figure in an example of an animated appearance:



Note the following information:

- ◆ You can bind it to an attribute.
- ◆ Its attribute types are Number and String.

To draw a Matrix Display control on a panel diagram:

1. Click the Matrix Display button  in the **Diagram Tools**.
2. Click the drawing area to create a Matrix Display control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Bind a control element to a model element](#).
4. Make whatever changes you might want for the control:
 - ◆ To change the settings for the control, see [Change the settings for a control element](#)
 - ◆ To change the properties for the control, see [Change the properties for a control element](#) and [Properties for a matrix display control](#)
 - ◆ To change the display name for the control, see [Changing the display name for a control element](#)

Drawing a digital display control

The Digital Display control is an output control, as shown in the following figure in an example of an animated appearance:



Note the following information:

- ◆ You can bind it to an attribute.
- ◆ Its attribute types are a Number and a String.

To draw a Digital Display control on a panel diagram:

1. Click the Digital Display button  in the **Diagram Tools**.
2. Click the drawing area to create a Digital Display control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Bind a control element to a model element](#).
4. Make whatever changes you might want for the control:
 - ◆ To change the settings for the control, see [Change the settings for a control element](#)
 - ◆ To change the properties for the control, see [Change the properties for a control element](#) and [Properties for a digital display control](#)
 - ◆ To change the display name for the control, see [Changing the display name for a control element](#)

Drawing an LED control

The LED control is an output control, as shown in the following figure in an example of an animated appearance:



Note the following information:

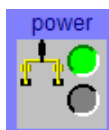
- ◆ You can bind it to a state or an attribute.
- ◆ Its attribute type is a Boolean.

To draw an LED control:

1. Click the LED button  in the **Diagram Tools**.
2. Click the drawing area to create a LED control.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Bind a control element to a model element](#).
4. Make whatever changes you might want for the control:
 - ◆ To change the settings for the control, see [Change the settings for a control element](#)
 - ◆ To change the properties for the control, see [Change the properties for a control element](#) and [Properties for a LED control](#)
 - ◆ To change the display name for the control, see [Changing the display name for a control element](#)

Drawing an on/off switch control

The On/Off Switch control is an input/output control, as shown in the following figure in one of its many possible shape styles in an example of an animated appearance:



Note the following information:

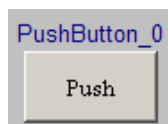
- ◆ You can bind it to a state or an attribute.
- ◆ Its attribute type is a Boolean.
- ◆ By default its control direction is set to In/Out, though you can change it to In or Out.

To draw an On/Off Switch control on a panel diagram:

1. Click the On/Off Switch button  in the **Diagram Tools**.
2. Click the drawing area to create a On/Off Switch control.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Bind a control element to a model element](#).
4. Make whatever changes you might want for the control:
 - ◆ To change the settings for the control, see [Change the settings for a control element](#)
 - ◆ To change the properties for the control, see [Change the properties for a control element](#) and [Properties for a on/off switch control](#)
 - ◆ To change the display name for the control, see [Changing the display name for a control element](#)

Drawing a push button control

The Push Button control is an input control, as shown in the following figure in its default non-animated appearance:



Note the following information:

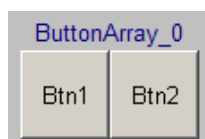
- ◆ You can bind it to an event.
- ◆ By default, this control injects a none parameter event.
- ◆ You can set a fix parameter for the event.

To draw a Push Button control on a panel diagram:

1. Click the Push Button button  in the **Diagram Tools**.
2. Click the drawing area to create a Push Button control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Bind a control element to a model element](#).
4. Make whatever changes you might want for the control:
 - ◆ To change the settings for the control, see [Change the settings for a control element](#)
 - ◆ To change the display name for the control, see [Changing the display name for a control element](#)

Drawing a button array control

The Button Array control is an input/output control, as shown in the following figure in its default non-animated appearance:



Note the following information:

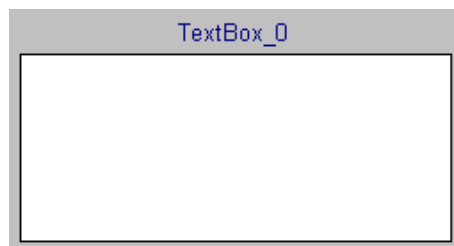
- ◆ You can bind it to an attribute. In addition, you can set a value for each switch to be set on the attribute.
- ◆ Its attribute types are a Number and a String.
- ◆ By default its control direction In/Out, though you can change it to In or Out.

To draw a Button Array control on a panel diagram:

1. Click the Button Array button  in the **Diagram Tools**.
2. Click the drawing area to create a Button Array control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Bind a control element to a model element](#).
4. Make whatever changes you might want for the control:
 - ◆ To change the settings for the control, see [Change the settings for a control element](#)
 - ◆ To change the value binding for the control; see [Setting the value bindings for a button array control](#).
 - ◆ To change the display name for the control, see [Changing the display name for a control element](#)

Drawing a text box control

The editable Text Box control is an input/output control, as shown in the following figure in its default non-animated appearance:



Note the following information:

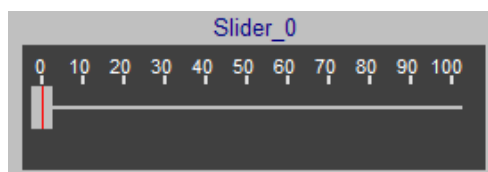
- ◆ You can bind it to an attribute.
- ◆ Its attributes types are a Number and a String.
- ◆ By default its control direction In/Out, though you can change it to In or Out.

To draw a Text Box control on a panel diagram:

1. Click the Text Box button  in the **Diagram Tools**.
2. Click the drawing area to create a Text Box control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Bind a control element to a model element](#).
4. Make whatever changes you might want for the control:
 - ◆ To change the settings for the control, see [Change the settings for a control element](#)
 - ◆ To change the display name for the control and/or its data flow direction, see [Changing the display name for a control element](#).
 - ◆ To change the format settings (for example, line, fill, and font) for a Text Box control, see [Change the format of a single element](#).

Drawing a slider control

The Slider control is an input/output control, as shown in the following figure in its default non-animated appearance:



Note the following information:

- ◆ You can bind it to an attribute.
- ◆ Its attribute type is a Number.
- ◆ By default its control direction In/Out, though you can change it to In or Out.

To draw a Slider control on a panel diagram:

1. Click the Slider button  in the **Diagram Tools**.
2. Click the drawing area to create a Slider control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Bind a control element to a model element](#).
4. Make whatever changes you might want for the control:
 - ◆ To change the settings for the control, see [Change the settings for a control element](#)
 - ◆ To change the properties for the control, see [Change the properties for a control element](#) and [Properties for a slider control](#)
 - ◆ To change the display name for the control, see [Changing the display name for a control element](#)

Bind a control element to a model element

In a panel diagram, the control element is a GUI that has to be connected to some “real” source/target element. In Rational Rhapsody, binding (mapping) ties the operation between the control element to the model element it is to regulate or monitor.

A binding definition for a control element defines the following binding settings of each control element.

- ◆ Element type:
 - Control element - input sets data to bound element
 - Monitor element - output gets data from bound element
- ◆ Valid model elements for binding (attribute, event, and state)
- ◆ Value attribute types that can be set/get by the control element (Number, String, or Boolean)

A binding definition for a control element is predefined in Rational Rhapsody. It cannot be changed. For example, the Bubble Knob has the following binding definition:

- ◆ Element role: input, output, or both
- ◆ Valid model elements: attributes
- ◆ Value types: numbers

Be sure that the bounded element type (for example, an `int`) is being supported by the control.

In the binding operation, you have to set the model element for binding. You can also set the instance path.

Binding a control element

To bind a control element to a model element in a panel diagram:

1. Right-click the control and select **Features** to open the Control Properties window.
2. On the **Element Binding** tab, depending on your situation:
 - ♦ If the control has no binding, the Control Binding Browser opens with the project container as the selected item. Use the browser to navigate to and select the elements for which you want to bind to the control element, or you can enter the element path in the **Instance Path** box.

Note that the browser root is the project and the end nodes are the meta classes that can be bound for the particular control element. If no relevant end node is found, the system notifies you that no relevant item was found and the Control Properties window does not open.

- ♦ If the control has a bound element, the browser opens with the bound model element selected.
- ♦ If no relevant element for binding is found in the model, the **Element Binding** tab displays blank with a note to that effect.

For more information about binding, see [More about binding a control element](#).

3. Click **OK**.

More about binding a control element

Note the following about binding a control element:

- ♦ **Binding an item of a modeled root Instance**

In the case where the element for binding is owned by a modeled root Instance, the object is displayed by the browser on the **Element Binding** tab on the Control Properties window containing all relevant items for binding. You can select the element from the browser or alternatively type in the element path (stating at modeled root object) in the **Instance Path** box.

- ♦ **Binding an item of a dynamic root Instance**

In the case where the element for binding is owned by a dynamic root Instance (modeled class that will be instantiated at run time), the element root class is displayed by the browser on the **Element Binding** tab containing all relevant items for binding. You can select the element from the browser or alternatively type in the element path (stating at modeled root class) in the **Instance Path** box.

In both ways, if the dynamic Instance name is different from the default class instance name, the name should be entered following item selection.

- ◆ **Binding an item with multiplicity on its root object and parts**

In the case where bound element owner parts has multiplicity: Selecting the element through the browser on the **Element Binding** tab creates the path in the **Instance Path** box with “0” multiplicity on the relevant parts. You can then set the multiplicity as needed. If you enter a path, it is your responsibility to add multiplicity where needed.

The following table summarizes the binding (mapping) characteristics for each panel diagram control element:

Control Name	Direction		Bound Element			Attribute Type		
	In	Out	Event	State	Attribute	Number	String	Boolean
Knob	✓	✓			✓	✓		
Gauge		✓			✓	✓		
Meter		✓			✓	✓		
Level Indicator		✓			✓	✓		
Matrix Display		✓			✓	✓	✓	
Digital Display		✓			✓	✓	✓	
LED		✓		✓	✓			✓
On/Off Switch	✓	✓		✓	✓			✓
Push Button	✓		✓					
Button Array	✓	✓			✓	✓	✓	
Text Box	✓	✓			✓	✓	✓	
Slider	✓	✓			✓	✓		

Attribute types

Only attributes that hold predefined primitive (or enumeration) types could be bound. The supported predefined types are:

- ◆ Number: int, unsigned int, short, unsigned short, long, double, float, RhpInteger, RhpPositive, RhpReal
- ◆ String: char, char*, RhpString, OMString, CString
- ◆ Boolean: Bool, OMBoolean, RhpBoolean, RiCBoolean

Change the settings for a control element

You can change the settings for the control elements available for a panel diagram. For example, when applicable, its:

- ♦ Minimum and maximum values
- ♦ Shape style (in the case of a On/Off switch)
- ♦ Caption (in the case of the Push Button control)
- ♦ Color scheme (in the case of the Matrix Display and Digital Display controls)

You do this through the **Settings** tab on the Control Properties window for a control element.

Where possible (for example, for the Bubble Knob control), you can also change their control direction (to input, output, or both). By default, they are set to **InOut**. You can use the **Control Direction** area of the **Settings** tab on the Control Properties window to change the control direction when possible.

Changing the settings for a control

To change the settings for a control:

1. Right-click a control and select **Features** to open the Control Properties window.
2. Make your changes on the **Settings** tab. Note that the settings that are available depend on the type of control element you have selected.
3. If applicable for a control, select an radio button in the **Control Direction** area:
 - ♦ **In** for input flow
 - ♦ **Out** for output flow
 - ♦ **InOut** for input/output flow
4. Click **OK**.

Change the properties for a control element

You can change the properties for many of the control elements available for a panel diagram (for example, its background color, range values, caption, and so on). You can make changes through the **Properties** tab of the Control Properties window for a control. These properties are ActiveX controls.

Note: The **Properties** tab displays only when appropriate for the selected control element. In addition, the tab shows only those settings that are applicable to that control element.

Properties for a bubble knob control

The following table lists the properties that appear on the **Properties** tab of the Control Properties window for a Bubble Knob control. You can change the properties as follows:

Device Settings	Explanation
BackgroundColor	To change the background color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
DivisionLineThickness	To change the thickness of the major division lines (also referred to tick markers), change the value in the right column. The default value is 2.
DotColor	To change the color for the dot (indicator mark) on the Bubble Knob control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
EndAngle	To change the distance around the dial (between the minimum and maximum values), change the value in the right column. The default value is -45.
Font	To change the font for the text (for example, the numbers) on the control, click the Ellipses button in the right column and select a font from the Font window that displays. The default value is Arial.
GradientFactor	To change the gradient factor for the control, change the value in the right column. The higher the number the more pronounced the gradient for the appearance of the knob, which displays as light to dark. The default value is 0.7.
LineColor	To change the color of all the tick markers, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
MaximumValue	To change the maximum value for the control, change the value in the right column. The default value is 100.
MinimumValue	To change the minimum value for the control, change the value in the right column. The default value is 0.

Device Settings	Explanation
NumberOfDivisions	To change the number of major division lines (tick markers) for the control, change the value in the right column. The default value is 10. For example, with the maximum value set at 100, minimum value at 0, and division value at 5, your Bubble Knob control would show major tick markers at 0, 20, 40, 60, 80, and 100.
NumberOfSubdivisions	To change the number of minor division lines (tick markers) between two major ones, change the value in the right column. The default value is 1, which means no minor tick marker displays between two major tick markers. For example, to make three subdivision areas appear between two major markers, enter a value of 3.
RelativeBubbleRadius	To change the relative bubble radius for the control, change the value in the right column. This regulates the relative size of the Bubble Knob control, which includes its number scale. The default value is 0.6.
RelativeDotPositionRadius	To change the placement (closer or farther away) of the dot indicator relative to the 0 value marker, change the value in the right column. The default value is 0.75.
RelativeDotRadius	To change the size of the dot indicator, change the value in the right column. The default value is 0.11.
RelativeExternalRadius	To change the relative external radius for the control, change the value in the right column. This setting changes the length of the major and minor tick markers while still touching the control. The default value is 1.1.
RelativeInternalRadius	To change the relative internal radius for the control, change the value in the right column. This setting changes the length of the major and minor tick markers and how far away they are from the bubble. The default value is 1.
RelativeTextRadius	To change the relative text radius for the control, change the value in the right column. This setting changes the distance between the scale numbers and their associate tick makers. The default value is 0.98.
StartAngle	To change the position of the minimum value marker and the dot indicator, change the value in the right column. The default value is 225.
SubdivisionLineThickness	To change the thickness of the minor division lines (tick markers), change the value in the right column. The default value is 1.
TextColor	To change the color for the text (scale numbers) on the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
Value	To change the default placement of the dot indicator, change the value in the right column. The default value is 0.
ValueFormatString	To change the value format of the numbers on the control, change the value in the right column. The default value is %.0f, which shows numbers, for example, as 0, 10, 20, and so on. For the value %.1f, the control would show number as 0.0, 10.0, 20.0, and so on.

Properties for a gauge control

The following table lists the properties that appear on the **Properties** tab of the Control Properties window for a Gauge control. You can change the properties as follows:

Device Settings	Explanation
BackColor	To change the back color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
BackgroundPicture	To add/change an image that displays as the background for the control, click the Ellipses button in the right column and open a bitmap file.
Caption	To insert a caption to show in the center of the Gauge control, type your text in the right column. You can use <code>RelativeCaptionY</code> , <code>RelativeCaptionX</code> , <code>RelativeCenterY</code> , and <code>RelativeCenterX</code> to position the caption somewhere on the gauge other than its center.
CaptionColor	To change the color for your caption text, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
CaptionFont	To change the font for your caption text, click the Ellipses button in the right column and select a font from the Font window that displays. The default value is <code>Arial</code> .
DivisionLineThickness	To change the thickness of the major division lines (also known as tick markers), change the value in the right column. The default value is 2.
EnclosingCircleColor	To change the background color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
EndAngle	To change the angle of the scale on the high end of the control, change the value in the right column. The value widens (lower value) or shrinks (higher value) the empty space between the low and high ends of the control. The default is -45.
ExternalCircleThickness	To change the thickness of the external circle of the control, change the value in the right column. The default value is 3.
ForeColor	To change the foreground color, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
GreenColor	To change the color (typically green) of the OK area of the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
GreenStartValue	To change the beginning value for the OK area on the control, change the value in the right column. The default value is 60.
IndexColor	To change the color of the needle indicator for the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
IndexLineThickness	To change the thickness of the needle indicator for the control, change the value in the right column. The default value is 0.

Device Settings	Explanation
InternalCircleThickness	To change the thickness of the internal circle of the control, change the value in the right column. The default value is 2.
MaximumValue	To change the maximum value for the control, change the value in the right column. The default value is 100.
MinimumValue	To change the minimum value for the control, change the value in the right column. The default value is 0.
NumberColor	To change the color of the numbers on the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
NumberFont	To change the font for the numbers on the control, click the Ellipses button in the right column and select a font from the Font window that displays. The default value is Arial.
NumberOfDivisions	To change the number of major division lines (tick markers) for the control, change the value in the right column. The default value is 10. For example, with the maximum value set at 100, minimum value at 0, and division at 5, your control would show major tick markers at 0, 20, 40, 60, 80, and 100.
NumberOfSubdivisions	To change the number of minor division lines (tick markers) between two major ones, change the value in the right column. The default value is 2, which means one minor tick marker displays between two major tick markers. For example, to make three subdivision areas appear between two major markers, enter a value of 3.
RedColor	To change the color (typically red) of the Warning area of the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
RedStartValue	To change the beginning value for the Warning area on the gauge, change the value in the right column. The default value is 90.
RelativeCaptionX	To change the relative X coordinate (east/west) of the text entered for Caption, change the value in the right column. The default value is 0.5.
RelativeCaptionY	To change the relative Y (north/south) coordinate of the text entered for Caption, change the value in the right column. The default value is 0.5.
RelativeCenterX	To change the relative center Y coordinate of the control, change the value in the right column. The default value is 0.5.
RelativeCenterY	To change the relative center X coordinate of the control, change the value in the right column. The default value is 0.55.
RelativeEnclosingCircleRadius	To change the relative enclosing circle radius of the control, change the value in the right column. The default value is .98.
RelativeExternalRadius	To change the relative external radius of the control, change the value in the right column. The default value is 1.1.

Device Settings	Explanation
RelativeIndexBackLength	To change the relative length the needle indicator from the back (not pointy) end, change the value in the right column. The default value is 0.3.
RelativeIndexLength	To change the relative length of the needle indicator from the front (pointy) end, change the value in the right column. This lengthens the needle indicator at its back (wider) end. The default value is 1.2.
RelativeInternalRadius	To change the relative internal radius of the control, change the value in the right column. The default value is 0.35.
RelativeTextRadius	To change the relative position of the text (scale numbers), change the value in the right column. The default value is 1.1.
ScaleCircleColor	To change the color of the scale circle, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
StartAngle	To change the position of the minimum value marker and the needle indicator, change the value in the right column. The default value is 225.
StepValue	To change the step value, change the value in the right column. The default is 1.
SubdivisionLineThickness	To change the thickness of the minor division lines (tick markers), change the value in the right column. The default value is 1.
TailAngle	To change the thickness of the needle indicator, change the value in the right column. The default is 165.
Value	To change the default value for the needle indicator, change the value in the right column. The default value is 0.
ValueFormatString	To change the value format of the scale number, change the value in the right column. The default value is %.0f, which shows numbers, for example, as 0, 10, 20, and so on. For the value %.1f, the control would show number as 0.0, 10.0, 20.0, and so on.
YellowColor	To change the color (typically yellow) of the Caution area of the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
YellowStartValue	To change the beginning value for the Caution area on the gauge, change the value in the right column. The default value is 75.

Properties for a meter control

The following table lists the properties that appear on the **Properties** tab of the Control Properties window for a Meter control. You can change the properties as follows:

Device Settings	Explanation
BackgroundImage	To add an image so that it displays as the background for the control, click the Ellipses button in the right column and open a bitmap file.
BottomCoverColor	To change the color of the bottom cover for the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays. You can use this setting in conjunction with BottomCoverRelativeHeight.
BottomCoverLabelFront	To change the font for the text on the bottom cover for the control, click the Ellipses button in the right column and select a font from the Font window that displays. The default value is Arial. You use this setting in conjunction with Caption and BottomCoverTextColor.
BottomCoverRelativeHeight	To change the height of the bottom cover area for the control, change the value in the right column. The default is 0.2. You can use this setting in conjunction with BottomCoverColor.
BottomCoverTextColor	To change the color for the text that displays in the bottom cover area, click the drop-down arrow in the right column and select a color from the Palette tab that displays. You use this setting in conjunction with Caption and BottomCoverLabelFront.
Caption	To insert a caption to appear in the bottom cover area of the control, type your text in the right column. You can use this setting in conjunction with BottomCoverLabelFront and BottomCoverTextColor.
GreenColor	To change the color (typically green) of the OK area of the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
GreenStartValue	To change the beginning value for the OK area on the control, change the value in the right column. The default value is 60.
IndexColor	To change the color of the needle indicator for the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
IndexThickness	To change the thickness of the needle indicator, change the value in the right column. The default value is 2.
InstrumentBackgroundColor	To change the background color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
MaximumValue	To change the maximum value for the control, change the value in the right column. The default value is 100.

Device Settings	Explanation
MinimumValue	To change the minimum value for the control, change the value in the right column. The default value is 0.
NumberOfDivisions	To change the number of major division lines (tick markers) for the control, change the value in the right column. The default value is 5. For example, with the maximum value set at 100, minimum value at 0, and division at 5, your control would show major tick markers at 0, 20, 40, 60, 80, and 100.
NumberOfSubdivisions	To change the number of minor division lines (tick markers) between two major ones, change the value in the right column. The default value is 1, which means no minor tick marker displays between two major tick markers. For example, to make three subdivision areas appear between two major markers, enter a value of 3.
RedColor	To change the color (typically red) of the Warning area of the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
RedStartValue	To change the beginning value for the Warning area on the control, change the value in the right column. The default value is 90.
RelativeIndexLength	To change the relative length of the needle indicator, change the value in the right column. The default value is 1.1.
RelativeTextRadius	To change the relative position of the text (scale numbers), change the value in the right column. The default value is 0.9.
ScaleColor	To change the color of the scale outline and numbers, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
ScaledCircleRelativeDiameter	To change the relative diameter of the control, change the value in the right column. The default is 1.75.
ScaledCircleRelativeShifting	To change the relative height of the control, change the value in the right column. The default is 0.3.
ScaleRelativeWidth	To change the relative wide of the scale, change the value in the right column. The default is 0.1.
ScaleThickLineWidth	To change the top and bottom lines of the scale, change the value in the right column. The default is 2.
ScaleThinLineWidth	To change the thickness of the vertical lines of the scale, change the value in the right column. The default is 1.
ShadedAreaRelativeSize	The default is 0.
SmallScaleFont	To change the font for the scale numbers, click the Ellipses button in the right column and select a font from the Font window that displays. The default value is Arial.
StepValue	To change the step value, change the value in the right column. The default is 1.
Value	To change the default value for the needle indicator, change the value in the right column. The default value is 0.

Device Settings	Explanation
ValueFormatString	To change the value format of the scale number, change the value in the right column. The default value is <code>%.1f</code> , which shows numbers, for example, as 0.0, 10.0, 20.0, and so on. For the value <code>%.0f</code> , the control would show number as 0, 10, 20, and so on.
VisibleScaleRelativeSize	To change the relative size of the visible scale, change the value in the right column. The default value is <code>1.2</code> .
YellowColor	To change the color (typically yellow) of the Caution area of the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
YellowStartValue	To change the beginning value for the Caution area on the control, change the value in the right column. The default value is <code>75</code> .

Properties for a level indicator control

The following table lists the properties that appear on the **Properties** tab of the Control Properties window for a Level Indicator control. You can change the properties as follows:

Device Settings	Explanation
BackColor	To change the background color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
CurrentValue	To change the current (default) value for the control, change the value in the right column. The default is 20.
DrawTextLabels	The default is True.
EnableThreshold1	To enable/disable the appearance of the top middle threshold level line, change the value in the right column. True enables the appearance; False disables it. The default is True.
EnableThreshold2	To enable/disable the appearance of the bottom middle threshold level line, change the value in the right column. True enables the appearance; False disables it. The default is True.
Font	To change the font for the numbers on the control, click the Ellipses button in the right column and select a font from the Font window that displays. The default value is Arial.
FormatString	To change the value format of the scale numbers, change the value in the right column. The default value is %.0f, which shows numbers, for example, as 0, 10, 20, and so on. For the value %.1f, the control would show number as 0.0, 10.0, 20.0, and so on.
GradientFactor	To change the gradient factor for the control, change the value in the right column. The higher the number the more pronounced the gradient for the appearance of the knob, which displays as light to dark. The default value is 0.7.
HorizontalLayout	To change the layout of the level indicator to be horizontal, change the value in the right column to True. The default is False.
LiquidColor	To change the color of the level indicator, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
MaximumValue	To change the maximum value for the control, change the value in the right column. The default value is 100.
MinimumValue	To change the minimum value for the control, change the value in the right column. The default value is 0.
NumberOfDivisions	To change the number of major division lines (tick markers) for the control, change the value in the right column. The default value is 5. For example, with the maximum value set at 100, minimum value at 0, and division at 5, your control would show major tick markers at 0, 20, 40, 60, 80, and 100.

Device Settings	Explanation
NumberOfSubdivisions	To change the number of minor division lines (tick markers) between two major ones, change the value in the right column. The default value is 1, which means no minor tick marker displays between two major tick markers. For example, to make three subdivision areas appear between two major markers, enter a value of 3.
RelativeDepth	To change the appearance of the relative depth of the control, change the value in the right column. The default is 0.2. For example, a value of 0 gives the control a flat tall rectangular appearance. While a value of 0.2 gives it a 3-dimensional cylindrical appearance. Note that the appearance of the level indicator control is also affected by SquareShape, RelativeHeight and RelativeWidth. Note also that the appearance of the orientation of the level indicator is affected by HorizontalLayout.
RelativeHeight	To change the relative height of the control, change the value in the right column. The default is 0.9.
RelativeWidth	To change the relative width of the control, change the value in the right column. The default is 0.9.
ShadedAreaRelativeSize	The default is 0.
SquareShape	To change the appearance of the Level Indicator control to look like a 3-dimensional square, change the value in the right column to True. The default is False.
Threshold1Color	To change the color of the top middle threshold level line, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
Threshold1Thickness	To change the thickness of the top middle threshold level line, change the value in the right column. The default is 1.
Threshold1Value	To change the position of the bottom middle threshold level line, change the value in the right column. The default is 75.
Threshold2Color	To change the color of the bottom middle threshold level line, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
Threshold2Thickness	To change the thickness of the bottom middle threshold level line, change the value in the right column. The default is 1.
Threshold2Value	To change the position of the bottom threshold level line, change the value in the right column. The default is 35.
UseColorGradients	The default is True.

Properties for a matrix display control

The following table lists the properties that appear on the **Properties** tab of the Control Properties window for a Matrix Display control. You can change the properties as follows:

Device Settings	Explanation
BackColor	To change the background color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
Caption	To insert a caption to appear on the control, type your text in the right column.
Style	To change the appearance of the background of the control, change the value in the right column. The default is 0.

Properties for a digital display control

The following table lists the properties that appear on the **Properties** tab of the Control Properties window for a Digital Display control. You can change the properties as follows:

Device Settings	Explanation
BackColor	To change the background color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
DisplayedString	To insert text to display on the control, type your text in the right column.
Style	To change the appearance of the background for the control, change the value in the right column. The default is 4.

Properties for a LED control

The following table lists the properties that appear on the **Properties** tab of the Control Properties window for an LED control. You can change the properties as follows:

Device Settings	Explanation
BackColor	To change the background color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays
BlackWhenOff	To change the appearance of the control to black when the application is off, change the value in the right column to True. The default is False.
Blinking	To change it so that the LED is blinking, change the value in the right column to True. The default is False.
BlinkingTimeMillesec	To change the blinking rate, change the value in the right column. The default is 300.
Color	To change the color for the LED, change the value in the right column. The default is 2.
State	To change the state for the LED, change the value in the right column to True. The default is False.
Style	To change the appearance of the background of the control, change the value in the right column. The default is 0.

Properties for a on/off switch control

The following table lists the properties that appear on the **Properties** tab of the Control Properties window for an On/Off Switch control. You can change the properties as follows:

Device Settings	Explanation
BackColor	To change the background color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
IconSet	To change the appearance of the On/Off switch icon, change the value in the right column. 0 =  1 =  2 =  3 =  4 =  5 =  6 =  7 =  8 =  9 =  (default)
State	To change the state of the control, change the value in the right column. The default is True.
UserInteractionEnabled	To change is user interaction is enable, change the value in the right column. The default is True.

Properties for a slider control

The following table lists the properties that appear on the **Properties** tab of the Control Properties window for a Slider control. You can change the properties as follows:

Device Settings	Explanation
BackgroundColor	To change the background color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
DivisionLineThickness	To change the thickness of the major division lines (also referred to tick markers), change the value in the right column. The default value is 2.
DotColor	To change the color for the indicator mark on the Slider control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
EndAngle	To change the distance on the Slider rule (between the minimum and maximum values), change the value in the right column. The default value is -45.
Font	To change the font for the text (for example, the numbers) on the control, click the Ellipses button in the right column and select a font from the Font window that displays. The default value is Arial.
GradientFactor	To change the gradient factor for the control, change the value in the right column. The higher the number the more pronounced the gradient for the appearance of the Slider control, which displays as light to dark. The default value is 0.7.
LineColor	To change the color of all the tick markers, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
MaximumValue	To change the maximum value for the control, change the value in the right column. The default value is 100.
MinimumValue	To change the minimum value for the control, change the value in the right column. The default value is 0.
NumberOfDivisions	To change the number of major division lines (tick markers) for the control, change the value in the right column. The default value is 10. For example, with the maximum value set at 100, minimum value at 0, and division value at 5, your Slider control would show major tick markers at 0, 20, 40, 60, 80, and 100.
NumberOfSubdivisions	To change the number of minor division lines (tick markers) between two major ones, change the value in the right column. The default value is 1, which means no minor tick marker displays between two major tick markers. For example, to make three subdivision areas appear between two major markers, enter a value of 3.
RelativeBubbleRadius	To change the relative bubble radius for the control, change the value in the right column. This regulates the relative size of the Bubble Knob control, which includes its number scale. The default value is 0.1.
RelativeDotPositionRadius	To change the placement (closer or farther away) of the dot indicator relative to the 0 value marker, change the value in the right column. The default value is 0.75.

Device Settings	Explanation
RelativeDotRadius	To change the size of the indicator mark, change the value in the right column. The default value is 2.
RelativeExternalRadius	To change the relative external radius for the control, change the value in the right column. This setting changes the length of the major and minor tick markers and how far away they are from the indicator mark line. The default value is 6.25.
RelativeInternalRadius	To change the relative internal radius for the control, change the value in the right column. This setting changes the length of the major and minor tick markers and how far away they are from the the indicator mark line. The default value is 4.75.
RelativeTextRadius	To change the relative text radius for the control, change the value in the right column. This setting changes the distance between the scale numbers and their associate tick makers. The default value is 7.5.
StartAngle	To change the position of the minimum value marker and the indicator mark, change the value in the right column. The default value is 225.
SubdivisionLineThickness	To change the thickness of the minor division lines (tick markers), change the value in the right column. The default value is 1.
TextColor	To change the color for the text (scale numbers) on the control, click the drop-down arrow in the right column and select a color from the Palette tab that displays.
Value	To change the default placement of the indicator mark, change the value in the right column. The default value is 0.
ValueFormatString	To change the value format of the numbers on the control, change the value in the right column. The default value is %.0f, which shows numbers, for example, as 0, 10, 20, and so on. For the value %.1f, the control would show number as 0.0, 10.0, 20.0, and so on.

Setting the value bindings for a button array control

To set the value bindings for a Button Array control for a panel diagram:

1. Right-click a Button Array control and select **Features**.
2. On the **Value Binding** tab, change the name of a button and its value.
3. Optionally, you can click <New> to create another button in your array.
4. Click **OK**.

Changing the display name for a control element

You can change the display option for the name for any of the control elements available for a panel diagram. When you create a control, by default the system displays the name of the element (for example, Gauge_1 and Meter_5). You can use the Control Display Options window to change the display option for the name of your control element.

To change the display name and/or data flow options for a control element:

1. Right-click a control on the panel diagram and select **Display Options** to open the Control Display Options window.
2. In the **Control Name** area, select an radio button:
 - ♦ **Bound element full path** displays the full path of the bound element (for example, `Default.itsClass_0.speed`).
 - ♦ **Bound element** displays the name of the bound element (for example, `speed`).
 - ♦ **Name** displays the name of the control (for example, **Gauge_1**). This is the default.
 - ♦ **None** does not display any name.
3. Click **OK**.

Panel diagram limitations

The panel diagram feature has the following limitations:

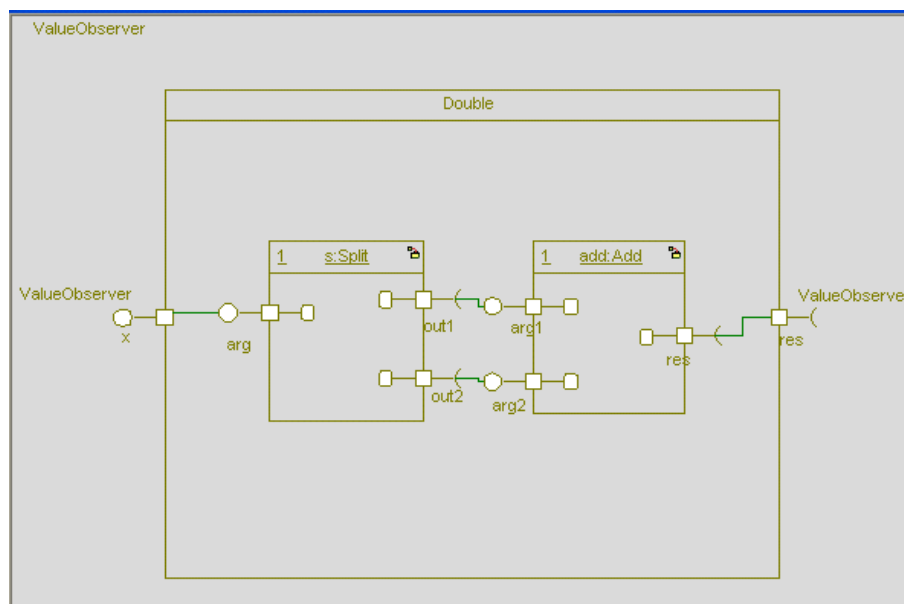
- ♦ Does not support composition made by relations (supports composition made by part only)
- ♦ Cannot launch an event with arguments
- ♦ Does not have support for graphical DiffMerge
- ♦ RiCString typed attribute (Rational Rhapsody Developer for C) cannot be bound

Structure diagrams

Structure diagrams model the structure of a composite class; any class or object that has an OMD can have a structure diagram. In addition, a structure diagram supports some the features supported by an OMD and uses the properties defined in the ObjectModelGE subject.

Object model diagrams focus more on the specification of classes, whereas structure diagrams focus on the instances used in the model. Although you can put classes in structure diagrams and objects in the OMD, the toolbars for the diagrams are different to allow a distinction between the specification of the system and its structure.

The following figure shows a structure diagram.



Structure diagram drawing Tools

The **Diagram Tools** for a structure diagram includes the following tools:

Drawing Tool	Name	Description
	Composite class	A container class. You can create objects and relations inside a composite class. See Composite classes for more information.
	Object	The structural building block of a system. Objects form a cohesive unit of state (data) and services (behavior). Every object has a public part and an private part. See Objects for more information.
	File	Only available for Rational Rhapsody in C. It allows you to create file model elements. A file is a graphical representation of a specification (.h) or implementation (.c) source file. See External files in C for more information.
	Port	Draws connection points among objects and their environments. See Structure diagram ports for more information.
	Link	Creates an association between the base classes of two different objects. See Links and associations for more information.
	Dependency	Creates a relationship in which the proper functioning of one element requires information provided by another element. See Dependency uses for more information.
	Flow	Specifies the flow of data and commands within a system. See Flows mechanism for more information.

The following sections describe how to use these tools to draw the parts of a structure diagram. See [Graphic editors](#) for basic information on diagrams, including how to create, open, and delete them.

Composite classes

For detailed information about composite classes, see [Creating composite classes](#) in [Object model diagrams](#).

Objects

Objects are the structural building blocks of a system. They form a cohesive unit of state (data) and services (behavior). Every object has a specification part (public) and an implementation part (private). See [Objects](#) for detailed information about objects.

Creating an object

To create an object:

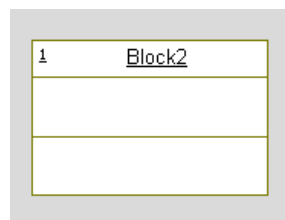
1. Click the **Object** icon in the **Diagram Tools**.
2. Double-click, or click-and-drag, in the drawing area.
3. Edit the default name, then press **Enter**.

If you specify the name in the format <ObjectName:ClassName> (for an object with explicit type) and the class <ClassName> exists in the model, the new object will reference it. If it does not exist, Rational Rhapsody prompts you to create it.

Alternatively, you can select **Edit > Add New > Object**. If you want to add a object to an OMD, you must use this method because the **Diagram Tools** do not include a **Object** tool.

In the OMD, an object is shown like a class box, with the following differences:

- ♦ The name of the object is underlined.
- ♦ The multiplicity is displayed in the upper, left-hand corner.



As with classes, you can display the attributes and operations in the object. See [Display option settings](#) for detailed information.

Features of objects

The Features window enables you to change the features of an object, including its concurrency and multiplicity.

An object has the following features:

- ♦ **Name** specifies the name of the element. The default name is `object_n`, where *n* is an incremental integer starting with 0.
- ♦ **L** specifies the label for the element, if any. See [Descriptive labels for elements](#) for information on creating labels.
- ♦ **Stereotype** specifies the stereotype of the object, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Stereotypes](#) for information on creating stereotypes.

Note: The COM stereotypes are constructive; that is, they affect code generation.

- ♦ **Main Diagram** specifies the main diagram for the object. This field is available only for objects with implicit type.
- ♦ **Concurrency** specifies the concurrency of the object. This field is available only for objects with implicit type. The possible values are as follows:
 - **Sequential** where the element will run with other classes on a single system thread. This means you can access this element only from one active class.
 - **Active** where the element will start its own thread and run concurrently with other active classes.
- ♦ **Type** specifies the class of which the object is an instance. To view that class features, click the Invoke Feature Dialog button next to the **Type** field.

In addition to the names of all the instantiated classes in the model, this list includes the following choices:

- **<Implicit>** specifies an implicit object
- **<Explicit>** specifies an explicit object
- **<New>** enables you to specify a new class
- **<Select>** enables you to browse for a class using the selection tree
- ♦ **Multiplicity** specifies the number of occurrences of this instance in the project. Common values are one (1), zero or one (0,1), or one or more (1..*).
- ♦ **Initialization** specifies the constructor being called when the object is created. See the [Actual Call window for objects](#).
- ♦ **Relation to whole** enables you to name the relation for a part. If the object is part of a composite class, enable the **Knows its whole as** check box and type a name for the relation in the text box. This relation is displayed in the browser under the Association Ends category under the instantiated class or implicit object.

If the **Relation to whole** field is specified on the **General** tab, the Features window includes tabs to define that relation and its properties. However, on the tab that specifies the features of its whole (in the illustration of the `itsController` tab), only the fields **Name**, **Label**, **Stereotype**, and **Description** can be modified. See [Association features](#) for more information on relation features.

Actual Call window for objects

If you click the Ellipsis button beside the Features window **Initialization** field, the Actual Call window opens so you can see the details of the call. If the part does not have a constructor with parameters, this field is dimmed and the button is unavailable.

Changing the order of objects

To change the order of objects:

1. In the browser, right-click the **Object** category icon and, then select **Edit Objects Order**.
2. Clear the **Use default order** check box.
3. Select the object you want to move.
4. Click **Up** to generate the object earlier or **Down** to generate it later.
5. Click **OK**.

Supported Rational Rhapsody functionality in objects

The following table lists the Rational Rhapsody functionality supported by objects.

Functionality	Description
Roundtrip	Full support.
Diff/Merge	Full support.
ReporterPLUS	ReporterPLUS shows objects separately from objects.
Internal reporter	Objects are printed in the group "Objects."
Search and replace	You can search for objects in the model and select their type from the list of possible types. See Searching in the model for more information.
Check Model	The same checks for objects are used for objects. See Checks for more information.
XMI	Controls export of objects.
DOORS	Objects can be exported to and checked by Rational DOORS.

Structure diagram ports

A *port* is a distinct interaction point between a class and its environment, or between (the behavior of) a class and its internal parts. A port enables you to specify classes independently of the environment in which they will be embedded; the internal parts of the class can be completely isolated from the environment, and vice versa. See [Ports](#) for detailed information about ports.

Links and associations

Rational Rhapsody separates links from associations so you can have unambiguous model elements for links with a distinct notation in the diagrams. This allows specification of the following items:

- ♦ Links without having to specify the association being instantiated by the link.
- ♦ Features of links that are not mapped to an association.

See [Links](#) for detailed information about links.

Dependency uses

A *dependency* exists when the implementation or functioning of one element (class or package) requires the presence of another element. For example, if class C has an attribute a that is of class D, there is a dependency from C to D. See [Dependencies](#) for detailed information about dependencies.

Flows mechanism

Flows and flowitems provide a mechanism for specifying exchange of information between system elements at a high level of abstraction. This functionality enables you to describe the flow of data and commands within a system at a very early stage, before committing to a specific design. As the system specification evolves, you can refine the abstraction to relate to the concrete implementation.

See [Flows and flowitems](#) for detailed information about flows and flowitems.

External files in C

Rational Rhapsody in C enables you to create model elements that represent external files. An *external file* (or simply *file*) is a graphical representation of a specification (.h) or implementation (.c) source file. This new model element enables you to use functional modeling and take advantage of the capabilities of Rational Rhapsody (modeling, execution, code generation, and reverse engineering) without radically changing the existing external files.

See [Files](#) for more information.

Collaboration diagrams

Collaboration diagrams, like sequence diagrams, display objects, their messages, and their relationships in a particular scenario or use case. Sequence diagrams emphasize message flow and can indicate the time sequence of messages sent or received, whereas collaboration diagrams emphasize relationships between objects.

Collaboration diagrams overview

Collaboration diagrams depict classifier roles and their interactions, or messages, via their association roles. A *classifier role* is an instance of a class (or classifier), that is defined only in the context of the collaboration. A classifier can be an object, a multi-object, or an actor. Similarly, an *association role* can be an instance of an association between the two classes and is the link that carries messages between the two classifier roles. This link is also limited to its purpose in the collaboration. In other words, the classifier and association roles are relevant only for that collaboration. An object can have different classifier roles in different collaborations; classifiers can exchange different sets of messages across different association roles.

In addition, collaboration diagrams display the messages passed across association roles. Messages are generally instances of class operations. They are numbered to indicate sequence order; they can be subnumbered (for example, 1a., 1b., 1.1.2, 1.1.3, 2.3a.1., 2.3a.2., and so on) either to indicate tasks that occur simultaneously or that are subtasks that achieve a larger task.

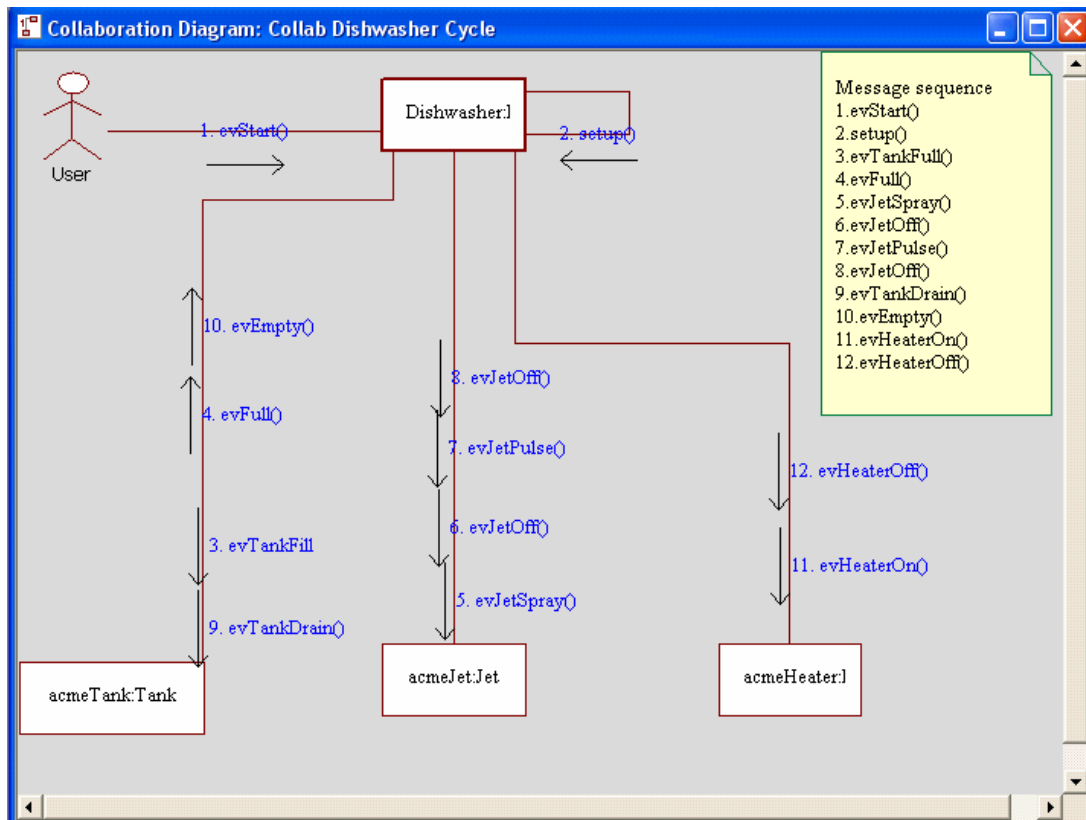
A numbering system that indicates parallelism might look like the following example:

1. Make sandwich.
 - 1a. Get jam.
 - 1b. Cut bread.

A numbering system that indicates subtasking might look like the following example:

1. Make sandwich.
 - 1.1 Get jam.
 - 1.2 Cut bread.
 - 1.3 Spread jam on bread slices.

Classifier roles, association roles, and messages are not displayed in the browser; however, the underlying classes and operations that they realize are displayed. The following figure shows a collaboration diagram.



Collaboration diagram tools

The **Diagram Tools** for a collaboration diagram contains the following tools:

Drawing Tool	Name	Description
	Object	Creates a new classifier role. A classifier role can be an instance of an existing class, a new class that you create in the collaboration diagram, or <Unspecified>, meaning that it is not a realization of a class. This could be useful if you create collaboration diagrams at the high-level analysis stage of system design. For more information, see Classifier roles .
	Multi Object	Creates a classifier role for a set of objects, which means that the classifier role has a set of operations and signals that addresses the entire set of objects, and not just a single object. In other words, the classifier role represents a set of objects that share a common purpose in the scenario or use case described by the collaboration diagram. For more information, see Multiple objects .
	Actor	Creates an actor, which represents an element that is external to the system. A classifier role based on an actor represents a coherent set of operations and messages of an external element when it interacts with system elements during a scenario. For more information, see Actors .
	Link	Draws a message link, or association role, between two classifier roles. Optionally, you can give the association role a name, perhaps to indicate the type of communication that occurs over this link. The association role can be an instance of an existing association between the two classes (from which the classifier roles are realized). For more information, see Link messages and reverse link messages .
	Link Message	Adds a message to the link between two classifier roles. The Message tool creates a message pointing toward the second classifier role in the link. For more information, see Link messages and reverse link messages .
	Reverse Link Message	Adds a reverse message to the link between two classifier roles. The Reverse Link Message tool creates a message pointing toward the first classifier role in the link. For more information, see Link messages and reverse link messages .
	Dependency	Creates a dependency between classifier roles.

The following sections describe how to use these tools to draw the parts of a collaboration diagram. See [Graphic editors](#) for basic information on diagrams, including how to create, open, and delete them.

Classifier roles

A classifier role can be an instance of an existing class, a new class that you create in the collaboration diagram, meaning that it is not a realization of a class. This could be useful if you create collaboration diagrams at the high-level analysis stage of system design.

To create a classifier role:

1. Click the **Object** button .
2. Click-and-drag to create the new classifier role.
3. You can type the classifier role name in the drawing or right-click the object to display the Features window and type the name and additional information.
4. Click **OK**.

By default, a new classifier role has an <Unspecified> **Realization** value. To make it an instance of a <New > or existing package, use the pull-down menu in the Features window. See [Modifying the features of a classifier role](#) more information.

Multiple objects

A multiple object signifies that the classifier role has a set of operations and signals that addresses the entire set of objects, not just a single object. In other words, the classifier role represents a set of objects that share a common purpose in the scenario or use case described by the collaboration diagram. A multiple object can represent multiple instances of one or more classes or object types that share a common purpose in the scenario. As with an object, a multiple object can be <Unspecified> in this representation.

Creating a classifier role for a multiple object involves the same steps as creating a classifier role for a single object.

To create multiple objects:

1. Click the **Multiple Object** button  in the **Diagram Tools** for the collaboration diagrams.
2. In the drawing area, click (or click-and-drag) to create the multiple object.
3. Use the Features window or edit the default name in the drawing.
4. Press **Enter** to save the name change in the drawing.

Actors

An actor represents an element that is external to the system. A classifier role based on an actor represents a coherent set of operations and messages of an external element when it interacts with system elements during a scenario. You can choose an existing actor, create a new one, or set the classifier role to `<Unspecified>`, meaning that the actor is not a realization of an existing actor. This could be useful in diagrams created at the high-level analysis stage of system design. Actors are a stereotype of a class and are defined through a Features window that is, for the most part, the same as that for classes.

If you create a new actor, the Features window opens so you can define the actor.

Creating an actor

To create an actor:

1. Click the **Actor** button .
2. In the drawing area, click (or click-and-drag) to create the actor.
3. Use the Features window or edit the default name in the drawing.
4. Press **Enter** to save the name change in the drawing.

By default, a new actor is an `<Unspecified>` **Realization** value. To make it an instance of a `<New >` or existing package, use the pull-down menu in the Features window. See [Modify the features of an actor](#) for more information.

Links

The **Link** tool draws a message link, or *association role*, between two classifier roles. Optionally, you can give the association role a name, perhaps to indicate the type of communication that occurs over this link. The association role can be an instance of an existing association between the two classes (from which the classifier roles are realized).

The association role of a link can be `<Unspecified>`, meaning that it is an unspecified association. This could be useful in design- and even detailed design-phase collaboration diagrams, because you can portray messages that are not passed through relations, such as communication with local or global variables (objects) or communication with variables passed as parameters of a method.

Association roles are themselves not directional, even if they are assigned a directional association. This is in keeping with the emphasis on message traffic, regardless of which class

initiated the flow. Once you have created an association role, you can draw the messages that go across it.

Note

You can physically move an association role from one set of classifier roles to another, but this is not typical because the connection of the association role to the association is lost. The association of an association role must be between the classes matched to the end classifier roles.

Creating a link

To create a link:

1. Click the **Link** button .
2. Click in a classifier role.
3. Click in another classifier role. The link is drawn between the two classifier roles, and the cursor automatically opens the association role name text box.
4. If wanted, type a name for the association role, then press **Enter**.

Features of links

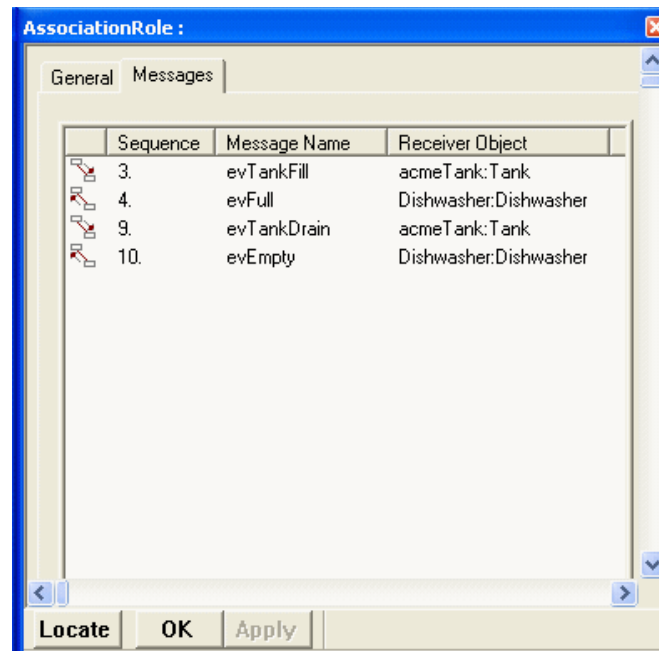
The Features window enables you to change the features of a link, including the role name and association. The **General** tab includes the following fields:

- ♦ **Role Name** specifies the name by which one class recognizes the other class.
- ♦ **Association** specifies the association being instantiated by the link.

Rational Rhapsody allows you to specify a link without having to specify the association being instantiated by the link. Until you specify the association with the pull-down menu, this field is set to <Unspecified>.

- ♦ **Description** allows the user to add more detailed information about the association role.

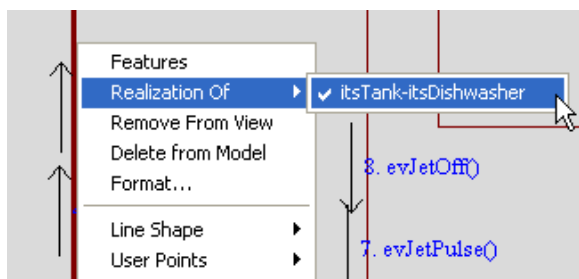
The **Messages** tab of the Features window for the link lists the messages sent across the link, as shown in the following figure.



Changing the underlying association

You can set the association on which the association role is based using the Features window. You can also change the association via the menu, as follows:

1. Click the **Select** button .
2. Right-click the association role and then select **Realization Of**. A menu of available associations is displayed, as shown in the following figure.



3. Highlight one of the associations, then click.

Link messages and reverse link messages

Once a link is created between two classifier roles, you can add messages to it. Link messages are numbered automatically, but can be edited and renumbered, for example, using a subnumbering system.

You need to use two tools to create the link messages in collaboration diagrams:



Link Message creates a message pointing toward the second classifier role in the link.



Reverse Link Message creates a link message pointing in the other direction.

Like classifier and association roles, messages can be `<Unspecified>`, meaning that they are abstract and not realizations of class operations. Link messages can be instances of existing operations of a class or instances of new operations. However, for a link message to realize some operation, the operation must be a method of the class associated with the target of the message.

Messages, whether abstract or instances of operations, have the notation `ReturnValue = MessageName(Arg, Arg, Arg...)`. You can use this notation in the message name when you first create it, or you can fill in these boxes explicitly in the Features window.

Note that a message that is an instance of an operation does not necessarily show the form of the actual call. You can specify just the items of interest in the collaboration. The `ReturnValue` is optional; the function might not return a value, or you might not want to specify the local variable to which the return value applies.

Creating a link message or reverse link message

To create a message:

1. Click either the **Link Message** or **Reverse Link Message** button. The cursor changes to a small arrow pointing upwards.
2. Move the point of the arrow onto the association role, then click with the left mouse button. A text box opens, containing an automatically generated number.
3. Type the name of the message. If wanted, you can change the numbering; the autonumbering will continue from whatever number you specify.

Note: If you edit the number, make sure the numbering sequence ends with a period (.) to clearly delineate it. If the period is missing, Rational Rhapsody will not autonumber the messages correctly.

4. Press **Enter** to complete the name.

By default, the new message is `<Unspecified>`. To make it an instance of a new or existing operation of the target class or actor, open its Features window (see [Modifying the features of a message](#)).

Component diagrams

You use *component diagrams* to create new or existing components, specify the files and folders they contain, and define the relations between these elements. These relations include the following items:

- ♦ **Dependency** shows a relationship in which the proper functioning of one element requires information provided by another element. In a component diagram, a dependency can exist between any component, file, or folder.
- ♦ **Interface** shows a set of operations that publicly define a behavior or way of handling something so knowledge of the internals is not needed. Component diagrams define interfaces between components only.

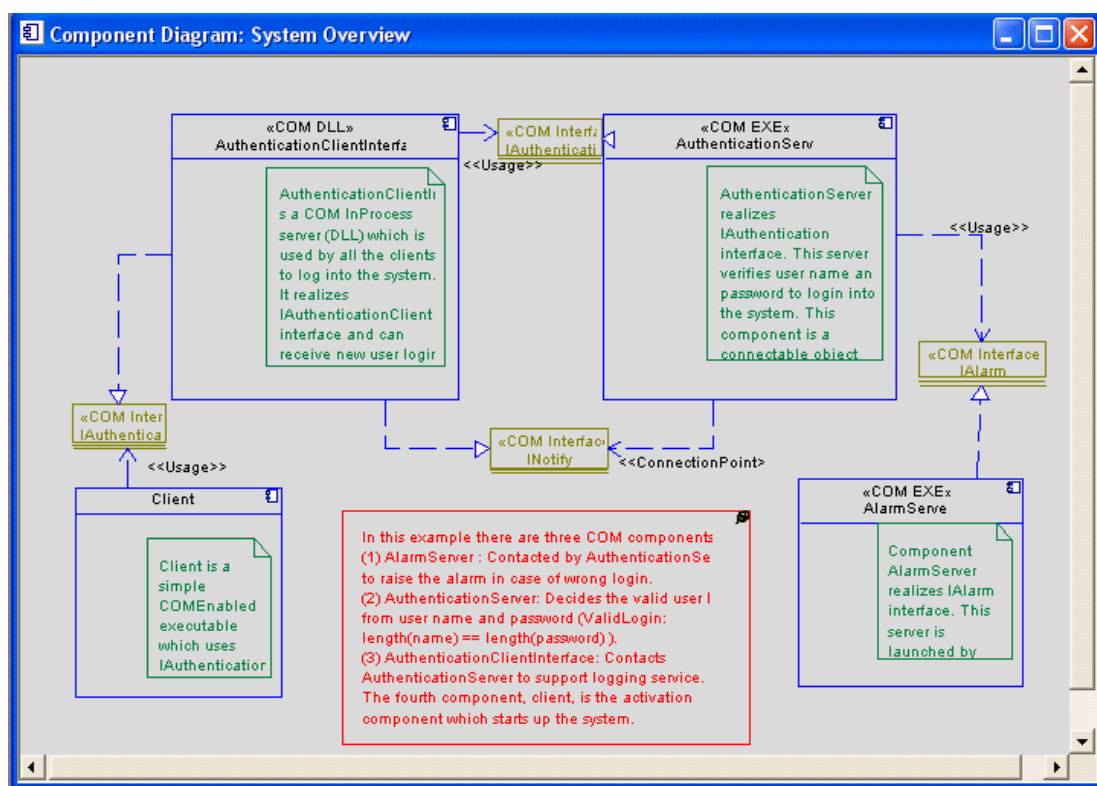
A *component* is a physical subsystem in the form of a library or executable program or other software components such as scripts, command files, documents, or databases. Its role is important in the modeling of large systems that comprise several libraries and executables. For example, the Rational Rhapsody application itself is made up of many components, including the graphic editors, browser, code generator, and animator, all provided in the form of a library.

Component diagram uses

Component diagrams are helpful in defining and organizing the physical file hierarchy of your model. You can assign model elements to be contained in certain files, instead of using the default Rational Rhapsody designations; you can organize files into folders or into components directly and organize folders into components.

One aspect of a component that is not included in a component diagram, but is included in this section, is how to create the configurations that are part of a component. Configurations specify how the component should be built, such as the target environment, initialization needed, and checks to perform on the model before code is generated. See [Configurations](#) for information.

The following figure shows a component diagram.



Component diagram drawing Tools

The **Diagram Tools** for a component diagram contains the following tools.

Drawing Tool	Name	Description
	Component	Specifies all the software code that it comprises: libraries, header files, and any other source files. See Components for more information.
	File	Specifies which model elements are generated in each file, the file names, and the directory paths. See Files for more information.
	Folder Component	Physically organizes files or other folders. See Folders for more information.
	Dependency	Shows when one element depends on the existence of another element. See Dependencies for more information.
	Interface	Creates an interface between components is a set of operations performed by a hardware or software element in the system. See Component interfaces and realizations for more information.
	Realization	Indicates that a component realizes an interface if it supports the interface. Then another component uses that interface.
	Flow	Provides a mechanism for specifying exchange of information between components. See Flows for more information.

The following sections describe how to use these tools to draw the parts of a component diagram. See [Graphic editors](#) for basic information on diagrams, including how to create, open, and delete them.

Elements of a component diagram

Component diagrams contain the following elements:

- ◆ [Components](#)
- ◆ [Files](#)
- ◆ [Folders](#)
- ◆ [Dependencies](#)
- ◆ [Component interfaces and realizations](#)
- ◆ [Flows](#)

The following sections describe these elements in detail.

Components

When you create a component, you specify all the software code that it comprises: libraries, header files, and any other source files. Component diagrams generate code for components that are labeled with the «Executable» or «Library» stereotype.

Creating a component

To create a component:

1. Click the **Component** icon  in the **Diagram Tools**.
2. Do one of the following actions:
 - a. Click once in the diagram to create a component with the default dimensions.
 - b. Click to begin the upper, left corner of the component, drag to the lower right corner, and release.
3. Edit the default name, then press **Enter**.

The new component is displayed in the diagram and in the browser, either in the Component s folder under the main project node, or nested under another component associated with this diagram.

Note: You can draw a component within another component; the browser will display the nested component accordingly.

Features of components

Use the Features window to define a component.

General Features

- ♦ **Name** specifies the name of the component.
- ♦ **L** specifies the label for the element, if any.
- ♦ **Stereotype** specifies the stereotype of the component, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Stereotypes](#) for information on creating stereotypes.

Note: The COM stereotypes are constructive; that is, they affect code generation.

- ♦ **Directory** specifies the root directory for all configurations of the component. This can be the project directory or another directory.
- ♦ **Libraries** specifies any additional libraries to be added to the link. This field is relevant for executables only. Libraries can be off-the-shelf libraries, legacy code, or Rational Rhapsody-generated libraries.
- ♦ **Additional Sources** specifies external source files to be compiled with the generated source files. Rational Rhapsody adds the files to the project makefile.
- ♦ **Standard Headers** specifies header files to be added to `include` statements in every file generated for the project. Specify either a full path or, if only a filename, add a path in the **Include Path** field.
- ♦ **Include Path** specifies the directory in which the include files are located.

Note that this field supports environment variables, such as `$ROOT\Project\ExternalCode`.

- ♦ **Type** specifies the build type. Select **Library**, **Executable**, or **Other**. Rational Rhapsody does not generate code for a component that has build type “other,” nor can such a component be set as the active component. “Other” could be used to designate script files or other non-code files.

Component scope

The **Scope** tab of the Component Features window allows you to specify which model elements should be included in the component.

If you select the **Selected Elements** radio button, you can use the check boxes next to each element to indicate which model elements should be included in the component.

If you select a check box for an element, all of the elements that it contains are included in the component scope (for example, all of the classes in a package). If you would like to be able to select sub-elements individually, right-click the check box of the parent element.

Files

Files owned by a component are compiled together to build the component. You can specify which model elements are generated in each file, the file names, and the directory paths. You can also create a file in the browser and drag-and-drop it into a component diagram.

A file must be nested within a component or a folder.

Note

However, a file cannot contain another element.

Creating a file

To create a file:

1. Right-click a Component in the browser and select **Add New > File**. The File window opens.
2. Begin to define this new file by typing the Name you want to replace the system generated name. Click **Apply** to save the name and keep the window open. The new file name displays in the diagram and in the browser, under the component with which it is associated.

Note: A file must be an element of a component or a folder. If the component diagram is under the project node, it is not yet associated with a component. First create a component, then nest the file by drawing it inside the component. If the diagram is already nested under an existing component, you can draw the file in the “free space” of the diagram editor.

3. Change the remaining fields to define the file as you want.
 - ♦ **Path** specifies where a file should be generated in relation to the configuration directory. If this field is blank, the file is generated in the configuration directory.
 - ♦ **File Type** specifies the type of file that should be generated. A specification file contains the specifications of all elements; an implementation file contains their implementations. They are specified by their suffixes, as follows:

File Type	C++	C	Java
Implementation	.cpp	.c	.java
Specification	.h	.h	N/A

Choose one of the following values:

- **Logical** creates both implementation and specification files.

- **Specification** generates only a specification file. Both specifications and implementations of all elements assigned to this file are generated into this file.
- **Implementation** generates only an implementation file. Both specifications and implementations of all elements assigned to this file are generated into this file.
- **Other** where one file is generated with the name and extension specified in the **Name** field. Both specifications and implementations of all elements assigned to this file are generated into the file.
- ♦ **Elements** lists the elements mapped to a file. Elements that are not explicitly mapped to files are generated in the default files that Rational Rhapsody would normally generate for these elements in the configuration directory.
- ♦ **Environment Settings** where Rational Rhapsody fills in the settings from your environment. The fields are as follows:
 - **Environment** where this read-only field specifies which environment (Microsoft, Solaris2, and so on) is selected for the active configuration. You cannot change the environment for an individual file.
 - **Build Set** where this read-only field specifies the build setting (Debug or Release mode) for the active configuration. You cannot change the build setting for an individual file.
 - **Compiler Switches** specifies the compiler switches for the configuration. Compiler switches default to those used for the configuration, but you can override them for an individual file.
 - **Link Switches** specifies the link switches used to link the active configuration. You cannot change link switches for an individual file.
- ♦ **Description** describes the element. This field can include a hyperlink. See [Hyperlinks](#) for more information.

4. Click **OK**.

Adding an element to a file

To add a package or class to a file:

1. In the window, click **Add Element**. The Add Elements window opens. The **Type** box lists the file type you selected in the file Features window. You can change the file type here if it is not logical. Logical files can contain only logical elements, but other types of files can contain whatever you want. See [The Features window](#) for more information.
2. Select the elements that you want to map to the file. Note that selecting a package automatically maps all its classes to the file.
3. Click **OK**.

Adding text to a file

To add a text element to a file:

1. In the file Features window, click **Add Text**. The File Text Element window opens.
2. If wanted, edit the default name of the text element in the **Name** field.
3. In the **Text Element** field, type the text you want to add to the file. For example, you can add `#ifdef` statements or `#define` statements, headers and footers, or additional comments.
4. In the **Description** field, enter a description of the text element.
5. Click **OK**.

Deleting an element from a file

To delete an element from a file:

1. In the file Features window, select the element to delete.
2. Click **Delete**. Rational Rhapsody asks you to confirm that you want to remove the element from the file.
3. Click **Yes** to delete the element.

Editing an element

To edit an element in a file:

1. In the file Features window, select the element you want to edit.
2. Click **Edit**. The File Text Element window opens.
3. Make the appropriate changes.
4. Click **OK**.

Rearranging elements in a file

You have explicit control over the order in which elements are generated in files. Moving an element up or down in the list means that it will be generated earlier or later in the file.

To rearrange elements in a file:

1. In the file Features window, select the element you want to move.
2. Click **Up** to generate the element earlier in the file, or **Down** to generate the element later in the file.

Component diagram files menu

- ◆ **Add New** allows you to add a new relations, requirement, annotations, or tag to the file.
- ◆ **Generate File** creates the file from the elements assigned to it. The number and type of files generated depends on the file type you have selected. See [The Features window](#).
- ◆ **Edit File** opens the generated files in a text editor.

You can select an external text editor using the `EditorCommandLine` property under `General::Model`. See [Using an External Editor](#) for more information.

Folders

Folders, or directories, help physically organize files. A folder must be nested within a component or another folder. A folder can contain files or folders.

Creating a folder

1. Click the **Folder** tool.
2. Do one of the following actions:
 - a. Click once in the diagram to create a folder with the default dimensions.
 - b. Click to begin the upper, left corner of the folder, drag to the lower, right corner, and release.
3. Edit the default name, then press **Enter**.

The new folder is displayed in the diagram and in the browser under the component or folder with which it is associated.

Note

A folder must be nested within a component or another folder. If the component diagram is under the project node, it is not yet associated with a component. You must first create a component, inside of which you can then draw a folder. If the diagram is already nested under an existing component, you can draw the folder in the “free space” of the diagram editor; it is nested within the component associated with the diagram and is displayed accordingly in the browser.

In the browser, folders are located under the components of which they are part.

Features of folders

The Features window enables you to change the features of a folder, including its name and path.

A folder has the following features:

- ♦ **Name** specifies the name of the folder. The default name is `folder_n`, where *n* is an incremental integer starting with 0.
- ♦ **Path** specifies where the folder should be generated in relation to the configuration directory. Folders are generated as subdirectories under the configuration directory.
- ♦ **Elements mapped to the folder** specifies the elements you want to map to a folder. Rational Rhapsody generates the default types of files it normally generates for these elements in the folder if the elements are not specifically mapped to other files.
- ♦ **Description** describes the folder. This field can include a hyperlink. See [Hyperlinks](#) for more information.

Note that the Features window for folders is available only for folders that you add to the configuration, not for the top folder under the component.

Folders menu

- ♦ **Add New** opens a cascading menu that allows you to add elements to the folder. You can add the following items:
 - **Folder** adds a subdirectory to the current directory.
 - **File** adds a file to the current directory.

Dependencies

A dependency exists when the functioning of one element depends on the existence of another element. A dependency between two components in a component diagram results in an `#include` statement in the makefile for the dependent (or client) component.

Dependencies in component diagrams have the same stereotype values as dependencies created in OMDs. See [Dependencies](#).

In a component diagram, a dependency relation is also used in the definition of a component interface. See [Creating a component interface](#) for more information.

Dependencies appear in the browser under the dependent, or client component.

Component interfaces and realizations

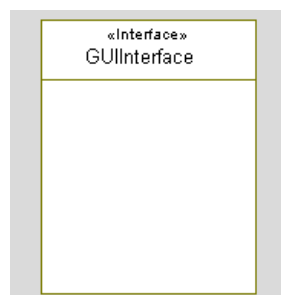
An *interface* between components is a set of operations performed by a hardware or software element in the system. A component realizes an interface if it supports the interface; another component then uses that interface. Interfaces promote design modularity; components are more easily replaceable when they use interfaces instead of directly depending on components.

Component interfaces can be seen only in a component diagram. They cannot be viewed in the browser. A component diagram supports only interfaces between components.

Creating a component interface

To create an interface:

1. Click the **Interface** icon  in the **Diagram Tools**.
2. Click once in the diagram, or click-and-drag to create the component interface.
3. By default, Rational Rhapsody creates an interface named `Interface_n`, where *n* is an integer value starting with 0. If wanted, rename the interface. The following example shows a component interface. Rational Rhapsody adds the `«Interface»` stereotype automatically.



Flows

Flows provide a mechanism for specifying exchange of information between system elements at a high level of abstraction. This functionality enables you to describe the flow of data and commands within a system at a very early stage, before committing to a specific design. As the system specification evolves, you can refine the abstraction to relate to the concrete implementation.

Component configurations in the browser

In the browser, components are displayed under the project main node or other components. There are a number of component features that you can set via the component menu in the browser that you cannot do in the component diagram. For example, in the browser you can set a component as the active configuration or define the configuration settings for the component.

When you highlight a component in the browser, the Features window opens. See [Features of components](#) for more information.

Component options

When you select a component in the browser, the menu includes the following component-specific options:

Add New opens a cascading menu that allows you to add the following items to the component:

- ♦ **Requirement** allows you to add a textual annotation that describes the intent of the component. Requirements are part of the model and are therefore displayed in the browser.
- ♦ **Relations** contains -
 - **Dependency** specifies the element on which the component depends, such as a package that is not part of the component.
 - **Derivation** adds a requirement that was derived from another
 - **Hyperlink** adds internal links to Rational Rhapsody elements or external links to a URL.
- ♦ **Annotations** contains -
 - **Constraint** defines a semantic condition or restriction.
 - **Comment** allows you to add a textual annotation that does not add semantics, but contains information that might be useful to the reader and is displayed in the browser.
 - **Controlled File** allows you to add a file or reference purposes that was produced in other programs, such as Word or Excel. These files become part of the Rational Rhapsody project and are controlled by it.
- ♦ **Tag** holds model information relating to the domain or platform.

Delete from Model

Unit contains **Save as Component Diagram Component**, **Unload Component Diagram Component**, and **Edit Unit**.

Active component

The active component is the one built when you make the code. The icon for the active component includes a red checkmark.



Setting the active component

To make a component active:

1. Select the component from the list on the toolbar.
2. In the browser, right-click the component and select **Set as Active Component**.

When you change the active component, the most recent active configuration within the component becomes the active configuration and is listed in the Current Configuration list in the **Code** toolbar.

Note

To become the active component, a component must be set to either the **Executable** or **Library** build type.

Configurations

If a component is a physical subsystem, as in a communications subsystem, a configuration (module) specifies how the component is to be produced. For example, the configuration determines whether to compile a debug or non-debug version of the subsystem, whether it should be in the environment of the host or the target (for example, Windows NT versus VxWorks), and so on.

A component can consist of several configurations. For example, if you want to build a VxWorks version and a pSOSystem version of the same component, you would create two configurations under the component, one for each operating system. The decision as to whether these should be two separate components or two configurations within the same component depends on whether you want to compile them differently, or whether there is some logical variation between them. Creating two separate components would require maintaining two separate implementation views, whereas using separate configurations would not.

For information on setting configuration parameters, see [Features of configurations](#).

Configuration menu

If you right-click a configuration in the browser, the menu contains the following configuration-specific options:

- ♦ **Set as Active Configuration** makes the configuration active.
- ♦ **Edit Makefile** enables you to edit the makefile generated for the configuration in a text editor.
- ♦ **Edit Configuration Main File** edits the main file generated for the component. This option is available for an executable or library component that has at least one package with a global instance in its scope. See [Making permanent changes to the main file](#) for more information.
- ♦ **Generate Configuration Main and Make Files** sets the configuration as the active configuration and generates the main file and the makefile.
- ♦ **Generate Configuration** makes the configuration active and generates it.
- ♦ **Build Configuration** builds the active configuration.

Setting the active configuration

The active configuration is the one generated when you generate code, unless you are generating selected classes. The active configuration, or the current configuration, displays in a list on the **Code** toolbar. You can change the active configuration using either the toolbar or the menu for the configuration.

To set the active configuration:

- ♦ Select the configuration from the Current Configuration list on the **Code** toolbar.
- ♦ Select the configuration in the browser. Right-click the configuration, and select **Set as Active Configuration** from the menu.

The name of the new active configuration is displayed in the Current Configuration list in the **Code** toolbar. In addition, the component that owns this configuration becomes active and is displayed in the Current Component list (see [Active component](#)).

Features of configurations

The Features window for a configuration contains the following tabs:

- ♦ [General tab](#)
- ♦ [Initialization tab](#)
- ♦ [Settings tab](#)
- ♦ [Checks tab](#)
- ♦ [Relations tab](#)
- ♦ [Tags tab](#)
- ♦ [Properties tab](#)

These tabs display configurable features and are described in detail in the following sections.

General tab

The **General** tab allows you to define general information for the configuration.

- ♦ **Name** specifies the name of the configuration. The default name for configurations is `configuration_n`, where *n* is an incremental integer starting with 0.
- ♦ **Description** describes the configuration (for example, the target environment).

Initialization tab

The **Initialization** tab allows you to specify which instances to initialize and whether to generate code for actors.

- ♦ **Initial instances** adds code to the main program to instantiate only those packages, classes, and actors that you specify. The possible values are as follows:
 - **Explicit** instantiates only the selected elements.
 - **Derived** instantiates the selected elements and any others to which these are related, either directly or indirectly.

For example, if class A is selected, and class B is related to A, B is added to the derived scope. If C is related to B, C is also added to the derived scope, and so on.

Two elements are related if there is a dependency or relation between them, use types of the other element, or use events defined in the other element.

- ♦ **Generate Code For Actors** generates code for the actors specified in the **Initial Instances** box. See [Generating Code for Actors](#) for more information about this feature.
- ♦ **Initialization code** where you type any user code you want to use to instantiate initial instances or to initialize any other elements. This code is added to the main program after any automatically generated initialization code and before the main program loop.

Settings tab

The **Settings** tab allows you to specify numerous settings for the configuration.

Note

The values of each of these fields are appended to the settings fields of the component that owns the configuration.

The **Settings** tab contains the following fields:

- ◆ **Directory** specifies the root directory for files generated for the configuration. This box is available only if the **Use Default** option is not checked. You can specify either a full path or a partial path that uses the current directory as the starting point.
- ◆ **Use Default** check this box to use the default directory for the configuration. The default location is named after the configuration and is a subdirectory of the project directory.
- ◆ **Libraries** specifies additional off-the-shelf, legacy code, or Rational Rhapsody-generated libraries to be added to the link. This box is relevant only for executables.
- ◆ **Additional Sources** specifies the external source files to be compiled with the generated sources. Rational Rhapsody adds these files to the project makefile.
- ◆ **Standard Headers** specifies the header files to be added to `#include` statements in every file generated for the project. Specify a full path or the file name. If you specify only the file name in this field, specify the directory in the **Include Path** field.
- ◆ **Include Path** specifies the directory in which the include files for a configuration are located. The include path is added to the makefile generated for a configuration. For example, if you set the include path to `d:\Rhapsody\MMM`, the following code is generated in the makefile:

```
INCLUDE_PATH= \  
$(INCLUDE_QUALIFIER)d:\Rhapsody\MMM
```

- ◆ **Instrumentation** specifies whether the executable will have animation or tracing capabilities, or neither. Select the appropriate value from the **Instrumentation Mode** list.

Click the **Advanced** button to specify the *instrumentation scope*, which determines the set of Rational Rhapsody classes, packages, and actors that are instrumented in the associated configuration. This functionality enables you to enable or disable animation of classes (or entire packages) without changing the model elements themselves.

See [Using selective instrumentation](#) for more information.

- ◆ **Webify** specifies whether to Web-enable the configuration. See [Managing Web-enabled devices](#) for more information.

Note: You cannot webify a file-based C model.

- ◆ **Time Model** specifies real or simulated time. With real-time emulation, timeouts and delays are computed based on the system clock. With simulated time, a virtual timer

orders timeouts and delays, which are posted whenever the system completes a computation.

- ♦ **Statechart Implementation** specifies whether the statechart implementation is Reusable or Flat (the default). The reusable model implements states as classes, whereas the flat model implements states as simple enumerated types. Reusable is preferable for models with deep class inheritance hierarchies, whereas flat is preferable for models with shallow or no inheritance.
- ♦ **Environment Settings** specifies the following information about the target environment:
 - **Environment** specifies the target environment
 - **Build Set** specifies whether the generate a debug or non-debug (Release) version of the executable
 - **Compiler Switches** specifies the compiler switches applied by default when compiling each file
 - **Link Switches** specifies the link switches applied by default when linking the compiled code
 - **Additional Settings** This button allows you to integrate CodeTEST[®] with Rational Rhapsody in C and C++, if you have CodeTEST on your system. Otherwise, this button is unavailable.

When you click this button, Rational Rhapsody displays the Additional Settings window. This window contains the following fields:

- **With Applied Microsystems CodeTEST** enables (select the check box) the integration with CodeTEST.
- **CodeTEST settings** enables you to edit the compilation switches that will be added to the CodeTEST instrumentation line in the generated makefile. The value of the CodeTEST settings field corresponds to the value of the `<lang>_CG::VxWorks::CodeTestSettings` property.

Checks tab

The **Checks** tab enables you to specify which checks to performed on the model before generating code. See [Checks](#) for detailed information on checks performed by Rational Rhapsody.

Relations tab

The **Relations** tab lists all the relationships (dependencies, associations, and so on) the configuration is engaged with. See [Define relations](#) for more information on this tab.

Tags tab

The **Tags** tab lists the available tags for this configuration. See [Profiles](#) for detailed information on tags.

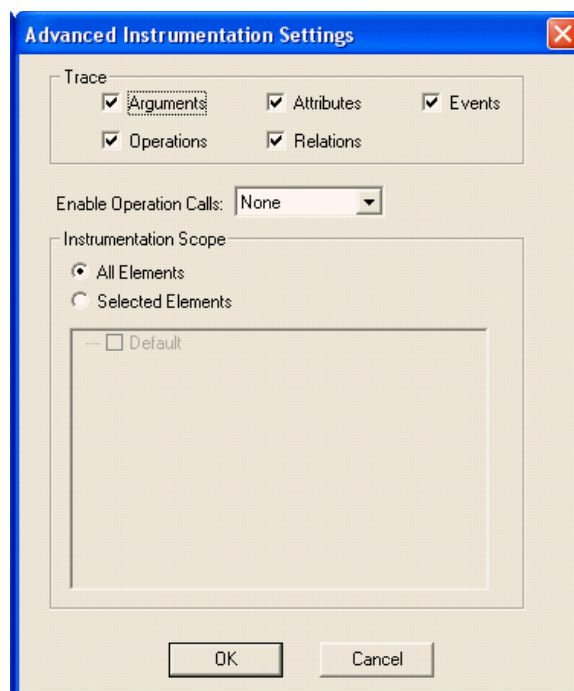
Properties tab

The **Properties** tab enables you to set properties that affect the configuration. For more information about using the Rational Rhapsody properties, see [Properties](#).

Using selective instrumentation

The window for selective instrumentation enables you to select the specific model elements and element types to be instrumented for animation or tracing in the given configuration. Using this functionality, you can use partial animation or tracing without changing the properties of the specific model elements.

When you click the **Advanced** button in the **Instrumentation** group, the Advanced Instrumentation Settings window opens, as shown in the following figure.



Using this window, you can easily control whether instrumentation is available for model elements, operations, and classes and packages for each configuration. The base model (stored in the development tree) could be non-instrumented: to validate parts of the model, you would simply change the animation settings in this window to enable or disable instrumentation.

The window contains the following fields:

- ♦ **Trace** determines whether tracing is available for the different model element types (arguments, operations, attributes, relations, and events). This is equivalent to setting the

Animate properties for the metaclass for the configuration.

For example, clearing the **Operations** check box sets the `CG::Operation::Animate` property for the configuration to `Cleared`, so animation/tracing will not monitor operations. You can override this behavior for a specific element by overriding the property at the element level. For example, to monitor operations in a specific package after clearing the **Operations** check box, set `CG::Operation::Animate` to `Checked` for the specific package.

By default, all model types are selected for instrumentation.

- ♦ **Enable Operation Calls** specifies whether you can launch operation calls from the **Animation** toolbar. The possible values are as follows:
 - **None** means operation calls cannot be launched.
 - **Public** means only public methods can be started.
 - **Protected** means only protected methods can be started.
 - **All** means all operation calls can be launched, regardless of visibility.
- ♦ **Instrumentation Scope** specifies which model elements (classes, packages, and actors) to animate. By default, all model elements are selected.

When the **All Elements** radial button is selected, the behavior is as follows:

- Tree control is disabled. Tree control is available when you click **Selected Elements**.

The tree view contains all the classes, actors, and packages in the scope of the component whose `<lang>_CG::<Metaclass>::Animate` property is set to `Checked`. Note that external elements (`UseAsExternal` is `Checked`) cannot be in the scope of the component. When you select a package in this tree, you also select all its aggregated classes and actors.

- All the elements in the code generation scope are instrumented, unless their `Animate` property is set to `Cleared`.

The following table shows how the instrumentation scope and the `Animate` property determine whether an element is instrumented.

Value of the Animate Property	Set in the Instrumentation Scope?	Will the Element be Instrumented?
Checked	Yes	Yes
Checked	No	No
Cleared	Yes	No
Cleared	No	No

Note the following behavior:

- ◆ If the **Animate** property is set to **Cleared**, it applies to all configurations, regardless of the instrumentation scope.
- ◆ If you change the instrumentation scope, all the source files of the component are regenerated.
- ◆ When you select a class in the package, it is implied that the entire package is instrumented (including all the events, types, and so on) even if the class does not use them.

Making permanent changes to the main file

1. In the browser, right-click the component whose main file you want to edit.
2. Select **Add New > File** to create a new file.
3. Name the new file (for example, myMain).
4. Open the Features window for myMain and set the **File Type** field to Logical or Implementation.
5. Select the **Properties** tab.
6. Click **OK**.
7. Right-click the active configuration for the component (used to build the application) and then select **Edit Configuration Main File**.
8. Copy the contents of this file, including the header files.
9. Open the Features window for myMain and click **Add Text**.
10. Paste the code from the Rational Rhapsody-generated main file into the **Text Element** field.
11. Customize the code as wanted.
12. Click **OK**.
13. Set the following properties for the configuration:
 - a. Set `CG::Configuration::MainGenerationScheme` to `UserInitializationOnly`. This property controls how the main is generated.
 - b. Set `<lang>_CG::<Environment>::EntryPoint` to myMain. This property specifies the name of the main program.

Now when you compile the application, Rational Rhapsody will compile your customized main instead of generating a new one.

Creating components under a package

When you create a Rational Rhapsody project, a component called *DefaultComponent* is created directly beneath the project level. You can also create additional components at this hierarchical level.

It is also possible to create a component as part of a package in the model. One of the advantages of this approach is that if you only want to generate code for a specific package, you only have to check out that package.

You can add a component to a package using any of the following methods:

- ♦ Create a new component in the package by right-clicking the package in the browser and selecting **Add New > Component**.
- ♦ Move an existing component in the browser to the package.
- ♦ Draw a component in a component diagram that is located under a package.
- ♦ Create a new component in a package using the Rational Rhapsody API.

When a component is created under a package, its default scope is the package to which it belongs.

Like other components, components that belong to packages can be assigned to be the active component for the project. When you create a new component in a package, it automatically becomes the active component for the project.

Note

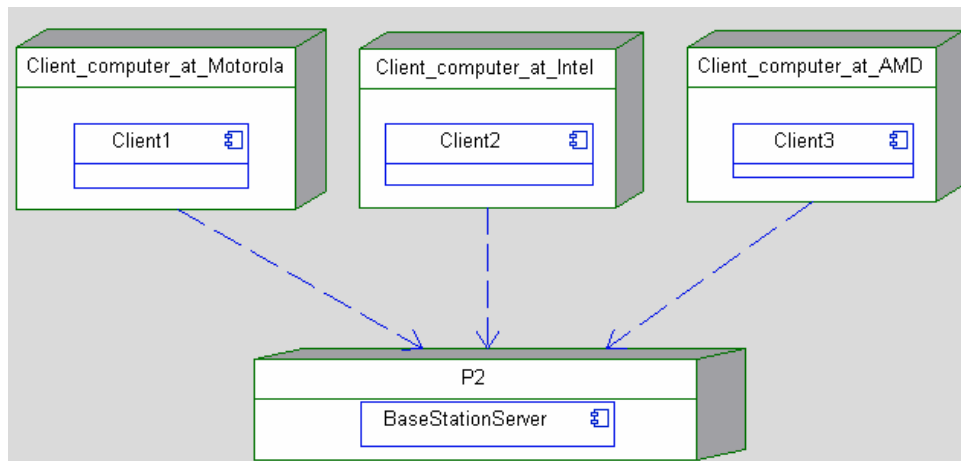
If you draw a component diagram under a package (rather than under the project), keep in mind that then it is not possible to draw a new folder on the diagram. This is because the folder element cannot be contained under a package.

Deployment diagrams

Deployment diagrams show the configuration of run-time processing elements and the software component instances that reside on them. Use deployment diagrams to specify the run-time physical architecture of a system.

Deployment diagrams are graphs of nodes connected by communication associations. Component instances are assigned to run on specific nodes during program execution. Relation lines represent communication paths between nodes.

The following figure shows a deployment diagram.



Opening an existing deployment diagram

To open an existing deployment diagram in the drawing area:

1. Double-click the diagram name in the browser.
2. Click **OK**. The diagram opens in the drawing area.

As with other Rational Rhapsody elements, use the Features window for the diagram to edit its features, including the name, stereotype, and description. See [The Features window](#) for more information.

Deployment diagram drawing tools

The **Diagram Tools** for a deployment diagram contains the following tools:

Drawing Tool	Name	Description
	Node	Represents devices or other resources that store and process instances during run time. See Nodes for more information.
	Component	Represents executable processes, objects, or libraries that run or reside on processing resources (nodes) during program execution. See Component instances for more information.
	Dependency	Represents a requirement by one component instance of information or services provided by another. See Dependencies for more information.
	Flow	Describes the movement of data and commands within a system. See Flows for more information.

The following topics describes how to use these tools to draw the parts of a deployment diagram. See [Graphic editors](#) for basic information on diagrams, including how to create, open, and delete them.

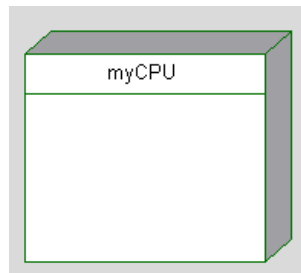
Nodes

Nodes represent devices or other resources that store and process instances during run time. For example, a node can represent a type of CPU. A node can be owned only by a package. In addition, nodes cannot be nested inside other nodes. Nodes can contain component instances.

Note

In Rational Rhapsody, nodes represent UML node instances.

The graphical symbol for a node in the UML is a three-dimensional cube with a name, such as the name of a processor.



Creating a node

You can create a node using the **Node** tool, Edit menu, or browser.

To use the **Diagram Tools** to create a node:

1. Click the **Node** icon  in the **Diagram Tools**.
2. Click or click-and-drag with the mouse to place the node on the diagram. Rational Rhapsody creates a node symbol with the default name of node_n, where *n* is an incremental integer starting with 0.

To use the Rational Rhapsody browser to create a node:

1. Depending on the method you want to use:
 - Right-click a package in the browser and select **Add New > Node**, or
 - Right-click a node category and select **Add New Node**.
2. Edit the default name of the new node.
3. With both the browser and the deployment diagram editor in view, click-drag-and-drop the node onto the diagram.

You can click-and-drag any node that exists in the browser to add it to a deployment diagram.

Changing the owner of a node

To change the package that owns a node:

1. In the Rational Rhapsody browser, select the node.
2. Drag the node from its current package to a new package.

Designating a CPU type

Nodes drawn in deployment diagrams represent specific node instances, rather than general node types. As such, a node should be given a specific name such as `myPersonalIntelPentium`, which describes a specific processor, rather than a general name such as `IntelPentium`, which can apply to many different processors of the same type.

1. Edit the name of the node, replacing the default name of `instance_n` with the name of a type of CPU (for example, `AMD Duron`).
2. Press **Enter**, or click outside the edit box, to terminate editing of the node name.

Features of nodes

Use the Features window for a node to change the features of a node, including its type and the event to which the reception reacts. A node has the following features:

- ♦ **Name** specifies the name of the node. The default name is `node_n`, where *n* is an incremental integer starting with 0.
- ♦ **Stereotype** specifies the stereotype of the element, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Stereotypes](#) for information on creating stereotypes.

Component instances

Component instances represent executable processes, objects, or libraries that run or reside on processing resources (nodes) during program execution. They are represented by the UML component symbol: a box with two small rectangles on the left side.

A component instance is an instance of a component type. Unlike components, there is no special naming convention for component instances. Drawing a component instance inside a node indicates that the component instance lives or runs on that node during run time.

Adding a component instance

You can add a component instance to a deployment diagram using the **Component** icon or the Rational Rhapsody browser.

To use the **Diagram Tools** to add a component instance:

1. With a node already drawn on your deployment diagram, click the **Component** icon  in the **Diagram Tools**.
2. Draw the component instance inside the node. Rational Rhapsody creates the component instance inside the selected node.
3. Edit the default name of the component instance, then press **Enter**.

Note that the component type is not part of the name for the component instance. The full name of a component instance is the same as it displays in the browser.

4. Assign the component instance a type by opening its Features window and setting an existing component in the **Component Type** box.

Note

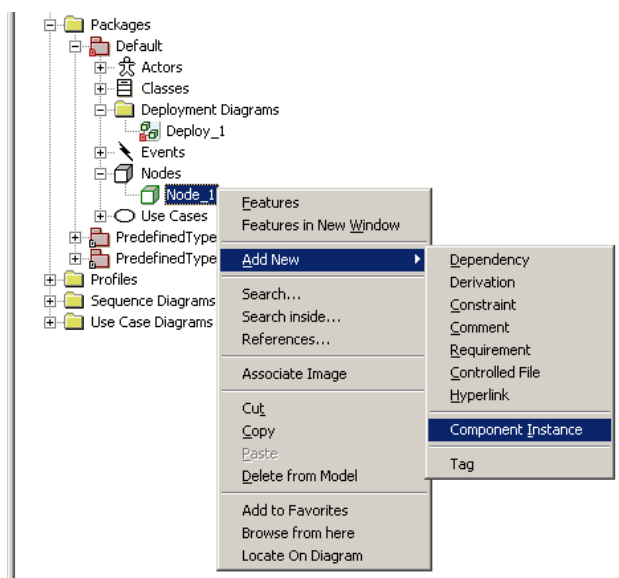
You cannot draw a component instance outside of a node in a deployment diagram.

You can populate a deployment diagram with component instances by dragging them from the Rational Rhapsody browser onto the diagram. Rational Rhapsody creates a component instance based on the selected node.

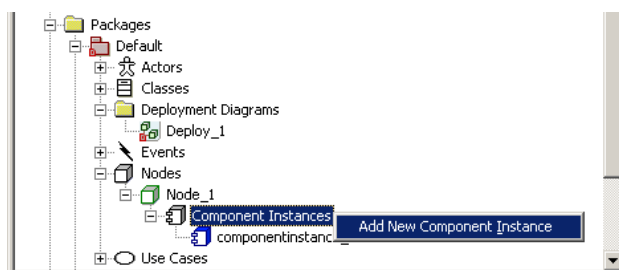
To use the Browser to add a component instance:

1. Depending on the method you want to use:

- Right-click a node in the browser and select **Add New > Component Instance**, as shown in the following figure, or



- Right-click a component instance category, and select **Add New Component Instance**.



2. Edit the default name of the component instance.
3. With both the browser and the deployment diagram editor in view, click-drag-and-drop the component instance from the browser onto the diagram.

Moving a component instance

During the course of development, you might decide that you want a component instance to run on a different node.

To move a component instance from one node to another:

1. In the Rational Rhapsody browser, select the component instance you want to move.
2. Use your mouse to drag the component instance to the new node.

Features of component instances

The Features window allows you to change the features of a component instance including its name and type.

- ♦ **Name** specifies the name of the component instance. The default name is `componentinstance_n`, where *n* is an incremental integer starting with 0.
- ♦ **Stereotype** specifies the stereotype of the element, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Stereotypes](#) for information on creating stereotypes.
- ♦ **Component Type** specifies the component type. This list includes all the components that exist in the model.
- ♦ **Node** specifies the name of the owning node. This box is read-only.

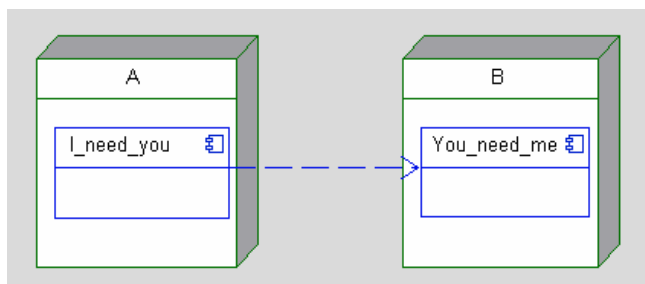
Dependencies

A dependency represents a requirement by one component instance of information or services provided by another. Dependencies can also be drawn between nodes. You can add a dependency using the **Dependency** tool or the Rational Rhapsody browser.

Adding a dependency

To add a dependency using the **Diagram Tools**:

1. Click the **Dependency** button  in the **Diagram Tools**.
2. Click the dependent node or component instance.
3. Click the node or component instance that is being depended on. The arrow at the end of the dependency points to the selected instance.



To add a dependency using the Rational Rhapsody browser:

1. Right-click the dependent node or component instance in the browser and then select **Add New > Dependency**.
2. On the Add Dependency window, select the node or component instance that is being depended on from the **Depends on** list. The dependency is added as a relation under the component instance. Click **OK**.
3. Click the **Dependency** icon.
4. Draw the dependency in the deployment diagram.

Note

You cannot add dependencies to a diagram by dragging.

Flows

Flows and flowitems provide a mechanism for specifying exchange of information between system elements at a high level of abstraction. This functionality enables you to describe the flow of data and commands within a system at a very early stage, before committing to a specific design. As the system specification evolves, you can refine the abstraction to relate to the concrete implementation.

See [Flows and flowitems](#) for detailed information about flows and flowitems.

Assigning a package to a deployment diagram

Like other diagrams, deployment diagrams belong to a package. When you create a deployment diagram in the browser at the project level, Rational Rhapsody assigns the diagram to the default package and displays it in the Deployment Diagrams folder located at the project level. To create a deployment diagram in a particular package, first select the package, then create the new diagram. The nodes and component instances in the deployment diagram belong to the package of that diagram.

To assign a deployment diagram to a different package:

1. In the browser, right-click the deployment diagram and select **Features** to open the Features window.
2. From the **Default Package** list, select the package to which you want to assign the deployment diagram.
3. Click **OK**.

Diagrams located in the project-level **Deployment Diagrams** category remain in that category, but any nodes and component instances that the diagram contains are listed under the selected package.

Checks

Before generating code, Rational Rhapsody automatically performs certain checks for the correctness and completeness of the model. You can also perform selected checks at any time during the design process. These predefined checks, also known as internal checks, are provided with the Rational Rhapsody product. For a list of these predefined internal checks, see [List of Rational Rhapsody checks](#).

In addition, you can create checks that you code and customize to meet your needs. These user-defined checks are also known as external checks because they are not part of the set of predefined internal checks.

Both types of checks are displayed in the Rational Rhapsody GUI, as described in this section.

For more specific information about external checks, see [User-defined checks](#).

Checker features

The checker works on either the active configuration or selected classes. It generates a list of the errors and warnings found in the model, with errors listed first. Errors prevent code generation from proceeding, while warnings draw your attention to unusual conditions in the model that do not prevent code generation. If there are no errors or warnings, the checker generates a message stating that all checks were completed successfully.

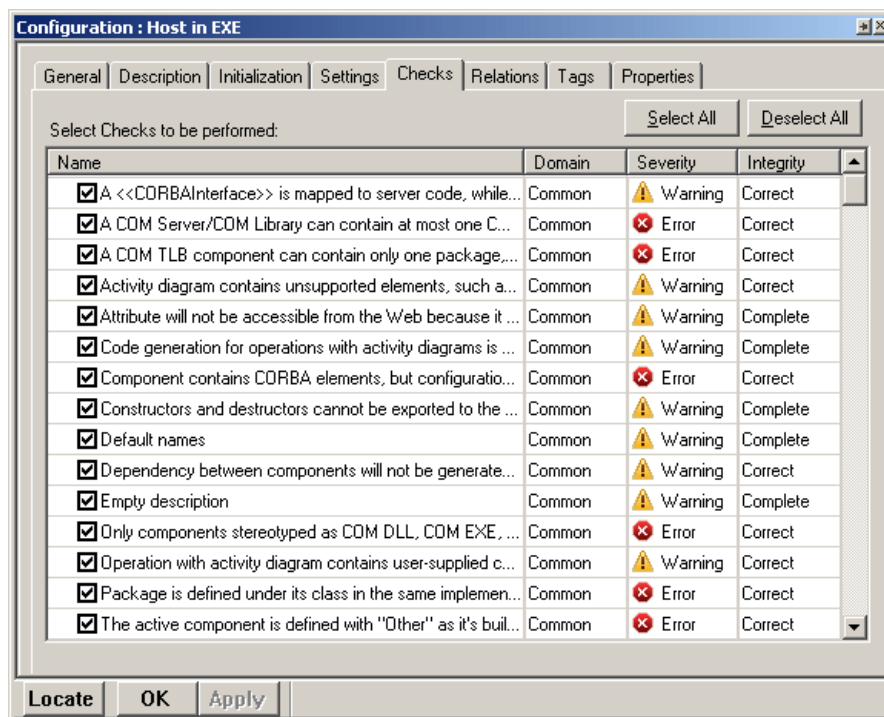
When you double-click a message, the checker opens the location in the model where the offending element or statement can be found, with the source of the error highlighted.

Note

The checker verifies the structural model by checking OMDs, and the behavioral model by checking statecharts. These are the main constructive diagrams in the model.

The Checks tab

The **Checks** tab of the Features window for a configuration, as shown in the following figure, lists all the available checks.



The **Checks** tab contains the following columns:

- ♦ **Name** describes the check to be performed. For example, **Attribute named the same as a state** checks whether an attribute and a state have the same name. By default, all possible checks are selected. To not include a check, clear the applicable check box. If all the checks are not selected and you want to do so, click the **Select All** button. To clear all of the checks (to make it easier to select only certain checks), click the **Deselect All** button.

Note the following that when a name is very long, you can move your mouse pointer over the name to see its full name in a tooltip.

- ♦ **Domain** specifies the area of the model that is searched. You can select checks that belong to one domain or another to limit the scope of the checks. The possible values are as follows:
 - **Class Model** searches the structural part of the model.
 - **Statechart** searches the behavioral part of the model.
 - **Common** searches both the structural and behavioral parts of the model. For example, **Default names** checks for default names in either classes or states.
 - There might be other domains that are from user-defined external checks.
- ♦ **Severity** specifies whether the condition being checked for constitutes an error , a warning , or is informational .

The following table lists the errors that cause code generation to stop.

Name conflicts	- An attribute is named the same as a state. - A class is named the same in a different subsystem. - An event and a generated state class have conflicting names. - An event is named the same as a class.
Other errors	- An OR state exists with no default state. - A fork to non-orthogonal states. - A join from non-orthogonal states. - A reference to an unresolved event. - A reference to an unresolved relational class. - A reference to an unresolved superclass. - A precondition for symmetric links failed.

- ♦ **Integrity** specifies whether the check has to do with the correctness or completeness of the model.

To sort by a column, click the column header.

Specifying which checks to run

You can control which checks are done. Note that Rational Rhapsody automatically performs the predefined code generation checks when you do a check model.

To specify which checks to run:

1. Open your model.
2. Set the configuration for the model whose code you want to check to be the active configuration. (See [Setting the active configuration](#).)
3. Open the Features window for the active configuration and select the **Checks** tab. Do either of the following actions:
 - ♦ Choose **Tools > Check Model > Configure**.
The Features window opens with the **Checks** tab selected.
 - ♦ From the main Rational Rhapsody browser, double-click the active configuration and select the **Checks** tab.
4. Depending on what you want to do:
 - ♦ To select all the checks, click the **Select All** button.
 - ♦ To unselect all the checks so that you can more easily select the checks that you do want, click the **Deselect All** button and then select the checks you do want to perform.
 - ♦ Select and clear the check boxes next to the checks as you want.
 - ♦ Right-click one or more checks and select **Select**, **Deselect**, **Invert Selection**, as applicable.
5. Click **OK**.

Checking the model

Before checking the model, be sure you have done the steps in [Specifying which checks to run](#).

To start checking the model:

1. If you want to perform checks only on selected packages or classes, select those elements on the Rational Rhapsody browser.

Note: You can use **Shift+Click** to select contiguous elements and **Ctrl+Click** to select non-contiguous elements.
2. Select **Tools > Check Model** and select one of the following options, when available:
 - ♦ The active configuration
 - ♦ **Selected Elements** to perform the checks on the selected elements only
3. Review the results on the **Check Model** tab of the Output window. See [Check Model tab](#).
4. For errors and warnings, you can double-click a message on the **Check Model** tab and Rational Rhapsody will open to the relevant model element (for example, the Features window for an association) or to the code on which you can make corrections or view the item more closely.

Checks tab limitations

There must be at least one check selected. Even if you clear all the check boxes and click **Apply** and **OK**, the next time you open the Features window for the active configuration, you will see that all checks on the **Checks** tab will be selected.

User-defined checks

You can create user-defined checks, which are also known as external checks, which are customized checks that you code yourself. System profiles, for example, often require domain-specific checks.

Just as the predefined internal checks provided by Rational Rhapsody, you can define if external checks are called from code generation or not. In addition, you can define on which metaclasses external checks will be executed.

You can implement user-defined external checks through the Rational Rhapsody API and use the GUI already in place in Rational Rhapsody to run them (the **Checks** tab as described in [The Checks tab](#)). Whether it is an internal check or an external check, the checks are performed and their results displayed through the same GUI in Rational Rhapsody.

Creating user-defined checks

Rational Rhapsody provides an API for registering, enumerating, and removing user-defined external checks through the use of the COM API for C++ and VB users, and the Java API for Java users. COM callbacks (connection points) allow you to open user-defined code when checks are executed. This capability is available for those users who use the COM API or Java API so that you can add, execute, or remove user-defined checks.

You decide which metaclasses (or new terms) you want to check, and your checks are called to check elements of whichever metaclasses you decided upon.

For example, for COM API users to create a user-defined check:

1. Implement a class defined from the interface `IRPExternalCheck` in the COM API.
2. Register this class using the `IRPExternalCheckRegister` Add method. You get this singleton via a method on `IRPApplication`.
3. You must implement `IRPExternalCheck` on your client machine.

The following table lists the methods in the interface that you must implement for a user-defined check.

Method	Explanation
<code>GetName()</code>	Returns the name attribute as a string.
<code>GetDomain()</code>	Returns the domain attribute as a string.
<code>GetSeverity()</code>	Returns one of the predefined severity strings: Error, Warning, or Info.
<code>IsCompleteness()</code>	Returns TRUE if the check is for completeness, otherwise FALSE (the check is for correctness)
<code>ShouldCallFromCG()</code>	Returns TRUE if this check should be called when the user generates code
<code>GetRelevantMetaclasses()</code>	Returns a list of relevant metaclasses and/or new terms. The check will be started by Rational Rhapsody for any element in the scope of the current configuration whose metaclass is returned.
<code>Execute()</code>	Called by Rational Rhapsody in order to run the check. This routine returns TRUE if the check passes or FALSE if the check failed. It has two parameters: • The first parameter provided by Rational Rhapsody is the <code>IRPModelElements</code> that the check should be run on (its metaclass is in the checks <code>GetRelevantMetaclass()</code> list). • The second parameter, returned by the check when relevant, is a collection of <code>IRPModelElements</code> that Rational Rhapsody will highlight should the check fail.

Removing user-defined checks

To remove a user-defined check, remove your class using the `IRPExternalCheckRegister` Remove method. You get this singleton via a method on `IRPApplication`.

Deploying user-defined checks

This capability is available for those C++ and VB users who use the COM API, and Java users who use the Java API, so that you can add, execute, or remove user-defined checks. You provide the code in a COM client.

- ♦ If using VB, the client is an EXE file.
- ♦ If using C++, the client is an EXE or DLL file.
- ♦ If using Java, the client is a CLASS or JAR file.

Rational Rhapsody uses the plug-in mechanism to load your code, typically with a HEP file or INI file. Typically, you added the HEP file next to the relevant project or file. For example, a user wanting to write a Java plug-in would write the plug-in using the Rational Rhapsody Java API and provide a HELP file similar to the one in the following example.

```
[Helpers]
numberOfElements=1
name1=ExternalChecks
JavaMainClass1=JavaPlugin.ExternalChecks
JavaClassPath1=$OMRROOT\..\DoDAF
```

Sample check projects are provided for Java and VB in the **ExternalChecksSample** subfolder of your Rational Rhapsody installation path (for example, <Rational Rhapsody installation>\Samples\ExtensibilitySamples\ExternalChecksSample).

External checks limitations

Rational RhapsodyCL is not supported because it does not support the COM API.

List of Rational Rhapsody checks

The following table lists all the predefined internal checks that can be performed by Rational Rhapsody. This table lists all the checks for all versions of Rational Rhapsody (Rational Rhapsody in C, Rational Rhapsody in Ada, and so on) so all the checks listed here will not necessarily appear in your version of the product. In addition, your system might have user-defined external checks. For more information about these types of checks, see [User-defined checks](#).

For ease-of-use, the table lists the checks by name in alphabetical order.

- ♦ The **Correct** column marks the checks for correctness, whereas checks for completeness are marked in the **Complete** column. In these columns, these are the possible values:
 - **E** for Error message
 - **I** for Informational message
 - **W** for Warning message

For example, the **Default names** check has a **W** in the Complete column, which means it is a warning for completeness.

- ♦ The **Domain** column denotes the domain of the check and has these possible values:
 - **C** for Common error
 - **M** for error in the class/object Model
 - **S** for error in the Statechart

Check	Correct	Complete	Domain	Notes
A <<CORBAInterface>> is mapped to server code, while the configuration is not a CORBA server (property Configuration::CORBA::CORBAEnable is not set to CORBAServer). Server mainline (property Configuration::ORBname::ServerMainLineTemplate) is ignored.	W		C	
A COM Interface can inherit only from a single COM Interface	E		M	
A COM Interface cannot have a 1-n relationship	E		M	

Check	Correct	Complete	Domain	Notes
A COM Server/COM Library can contain at most one COM Library package	E		C	
A COM TLB component can contain only one package, which should be a COM Library	E		C	
A Composition Association's End inverse cannot have a multiplicity >1. Multiplicity is ignored	W		M	
A Java class can inherit only from a single non-interface class	E		M	
A Java interface can inherit only from other interfaces	E		M	
A Package stereotyped as <<CORBAModule>> cannot contain functions or variables	E		M	
A circular composition relation was detected	W		M	
A singleton object cannot have a multiplicity other than one	W		M	
Activity diagram contains unsupported elements, such as events or triggered operations. The operation will not be generated!	W		C	
AddressSpaceName property is limited to 32 characters	E		M	
An auto-generated sequence for basic CORBA types is not generated	W		M	
Association End of composition kind cannot have qualifier. Qualifier is ignored.	W		M	
Attempt to create a global instance of an uninstantiable element	E		M	Instances of any kind can be created only from instantiable elements.

Check	Correct	Complete	Domain	Notes
Attempt to create an initial instance of an uninstantiable element	E		M	Instances of any kind can be created only from instantiable elements.
Attribute modifiers are not supported in COM/CORBA	W		M	
Attribute named the same as a state	E		M	
Attribute will not be accessible from the Web because it is missing both its accessor and mutator		W	C	
Attribute/Type references a template class as its type.	E		M	
Bad nesting	E		M	A CORBA check on permitted stereotypes in nested classes.
CG : : Package : : EventsBaseID property value is out of legal event ID range	E		M	
COM ATL class cannot be an active class	E		M	
COM ATL class cannot inherit from more than one COM Coclass	E		M	
COM Coclass can inherit only from COM Interfaces	E		M	
COM Interface can inherit only from COM Interface	E		M	
COM Library can contain only COM elements	E		M	
CORBAException has an operation	E		M	
CORBAException has an outgoing relation	E		M	
CORBAException involved in inheritance	E		M	
CORBAInterface inherits a non-CORBAInterface	E		M	
Cannot find template of template specialization	E		M	

Check	Correct	Complete	Domain	Notes
Class does not realize all the interfaces provided by its behavioral ports.		W	M	
Class does not use all its reactive interface's receptions and triggered operations		W	M	
Class named the same as its package	E		M	A class and package cannot have the same name, because this would interfere with proper code generation.
Class with empty statechart		W	M	
Code generation does not support instrumentation of symmetric associations to or from files. The instrumentation of association will not be generated.	W		M	
Code generation does not support the use of event/triggered operation arguments whose type contains a C++ reference ('&').	E		M	
Code generation for operations with activity diagrams is not supported for operations with variable-length argument lists. The operation will not be generated!		W	C	
Code generation ignores inheritance between classes in C.	W		M	
Code generation scope contains more than one SDLBlock class. This may result in compilation errors.		W	M	
Component contains CORBA elements, but configuration is neither a CORBA client nor a CORBA server (property Configuration::CORBA::CORBAEnable is set to No	E		C	

Check	Correct	Complete	Domain	Notes
Component file contains unsupported fragments by Classic Code Generation	W		M	
Composite with single component		W	M	
Const attribute cannot have initial value. Initial value is ignored.	W		M	
Constructors and destructors cannot be exported to the web. Web Instrumentation code will not be generated for them		W	C	
Cross package link requires component based initialization scheme (CG::Component::InitializationScheme)	W		M	
Currently only rapid ports (relay of events) are supported in C		W	M	
Dangling transition	E		S	A dangling transition is an transition that does not connect to another element. This can occur in Rational Rhapsody if the element to which an transition is connected to is deleted. Rational Rhapsody will not automatically delete the transition in this case so that you do not lose any data on the transition.
Default names		W	C	Some elements in the model use the default names assigned by Rational Rhapsody.
Initial Connector not targeted to its state's substate	E		S	Every Or state with more than one substate must have an initial connector. This error occurs when the initial connector leads to something other than one of the substates for the Or state.

Check	Correct	Complete	Domain	Notes
Dependency between components will not be generated based on the <<Usage>> dependency since a matching configuration on the dependent component is not found	W		C	
Dependency on unresolved element	E		M	Rational Rhapsody cannot find the element referenced. See Unloaded units .
Dynamic allocation should be allowed for a non-embeddable object	E		M	
ESTL does not support multiple/virtual inheritance	W		M	
Element with no relations		I	M	
Empty body of primitive operations or global functions		W	M	The implementations of these operations/functions are not defined.
Empty description		W	C	
Event ID is not unique	E		M	Event IDs should be unique to avoid conflicts.
Event and generated state in a class have conflicting names	E		M	The implementation names of events and state cannot be the same.
Event is defined in the package but is not referenced in the interface of this package's Classes		W	M	You have defined an event in a package but there is no class that actually uses this event.
Event named the same as a class	E		M	
File includes type with the same name	W		M	
File name has to be in F8.3 format	E		M	If the Filename property (under CG::Package/Class) is defined as a file name longer than eight characters and the <lang>_CG::<Environment>::IsFileNameShort property is set to Checked, the checker reports an error.

Check	Correct	Complete	Domain	Notes
Flow charts and blocks in flow charts must reach exactly one final activity	E		S	
Flowport must have a matching attribute (by name and type) of its class owner	W		M	
Flowports connected by a link must have the same type, and in Atomic flowports, their direction should be opposite of each other (one 'In' and one 'Out')	W		M	
For dual interfaces, operations and attributes must have unique IDs (IDs cannot be blank). Rhapsody has generated unique IDs for one or more operations or attributes.		W	M	
Fork to non-orthogonal states	E		S	
Friend dependency of template class is ignored	W		M	
Global functions and variables are illegal in Java	E		M	
Ill-formed link across composite boundaries, code will not be generated	W		M	
Illegal connections of a diagram/stub connector	E		S	
Illegal for COM Coclass or COM Interface to have nested classes	E		M	
Illegal for COM Coclass to have attributes	E		M	
Illegal for COM Coclass to have operations	E		M	
Illegal for COM Library to have nested packages	E		M	
Illegal initialization of internal objects (Configuration dialog, Initialization tab)	E		M	

Check	Correct	Complete	Domain	Notes
Illegal outgoing relation for COM Coclass	E		M	
Illegal relation to a template	E		M	
Implement statechart property differs for derived and base Classes	E		M	The <code>CG::Class::ImplementStatechart</code> property must be the same for base and derived classes.
ImplementActivityDiagram is not supported in Classic Code Generation	W		M	
Implementation not supported in the generated language.	E		M	
Inconsistent multiplicity in symmetric relation: instances won't be connected	W		M	
Inheritance is illegal in template instantiation	E		M	
Isolated states		W	S	A state exists in a statechart that is not connected to any other state.
Join from non-orthogonal states	E		S	There is a join connector that is coming from a non-orthogonal state. The transition segments entering a join connector must originate from states residing in different orthogonal components.
Link doesn't instantiate an association. Link is ignored.		W	M	
Link is based on unresolved relation.	E		M	
Link via ports with no matching interfaces. Link is ignored.		W	M	
Link will not be instantiated - Duplicated link between the same ends and over the same relation	W		M	
Link will not be instantiated - ill-formed link across composite boundaries	W		M	

Check	Correct	Complete	Domain	Notes
Methods of dual and custom interfaces must return HRESULT		W	M	
Mismatch between implementation and multiplicity	E		M	The Implementation property setting is not appropriate for the multiplicity of the relation.
Missing runtime libraries required for Webify Toolkit. Check the value of GetConnectedRuntimeLibraries property for your current environment.		W	M	
Missing template instantiation parameters value	W		M	
Missing template specialization parameters value	W		M	
Misuse of embedded implementation in a relation		E	M	Embedded <Fixed/Scalar> properties are not correctly set for a relation.
Modeling of composite types (Enumeration/ Typedef) is not supported in COM/CORBA	W		M	
Multiple inheritance from reactive classes is not supported	E		M	
Multiple timeouts and duplicate triggers from the same state	E		M	Each state should only have a single timeout or trigger.
Multiple transitions with the same origin and destination - only one allowed	E		S	
Name already in use by the component	E		M	
Networkport and Flowport connected by a link must have the same type, and their direction should match - Input Networkport linked to "In" Flowport and vice versa.	E		M	

Check	Correct	Complete	Domain	Notes
Node has no outgoing transitions, an implicit transition to the final activity is created		W	S	
Non-behavioral port not connected to internal part. Assuming port is meant to be behavioral.		I	M	
Non-behavioral port with explicit interfaces is not connected to an internal part. Messages might not be relayed.		W	M	
Non-interface classes are being specified as provided or required by the port. Please revise contract.		E	M	
Not enough values for initializer arguments	W		M	
Number of events in the package exceed the event ID range (defined in the property <code><lang>::Component::PackageEventIdRange</code>)	E		M	
Objects with multiplicity greater than one are not initialized correctly when <code>JAVA_CG::Component::InitializationScheme</code> is set to <code>ByComponent</code> . Either set the property to <code>ByPackage</code> or initialize objects programmatically.	W		M	
Only a COM Library can contain COM elements	E		M	
Only components stereotyped as COM DLL, COM EXE, and COM TLB can contain a COM Library.	E		C	
Only one 'SFunctionBlock' could be in a component scope	E		M	
Operation with activity diagram contains user-supplied code, which will be ignored.	W		C	

Check	Correct	Complete	Domain	Notes
Or state with no default state		E	S	You have created an Or state without determining which is the default state. Use an initial connector in the statechart to determine the default state (error of completeness).
Out of event IDs. There are more packages with events than possible event IDs. Modify the CG::Package::PackageEventIdRange property or reduce the number of packages with events	E		M	
Out of triggered operation IDs	E		M	There is a limit of 1,768 triggered operations for a class and all its base classes.
Outgoing interface must be a COM Interface	E		M	
Outgoing relation from a COM Interface must be stereotyped as connection point	E		M	
Outgoing relation from a CORBAInterface to a non-CORBAInterface	E		M	CORBAInterfaces can only have outgoing relations to other CORBAInterfaces.
Package is defined under its class in the same implementation file	E		C	
Port connected to more than one end that provides the same interface.	W		M	
Port has an empty contract - no provided or required interfaces were specified. Assuming port is meant to replay an event.		I	M	
Port has unresolved Contract	E		M	
Port provides and requires same interface(s) - please revise contract details.	E		M	

Check	Correct	Complete	Domain	Notes
Port with empty contract owned by non-reactive class/object. Port will not relay messages.		W	M	
Ports code generation is only supported in C++ and C. They will be ignored.	W		M	
Primitive, triggered operation or event is named the same as a state	E		M	You created a state with the same name as an event.
Published object's name is limited to 32 characters	E		M	
Qualifier for qualified relation not found	E		M	No qualifier was defined for a qualified relation.
Reactive interface with a reactive super class; code cannot be generated	E		M	
Reactive interface with a statechart or an activity diagram; code cannot be generated	E		M	
Reactive interface without receptions or triggered operations		I	M	
Reactive template and Reusable statechart generation scheme	E		M	The Flat implementation of statecharts must be used with reactive template classes.
Reference To Template Parameter Type From Another Class	E		M	
Reference to unresolved element in the scope of the active component	E		M	
Reference to unresolved event		E	M	A reference to an event exists in one view, but that event does not appear in at least one other view. This can occur when you are collaborating with other developers.
Reference to unresolved relational class		E	M	Same as "Reference to unresolved event," except that the unresolved element is a relational class.

Check	Correct	Complete	Domain	Notes
Reference to unresolved statechart		E	M	Same as "Reference to unresolved event," except that the unresolved element is a statechart.
Reference to unresolved stereotype	E		M	Same as "Reference to unresolved event," except that the unresolved element is a stereotype
Reference to unresolved super class		E	M	Same as "Reference to unresolved event," except that the unresolved element is a superclass.
Reference to unresolved type		E	M	At least one element is defined to be of a type that is not defined in the model.
Relation should be implemented as static array when static architecture is used	W		M	The Implementation property of the relation should be set to StaticArray when using static architecture.
Relation to a CORBAException	E		M	Relations to CORBAExceptions are not allowed.
Relation without a multiplicity		W	M	
Relations from Java interfaces cannot be generated	E		M	
Rhapsody code generation does not support link between port required and CORBA interfaces. Code will not be generated.	W		M	
Rhapsody doesn't support multiplicity of more than 1 for Flowports	E		M	
SDL model data required for code generation is missing. Check the SDL model execution information that you provided in the Import/Sync SDL model dialog.		W	M	
Sendaction, unresolved event or its arguments		W	S	

Check	Correct	Complete	Domain	Notes
Sendaction, unresolved index of a target	E		S	
Since Arguments cannot take on default values within the CORBA domain, these default values will be ignored within the CORBA domain.	W		M	
Since Attributes cannot take on initial values within the CORBA domain, these initial values will be ignored.	W		M	
Since Enumeration Constants cannot take on default values within the CORBA domain, these default values will be ignored.	W		M	
Since animation of class nested in template is not supported, instrumentation code will not be generated.	W		M	
Since attributes within the CORBA domain cannot be marked with a Static modifier, attributes marked as Static will ignore that marking during code generation.	W		M	
Since attributes within the CORBA domain cannot be of Reference Types, attributes marked with a Reference modifier will ignore that marking during code generation.	W		M	
Since structure attributes within the CORBA domain cannot be marked with a Constant modifier, attributes marked as Constant will ignore that marking during code generation.	W		M	

Check	Correct	Complete	Domain	Notes
Since the <code>CORBA::Attribute::ConstantAsReadOnly</code> property is set to <code>False</code> , attributes marked as <code>Constant</code> will ignore that marking during code generation.	W		M	
Since typedefs within the CORBA domain cannot be marked with a <code>Constant</code> modifier, typedefs marked as <code>Constant</code> will ignore that marking during code generation.	W		M	
Since typedefs within the CORBA domain cannot be of <code>Reference Types</code> , typedefs marked as a <code>Reference</code> modifier will ignore that marking during code generation.	W		M	
Singleton stereotype ignored: instance located in a different package		W	M	
Singleton stereotype ignored: instance multiplicity is not 1		W	M	
Singleton stereotype ignored: matching instance is not owned by a package		W	M	
Singleton stereotype ignored: multiple instances found		W	M	
Singleton stereotype ignored: no matching instance found		W	M	
Singleton stereotype ignored: the <code>C_G::Class::ObjectT</code> <code>typeAsSingleton</code> property is set to <code>False</code>		W	M	
SourceArtifacts under class or package are not supported in Class code generation	W		M	

Check	Correct	Complete	Domain	Notes
State named the same as its own class, super class, or related class	E		M	
Static memory Class with non flat statechart	E		M	Flat implementation of statecharts must be used with static architectures.
Static memory class with override of operator new or delete	E		M	
Static memory element cannot be initialized	E		M	
Static reaction without action		W	S	You have defined a static reaction for a state but have not defined an action for it in the Features window.
Static reaction without guard or trigger		E	S	
StaticImport of a non-static class member/method	W		M	
Stereotype exception is ignored when inheriting from a non-exception class	W		M	
Symmetric relation with bi-directional inline in specification causes a dependency loop that is not supported by <language>; the inline is ignored during code generation	W		M	
Template Specialization In Different Place Than Its Template	E		M	
Template instantiation of unresolved template	E		M	
The Active Configuration must have the 'S-FunctionConfig' stereotype when creating a non-animated S-Function or the Instrumentation Mode set to animation for animated S-Function	W		M	

Check	Correct	Complete	Domain	Notes
The SDL_Suite environment header file was not found. It is recommended that you return to SDL Suite, select the Generate environment header file check box in the Generate > Make dialog and remake.		W	M	
The Trigger of an transition can't be abstract	W		M	
The active component is defined with "Other" as its build type	E		C	
The body of an abstract method is ignored	W		M	
The contract of the port is not an interface. Please replace contract or convert it to an interface.	E		M	
The events base ID set by the CG::Package::EventsBaseID property collides with the generated base events IDs	E		M	
The name of the actor is not a legal code name	E		M	
The scope of the active component contains File-s that are not external	E		M	
Transition's edge is inconsistent with the pin's direction	W		S	
Transitions cannot cross block boundaries in flow charts	E		S	
Trigger of transition is not under its swimlane's represented class	W		S	
Type is mapped to File. Mapping is ignored.	W		M	
Typedef with Constant modifier is based on a type with a Constant modifier.	E		M	

Checks

Check	Correct	Complete	Domain	Notes
Unable to generate link - multiplicities do not match or specified using *, or a range of numbers	W		M	
Unresolved type referenced by a template parameter.		E	M	
Unspecified AssociationRole - the AssociationRole is not connected to a 'formal Link'	W		M	
Unspecified ClassifierRole - the object is not connected to a 'formal classifier'	W		M	
Unspecified message	W		M	
Unsupported elements in flowchart	E		S	
Usage dependencies on IDE configurations are currently not generated. If needed, specify the build details in the configuration's Features window	I		M	
Web support is unavailable for a language variable of this type. Web Instrumentation code will not be generated for it.		W	C	
Web support is unavailable for global functions and global variables. Web Instrumentation code will not be generated for them.		W	C	
Web support is unavailable for operations or events with more than one argument. Web Instrumentation code will not be generated for them.		W	C	
Web support is unavailable for templates and template instantiations. Web Instrumentation code will not be generated for them.		W	C	

Check	Correct	Complete	Domain	Notes
When the property <code>C.CG::Class::EnabledDynamicAllocation</code> is set to FALSE, <code>C.CG::Configuration::InitializeEmbeddableObjectsByValue</code> should be set to TRUE.	E		C	
When working with an IDF environment, the IDF profile should be used	W		M	
Wrong Nested Statechart Hierarchy, try to use 'MergeBack' or Delete AndState to fix hierarchy	E		M	
Wrong language of element in scope	E		M	
Wrong language of initial instance	E		M	

Basic code generation concepts

This section provides you with basic code generation concepts in Rational Rhapsody. While this section focuses mostly on C++, information about other languages (C, Java, and Ada) might also appear.

Rational Rhapsody generates implementation code from your UML model. You can generate code either for an entire configuration or for selected classes. Inputs to the code generator are the model and the code generation (<lang>_CG and CG) properties. Outputs from the code generator are source files in the target language: specification files, implementation files, and makefiles.

Note that you can set up roundtripping and reverse engineering in Rational Rhapsody Developer for C and C++ so that they respect the structure of the code and preserve this structure when code is roundtripped/regenerated from the Rational Rhapsody model. For details on how to activate the code respect ability, see [Reverse engineering](#).

C code generation in Rational Rhapsody is compliant with MISRA-C:1998. Note that there are justified violations, which are noted where appropriate.

Code generation overview

Between the code generator and the real-time Object Execution Framework (OXF), which is provided as a set of libraries, Rational Rhapsody can implement most low-level design decisions for you. These decisions include how to implement design elements such as associations, multiplicities of related objects, threads, and state machines.

- ♦ **Elaborative.** You can use Rational Rhapsody simply as a code browser, with all relations, state machine generation, and so on disabled. No instance initializations or links are generated. You do everything manually.
- ♦ **Translative.** You can draw a composite with components, links, and state machines, click a button, and get your application running while having to write only a minimum of code (you would have to write at least some actions in a statechart).

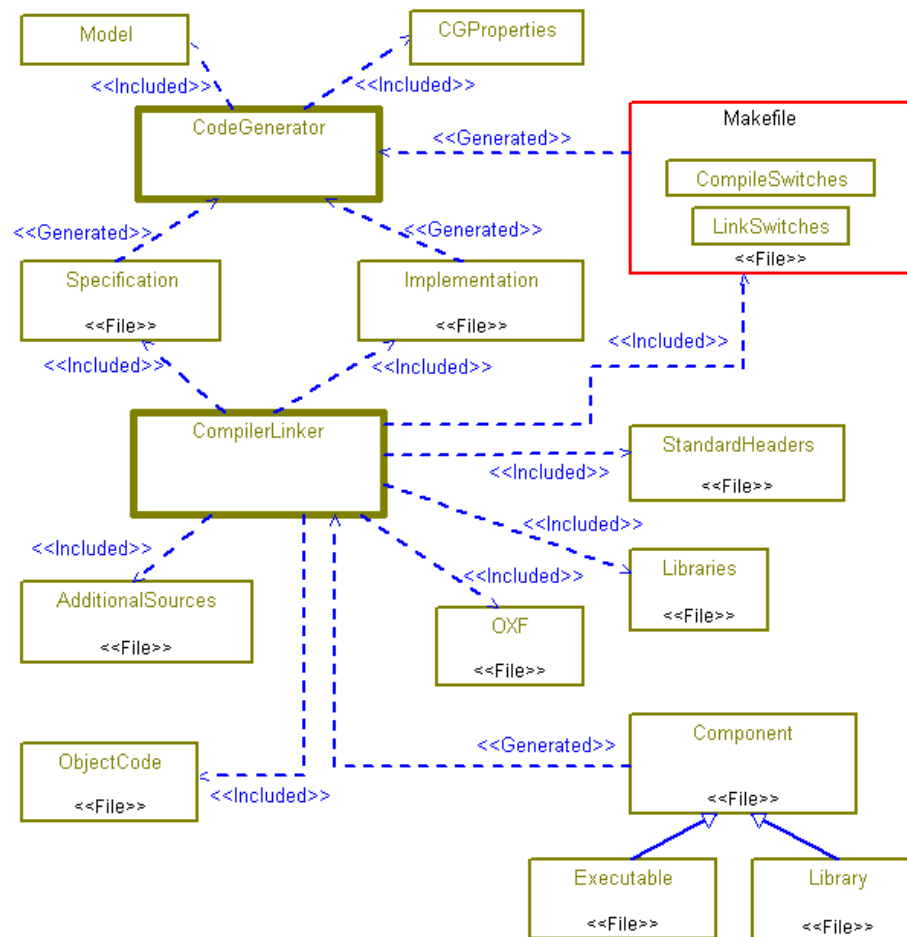
The Rational Rhapsody code generator can be both elaborative and translative to varying degrees. Rational Rhapsody does not force translation, but allows you to refine the code generation process to the wanted level. Rational Rhapsody can run in either mode, or anywhere between these two extremes.

Note

Microsoft is the default working environment for Rational Rhapsody. You can specify other “out-of-the-box” environments in the environment settings for the configuration. For more information, see [Component configurations in the browser](#).

The following figure shows the elements involved in generating code and building a component in Rational Rhapsody. The dependency arrows indicate which files are generated and which files are included by the code generator and compiler, respectively. The thick borders around the code generator and compiler indicate that these are active classes.

Object Model of Generate, Make, Run



The Code Toolbar

When you are ready to build and execute your program, you can use the Code menu or the **Code** toolbar. If the **Code** toolbar is not displayed, choose **View > Toolbars > Code**.

The code generation process has the following operations and icons:

- ♦  **Make** builds the executable. You must have already generated the code for the model.
- ♦  **GMR** (Generate, Make, Run) generates the code, builds the executable, and runs the executable.
- ♦  **Stop Make/Execution** stops the build, or stops the program execution.
- ♦  **Run Executable** launches the executable portion of the model.
- ♦  **Disable dynamic model code associativity** disables automatic changes to the code whenever you make changes to the model. By default, dynamic model-code associativity is available.

Generating Code

Before you generate code, you must set the active configuration, as described in [Setting the active configuration](#).

The code generator automatically runs the checker to check for inconsistencies that might cause problems in generating or compiling the code. Some of the checks performed by the checker detect potentially fatal conditions that might cause code generation to stop processing if not corrected before generating code. For information about selecting checks to perform, see [Checks](#).

- ♦ Select **Code > Generate > (active configuration)**, or
- ♦ Press **Ctrl+F7**

Note that it is possible to generate code without intertask communication and event dispatching from Rational Rhapsody, but this will disable the animation and visual debugging features. You can mitigate this effect by wrapping your in-house intertask communication and event dispatching routines inside an operation that is defined inside the model. In this case, the visualization is of the operation as a representative of your “real” intertask communication and event dispatching.

Incremental Code Generation

After initial code generation for a configuration, Rational Rhapsody generates code only for elements that were modified since the last generation. This provides a significant performance improvement by reducing the time required for generating code.

In support of incremental code generation, Rational Rhapsody creates a file in the configuration directory named <configuration>.cg_info. This file is internal to Rational Rhapsody and does not need to be placed under configuration management.

Forcing Complete Code Generation

In some instances, you might want to generate the entire model, rather than just the modified elements.

To force a complete regeneration of the model, select **Code > Re Generate > (configuration)**.

Note

Forcing a regeneration of code using **Code > Re Generate > (configuration)** does not always regenerate the source files. The model is being regenerated (as is shown in the Output window) but only to temporary files. These files are then compared with their original counterparts and then overwritten if changes have been made. If you want to ensure that the code is completely regenerated each time, then you need to delete the active configuration folder. To automate this with a macro, choose **Tools > Customize** to create your macro (helper application) and set the helper trigger to **Before Code Generation**.

Regenerating Configuration Files

When you delete classes or packages from the model, or add the first global instance to a package, you need to regenerate the configuration files (main and makefile).

- ◆ Select **Code > Re Generate > Configuration Files**, or
- ◆ Right-click the active configuration in the browser and select **Generate Configuration Main and Makefiles**

Smart Generation of Packages

Rational Rhapsody generates code for a package only if it contains meaningful information, such as instances, types, and functions. This streamlines the code generation process by preventing generation of unnecessary package files.

To modify this behavior, use the CG: :Package: :GeneratePackageCode property. See the definition provided for the property on the applicable **Properties** tab of the Features window.

To remove existing package files that do not contain meaningful information, select **Code > Clean Redundant Source Files**.

Generating Code Guidelines

When you generate code, consider the following information:

- ♦ A reactive class that inherits from a non-reactive class might cause a compilation warning in Instrumentation mode. You can ignore this warning.
- ♦ If a class has a dependency on another class that is outside the scope of the component, Rational Rhapsody does not automatically generate an `#include` statement for the external class. You must set the `<lang>_CG::Class::SpecInclude` property for the class of the dependent.

Dynamic Model-Code Associativity

If you choose **Code > Dynamic Model Code Associativity** and then either the **Bidirectional** or **Code Generation** command, Rational Rhapsody generates code automatically when an element changes and is in need of regeneration. These changes include the following possibilities:

- ♦ Changes to the element itself, such as its name or properties or by adding or removing parts of the element
- ♦ Changes in its owner, an associated element, a project, a selected component or configuration
- ♦ Adding an element to the model through the **Add to Model** feature
- ♦ Checking out the element with a CM tool
- ♦ Using the **Undo** command while editing the element

Code for an element is automatically generated if one of the following occurs:

- ♦ You open a Code View window for the element, either with the internal code editor or an external code editor. See [Viewing and Editing the Generated Code](#).
- ♦ You set focus on an existing Code View window for that element.

If you set dynamic model-code associativity to **None** or to **Roundtrip**, you must use **Code > Generate** to generate code for the modified elements.

Note

Dynamic model-code associativity is applicable to all versions of Rational Rhapsody (meaning, Rational Rhapsody Developer for C, C++, Java, and Ada).

Generating Makefiles

Rational Rhapsody generates plain makefiles that include the list of files to be compiled and linked, their dependencies, and the compilation and link command.

Rational Rhapsody generates makefiles when it generates the target file. The `<lang>_CG::<Environment>::InvokeMake` property defines how to execute the makefile. You can customize the content of the makefile by modifying the `<lang>_CG::<Environment>::MakeFileContent` property. See the definition provided for these properties on the applicable **Properties** tab of the Features window.

Stopping Code Generation

To stop a long code generation session, select **Code > Stop X**. For example, if you are in the middle of building the code, the name of the option is **Stop Build**.

Another way to stop code generation is to create a file named `abort_code_gen` (no extension) in the root generation directory of the configuration. Rational Rhapsody checks for the file while generating code; if the file is found, Rational Rhapsody deletes the file and stops code generation.

Note

If you start code generation from the Active Code View, you can stop only by using the `abort_code_gen` file. You cannot stop using the UI.

Targets

Once the code has been generated correctly, you can build the target component and delete old objects.

Building the Target

To build the target, use one of the following methods:

- ♦ Select **Code > Build <targetname>.exe**
- ♦ Click the **Build Target** button on the **Code** toolbar

As Rational Rhapsody compiles the code, compiler messages are displayed in the Output window. When compilation is completed, the message **Build Done** is displayed.

Note

If you want to build applications for 64-bit targets, you must first rebuild the Rational Rhapsody framework libraries. If you are running Rational Rhapsody on a 64-bit system, then if you rebuild the libraries using **Code > Build Framework**, the Rational Rhapsody libraries will be rebuilt such that you will be able to build applications for 64-bit targets. However, if you are running Rational Rhapsody on a 32-bit system, you will have to rebuild the Rational Rhapsody framework libraries manually.

Deleting Old Objects Before Building Applications

In some cases, it is possible for the linker to link an application using old objects. To clean out old objects, use one of the following methods:

- ◆ Use the **Code > Clean** command to delete all compiled object files for the current configuration.
- ◆ Use the **Code > Rebuild** command every time, which might be time-consuming for complex systems.
- ◆ Modify the start of make within etc/msmake.bat and remove the /I option. This will stop the build after the first source with errors.
- ◆ Within the **site.prp** file, change the CPPCompileCommand property from String to MultiLine, and change the content so that before a file is compiled, the existing .obj file is deleted.

For example:

CPPCompileCommand Property

Metaclass Microsoft:

```
Property CPPCompileCommand MultiLine
" if exist $OMFileObjPath erase $OMFileObjPath

$(CPP) $OMFileCPPCompileSwitches
/Fo\"$OMFileObjPath\" \"$OMFileImpPath\" "
```

Metaclass VxWorks:

```
Property CPPCompileCommand MultiLine
" @echo Compiling $OMFileImpPath

@$(RM) $OMFileObjPath

@$(CXX) $(C++FLAGS) $OMFileCPPCompileSwitches -o
$OMFileObjPath $OMFileImpPath"
```

Running the Executable

Once you have built the target file, you can run it by using one of the following methods:

- ♦ Select **Code > Run <config>.exe**.
- ♦ Click the **Run** tool in the **Code** toolbar.

The default executable application is a console application. Once the application is started, regardless of its instrumentation level, a console window is displayed. If the application sends text messages to the standard output device, they are displayed in the console window.

Shortcut for Creating an Executable

- ♦ Select **Code > Generate/Make/Run**.
- ♦ Click the **Generate/Make/Run** tool in the **Code** toolbar.

Instrumentation

If you included animation instrumentation in your code, running the code displays the **Animation** toolbar. If you included trace instrumentation in your code, running the code displays the Trace window. You can choose animation or trace instrumentation when you set the parameters of your configuration. See [Configurations](#).

For more information about instrumentation, see [Animation](#) and [Tracing](#).

Stopping Model Execution

- ♦ Select **Code > Stop**.
- ♦ Click the **Stop Make/Execution** tool in the **Code** toolbar.

Generating Code for Individual Elements

You can generate code for a package and all its classes, or generate code for selected classes within a package. You can generate code from the Code menu, the browser, or an OMD.

Before you generate code for a class or package, you must set the active configuration, just as if you were generating the executable for the entire model. For more information, see [Setting the active configuration](#).

Using the Code Menu

1. Select a class from the browser, or select one or more classes in an OMD.
2. Select **Code > Generate > Selected Classes**.

If you have generated code for the entire model at least once, the **Generate** command generates code only for elements that have been modified since the last code generation.

1. Select a class from the browser, or select one or more classes in an OMD.
2. Select **Code > Re Generate > Selected Classes**.

Using the Browser

1. In the browser, right-click a package or class.
2. Depending on your choice:
 - For a package, select **Generate Package** from the menu to generate code for all classes in the package.
 - For a class, select **Generate Class** to generate code only for that class.

Using an Object Model Diagram

To generate code for a class in an OMD, right-click the class and select **Generate Class**.

Results of Code Generation

When code generation completes, Rational Rhapsody informs you of the results. Code generation messages are displayed in the Output window. Click a message to view the relevant location in the model.

Output Messages

When you generate code and build the target, messages are displayed in the Output window that either confirm the success of the operation or inform you of errors.

Some modeling constructs can cause code generation errors. Checking the model before generating the code enables you to discover and correct most of these errors before they can cause serious problems.

Locating and Fixing Compilation Errors

Rational Rhapsody displays compilation errors that occur. To find the source of the compilation error in the code, double-click the relevant error message. Rational Rhapsody tries to get you as close as possible to the source of the compilation error:

- ◆ If the source of the problem is in the implementation of an operation, the Features window for the operation is opened with the **Implementation** tab displayed and the problematic line of code highlighted.
- ◆ If the source of the problem is in the initialization for a configuration, the Features window for the configuration is opened with the **Initialization** tab displayed and the problematic line in the initialization code highlighted.
- ◆ If the source of the problem is in the actions defined for a state, the Features window for the state is opened with the **General** tab displayed and the problematic line of code highlighted.
- ◆ If the source of the problem is in a reaction defined for a state, the Features window for the relevant reaction is opened and the problematic line of code is highlighted.
- ◆ If the source of the problem is in the action code defined for a transition, the Features window for the transition is opened with the **General** tab displayed and the problematic line of code is highlighted.
- ◆ For other compilation errors, Rational Rhapsody opens up the relevant file in the code editor and highlights the problematic line of code. If you know where to correct this code within the model, you should do so. If not, you can correct the code manually and roundtrip the corrected code back into the model.

Note

If there is no apparent problem in the model and you have been generating code on selected classes, clean the whole configuration using the **Code/Clean** menu command. Compilation errors can occur if files were generated with different versions of the model.

Viewing and Editing the Generated Code

The Code View feature enables you to edit code for classes or a selected package (but not configurations). You can select one or more classes and bring up a text editor of your choice to edit the code files directly.

Setting the Scope of the Code View Editor

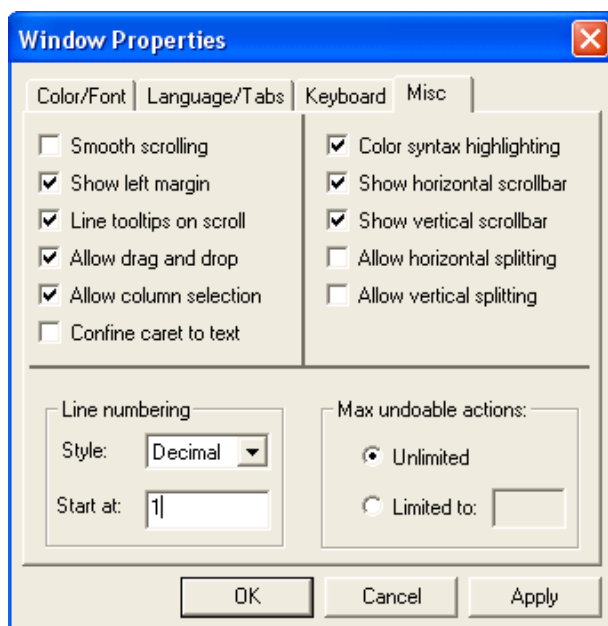
Before you can view or edit code, you must set the scope of the code view editor:

1. Right-click the component that includes the packages or classes whose code you want to view and select **Set as Active Component**.
2. In the **Current Configuration** field, select the configuration you want to use.

Adding Line Numbers

To display line numbers for the generated code:

1. Select **View > Active Code View** from the menu bar. Rational Rhapsody displays the generated code in the results window.
2. To display line numbers, click the tab for the generated code and right-click in the code window.
3. Select **Properties** from the menu and click the **Misc** tab.
4. In the Line Numbering area, select **Decimal** from the **Style** menu and enter **1** in the **Start at** field as shown in the following figure.



5. Click **OK**.

Editing Code

There are two basic methods that can be used to edit code:

- ◆ Highlight the element, then select **Code > Edit > Selected classes**.
- ◆ Right-click the element, then select **Edit <element>**.

The specification and implementation files for the selected classes or package open in separate windows. You can use the internal code editor or an external code editor to edit the files. For more information, see [Using an External Editor](#).

Depending on the dynamic model-code associativity (DMCA) setting, the changes you make to the code can be automatically roundtripped, or incorporated into the model. For more information, see [Automatic and forced roundtripping](#).

In addition, the DMCA setting determines whether open code views are updated if the code is modified by an external editor or the code generator. In the case of code generation, the code might have changed significantly. If this happens, the following message might be displayed:

```
filename: This file has been modified outside the source editor. Do you want  
to reload?
```

If you click **Yes**, the code view is refreshed with the new file contents and does not replace implementation files with obsolete information.

Locating Model Elements

When viewing code, Rational Rhapsody provides a rapid method to locate in the browser the model element represented by the current position in the code.

To locate the model element, do one of the following actions:

- ◆ Right-click in the code window and then select **Locate in Model**.
- ◆ Press **Ctrl+L**.

The element represented by the code is highlighted in the browser and the Features window is opened with the relevant line of code highlighted in the window.

If the model element is part of a statechart, the statechart is opened, with the relevant element highlighted in the diagram, and the Features window is opened with the relevant line of code highlighted.

Note

This feature is available both in the normal code view and in the active code view.

Regenerating Code in the Editor

To re-generate code for files that are currently open in the editor, choose **Code > Re Generate > Focused Views**.

Associating Files with an Editor

You can associate a specific editor with the extension of a source file:

1. Set the `General::Model::ClassCodeEditor` property to **Associate**.
2. Click **OK**.

Note

Microsoft Visual C++ 6.0 includes an **Open with MSDEV** operation in the list of operations associated with *.cpp and *.h files. If you have MVC++ 6.0, Rational Rhapsody can erroneously associate the source files with the **Open with MSDEV** operation instead of with the **Open** operation.

To make sure that Rational Rhapsody associates the MVC++ **Open** operation with the source files:

1. From Windows Explorer, select **View > Options > File Types**.
2. Select a file type (for example, **C Header File** associated with the file extension H), then select **Edit**.

The Edit File Type window opens. Generally, the first action listed is **Open with MSDEV**.

3. Click **New**, add an “Open” action, and associate the **Open** action with your preferred editor application.
4. Remove the **Open with MSDEV** action.

Using an External Editor

Note that if you use an external editor to edit code for a Rational Rhapsody model, you can set up Rational Rhapsody Developer for C++ and C so that it respects the structure of the code and preserves this structure when code is regenerated from the Rational Rhapsody model. For details on how to activate the code respect ability, see [Code respect](#).

To specify an external editor when you edit code:

1. Select **File > Project Properties**.
2. Select the **Properties** tab.
3. Navigate to the `General::Model::EditorCommandLine` property.
4. Click in the property value cell in the right column to activate the field, then click the ellipsis (...) to open the Browse for File window.
5. Browse to the location of the editor you want to use (for example, Notepad) and select the editor. Click **OK** to close the window. Rational Rhapsody fills in the path in the property value field.
6. Click **OK**.

Viewing Generated Operations

For each model, Rational Rhapsody provides setter and getter operations for attributes and relations, initializer and cleanup operations, and one called `startBehavior()`. However, Rational Rhapsody displays these automatically generated operations in the browser only if you set the `CG::CGGeneral::GeneratedCodeInBrowser` property to `Checked`. After you modify this property and regenerate the code, you will be able to view the automatically generated operations in the browser.

Deleting Redundant Code Files

A file can become redundant for the following reasons:

- ♦ An element mapped to a file is deleted or renamed.
- ♦ A change is made in the component scope.
- ♦ Changes are made in the mapping of elements to files.

Changes are made in component-level elements (components, configurations, folders, and files).

To delete redundant source files from the active configuration, select **Code > Clean Redundant Source Files**.

Generating Code for Actors

When you create a configuration, you can choose to generate code for actors (see [Features of configurations](#)). If you set the active configuration to generate code for actors, Rational Rhapsody generates, compiles, and links the actor code into the same library or executable as the rest of the system. This enables you to use actors to simulate inputs and responses to the system during system tests.

There are limits on the code generated for an actor, as described in [Limitations on Actor Characteristics](#).

Selecting Actors Within a Component

Classes have code generation properties that determine whether Rational Rhapsody should generate code for relations and dependencies between actors and classes.

To select code generation for actors within a component:

1. Right-click the component in the browser and then select **Features**.
2. Select the **Initialization** tab.
3. Under **Initial Instances**, select the actors to participate in the component.
4. Make sure **Generate Code for Actors** is selected.

Generate Code for Actors is selected by default, but is cleared when loading pre-version 3.0 models to maintain backward compatibility.

By default, relations between actors and classes are generated only if the actor is generated. You can fine-tune this within the class by using the `CG::Relation::GenerateRelationWithActors` property. See the definition provided for the property on the applicable **Properties** tab of the Features window.

Limitations on Actor Characteristics

There are limitations on how actors can be created, which affect code generation. These limitations are as follows:

- ◆ An actor cannot be specified as a composite in an OMD. Therefore, Rational Rhapsody does not generate code that initializes relationships among its components.
- ◆ The base of an actor must be another actor. The base of a class must be another class.
- ◆ An actor can embed only nested actors. A class can embed only nested classes.
- ◆ The name of an actor does not have to conform to code standards for a class name. But if an actor name is illegal (that is, it does not have a name that can be compiled) an error is generated during code generation.

Generating Code for Component Diagrams

The following code generation semantics apply to component diagrams:

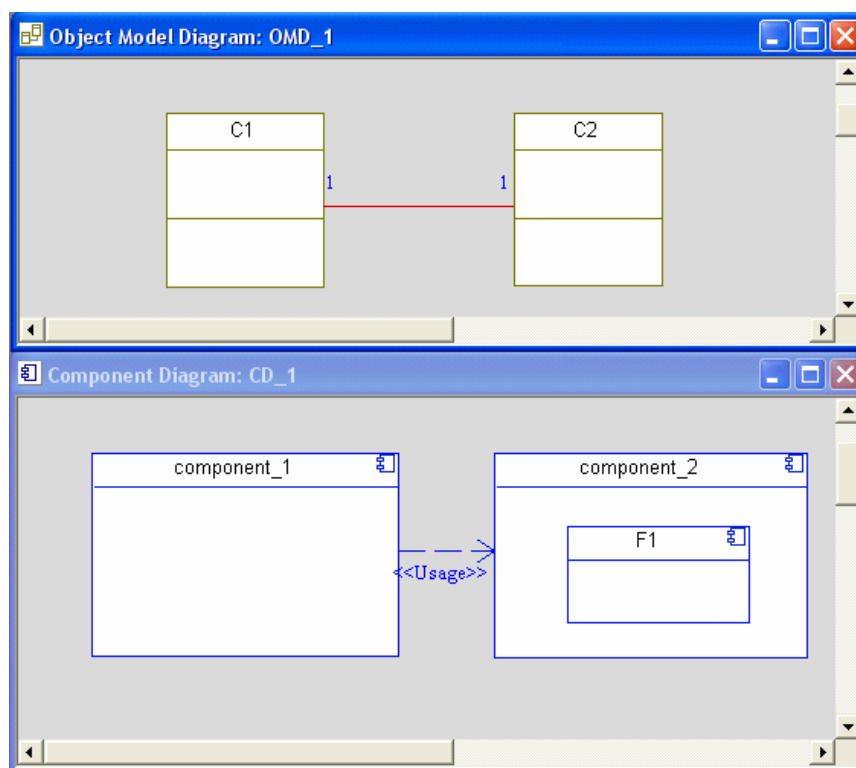
- ♦ Code generation is provided for library and executable build type components.
- ♦ Code generation is provided for folder and file component-related metatypes.
- ♦ Code is generated only with regard to relations between the binary components (library and executable).
- ♦ The following parts of a component diagram provide no code generation:
 - Relations involving files and folders
 - Interfaces realized by a component
 - All other component types that are not stereotyped «Library» or «Executable»
- ♦ A dependency between components generates code only if it has the «Usage» stereotype, with the following restrictions and limitations:
 - Related components must have a configuration with the same name.
 - If the two related components map the same element, code generation uses the first related component that was checked.
 - If an element is uniquely mapped (in the scope of a single binary component), the element file name is taken from that component.
 - If the element is not found in the scope of the generated component, related component, or uniquely mapped to some other component, the file name of the element is synthesized.

To disable the synthesis of the name, set the

`CG::Component::UseDefaultNameForUnmappedElements` property to `Cleared`.

The `CG::Component::ComponentsSearchPath` property specifies the name of related components, although the dependencies are checked before the property. For example, a dependency from component A to component B is equivalent to putting B in the `ComponentsSearchPath` property (with the obvious advantage on name change recovery).

Consider the diagrams shown in the following figure.



Classes C1 and C2 have a relation to each other (an association). The model has two components, component_1 and component_2, that each have a configuration with the same name. Component_1 has a dependency with stereotype «Usage» to component_2. Class C1 is in the scope of component_1. Class C2 is not in the scope of component_1, but is mapped to a file F1 in component_2.

- ◆ Look for an element file name in related components (when the element is not in the scope of the current component).

For example, when generating component_1, when Rational Rhapsody needs to include C2, it will include F1 (the file in component_2).

- ◆ Add related components to the makefile include path. For example, in the component_1 makefile, a new line is added to the include path with the location of component_2.
- ◆ If the current component build type is executable, and a related component build type is library, add the library to the build of the current component. For example, if the build type of component_1 is executable, and the build type of component_2 is library, the component_1 makefile will include the library of component_2 in its build.

Cross-Package Initialization

Rational Rhapsody has properties that enable you to specify code that initializes package relations after package instances are initialized, but before these instances react to events. More generally, these properties enable you to specify any other initialization code for each package in the model. They enable you to initialize cross-package relations from any of the packages that participate in the relation or from a package that does not participate in the relation.

The properties that govern package initialization code are as follows:

- ◆ `CG::Package::AdditionalInitialization` specifies additional initialization code to run after the execution of the `package initRelations()` method.
- ◆ `CG::Component::InitializationScheme` specifies at which level initialization occurs. The possible values are as follows:
 - `ByPackage` where each package is responsible for its own initialization; the component needs only to declare an attribute for each package class. This is the default option and maintains backward compatibility with pre-V3.0 Rational Rhapsody models.
 - `ByComponent` where the component must initialize all global relations declared in all of its packages. This must be done by explicit calls in the component class constructor for each package `initRelations()`, additional initialization, and `startBehavior()`.

The following example shows C++ code generated from the model in the diagram above when the `InitializationScheme` property is set to `ByPackage`.

The component code is as follows:

```
class DefaultComponent {
private :
    P1_OMInitializer initializer_P1;
    P2_OMInitializer initializer_P2;
};
```

The P1 package code is as follows:

```
P1_OMInitializer::P1_OMInitializer() {
    P1_initRelations();
    < P1 AdditionalInitializationCode value>
    P1_startBehavior();
}
```

The following example shows C++ component code generated when the `InitializationScheme` property is set to `ByComponent`:

```
DefaultComponent::DefaultComponent() {  
    P1_initRelations();  
    P2_initRelations();  
        < P1 AdditionalInitializationCode value>  
        < P2 AdditionalInitializationCode value>  
    P1_startBehavior();  
    P2_startBehavior();  
}
```

Class Code Structure

This section describes the structure of the generated code, including information on how the model maps to code and how you can control model mapping. In addition to the model elements, the generated source code includes annotations and, if instrumented, instrumentation macros.

Note

Annotations map code constructs to design constructs. They play an important role in tracing between the two. Do not touch or remove annotations. If you do, you hinder tracing between the model and the code. Annotations are comment lines starting with two slashes and a pound sign (`//#` for C and C++) or two dashes and a plus sign (`--+` for Ada).

Instrumentation macros become hooks and utility functions to serve the animation/trace framework. They are implemented as macros to minimize the impact on the source code. If you remove instrumentation macros, animation cannot work correctly.

The code structures shown are generic, using generic names such as *class* and *state*. Your code will contain the actual names.

Class Header File

The class header file contains the following sections:

- ◆ Prolog
- ◆ Forward declarations
- ◆ Instrumentation
- ◆ Virtual and private inheritance
- ◆ User-defined attributes and operations
- ◆ Variable-length argument lists
- ◆ Relation information
- ◆ Statechart implementation
- ◆ Events interface
- ◆ Serialization instrumentation
- ◆ State classes

Prolog

The prolog section includes the framework file, and a header or footer file (if applicable).

You can add additional include files using the `<lang>_CG::Class/Package::SpecIncludes` property. You might have to do this if you create dependencies to modules that are not part of the Rational Rhapsody design.

For example:

```
#ifndef class_H
#define class_H
#include <oxf.h> // The framework header file
//-----
// class.h
//-----
```

Relationships to Other Classes

This section of the header file includes forward declarations for all the related classes. These are based on the relationships of the class with other classes specified in the model.

For example:

```
class forwarded-class1;
class forwarded-class2;

class class : public OMReactive {
// Class definition, and inheritance
// from framework class (if needed!).
```

Instrumentation

If you compiled with instrumentation, instrumentation macros provide information to animation about the run-time state of the system.

For example:

```
DECLARE_META      // instrumentation for the class
```

User-Defined Attributes and Operations

In this section, all operations and attribute data members are defined. The code in this section is a direct translation of class operations and attributes. You control it by adding or removing operations and attributes from the model.

For example:

```
///// User explicit entries      /////
public :
    /// operation op1() // Annotation for the operation
    void op1();         // Definition of an operation
protected :
    // Attributes:
    /// attribute attr1 // Annotation for an attribute
    attrType1 attr1;    // Definition of an attribute
///// User implicit entries      /////
public :
    // Constructors and destructors:
    class(OMThread* thread = theMainThread);
    virtual ~class();    // Generated destructor
```

Generating Code for Static Attributes

When you generate code for a static attribute, the initial value entered is generated into a statement that initializes the static attribute outside the class constructor. Consider the following initial values for three static attributes:

Attribute	Type	Value
attr1	int	5
attr2	OMBoolean	true
attr3	OMString	"Shalom"

When you generate code, these values cause the following statements to be generated in the specification file A.h:

```
//-----
// A.h
//-----
class A {
    ///// User explicit entries      /////
protected:
    /// attribute attr3
    static OMString attr3;

    /// attribute attr1
    static int attr1;

    /// attribute attr2
    static OMBoolean attr2;
};
```

In the implementation file, A.cpp, the following initialization code is generated:

```
#include "A.h"

//-----
// A.cpp
//-----

// Static class member attribute
OMString A::attr3 = "Shalom";

// Static class member attribute
int A::attr1 = 5;

// Static class member attribute
OMBoolean A::attr2 = true;
A::A() {
};
```

Variable-Length Argument Lists

To add a variable-length argument list (. . .) as the last argument of an operation, open the Properties window for the operation, and set the CG::Operation::VariableLengthArgumentList property to Checked.

For example, if the VariableLengthArgumentList property is set to Checked for an operation void testA(int i), the following declaration generated for testA():

```
void testA(int i, ...);
```

Synthesized Methods and Data Members for Relations

This section of the header file includes every relation of the class defined in the model. If appropriate, it includes a data member, as well as a set of accessor and mutator functions to manipulate the relation.

You can control this section by changing the relation properties and container properties. For more information about the CG properties, see [Properties](#).

For example:

```
OMIterator<rclass1*> getrelation1()const;
void addrelation1(rclass* p);
void removerelation1(rclass* p);
void clearrelation1();
protected :
//
OMCollection<rclass*> relation1;
attrType1 getattr1()const;
void setattrType1(attrType1 p);
```

Statechart Implementation

- ♦ Change the statechart implementation strategy from reusable to flat.
- ♦ Disable code generation for statecharts.

For example:

```
////    Framework entries    ////
public :
    State* state1;           // State variables
    State* state2;
    void rootStateEntDef(); // The initial transition
    // method
    int state1Takeevent1(); // event takers
    int state2Takeevent2();

private :
    void initRelations(); //InitRelations or
    //InitStatechart is
    //generated to initialize
    //framework members
    void cleanUpRelations();//CleanupRelations or
    //CleanupStatechart is
    //generated to clean up
    //framework members in
    //destruction.
```

Note

The `initRelations()` and `cleanUpRelations()` operations are generated only if the `CG::Class::InitCleanUpRelations` property is set to Checked. See the definition provided for the property on the applicable **Properties** tab of the Features window.

Events Interface

The section of the code illustrates events consumed by the class. It is for documentation purposes only because all events are handled through a common interface found in `OMReactive`. This section is useful if you want to implement the event processing yourself.

For example:

```
////    Events consumed    ////
public :
    // Events:
    // event1
    // event2
```

Note that code generation supports array types in events. Is it possible to create the following `GEN()`:

```
Client->GEN(E("Hello world!"));
```

In this call, `E` is defined as follows:

```
class E : public OMEvent {
    Char message[13];
};
```

Serialization Instrumentation

If you included instrumentation, the following instrumentation macro is included in the header file:


```
DECLARE_META_EVENT  
};
```

It expands to method definitions that implement serialization services for animation.

State Classes

The state defines state classes to construct the statechart of the class. State classes are generated in the reusable state implementation strategy.

For example:

```
class state1 : public ComponentState {  
public :  
    // State class implementation  
};  
class state2 : public Orstate {  
public :  
    // State class implementation  
};  
#endif
```

You can eliminate the state classes by choosing the flat implementation strategy, where states are manifested as enumeration types.

Implementation Files

The class implementation file contains implementation of methods defined in the specification file. In addition, it contains instrumentation macros and annotations. As noted previously, eliminating or modifying either instrumentation macros and annotations can hurt the tracing and functionality between your model and code.

Headers and Footers

You can define your own headers and footers for generated files. See the property definitions displayed in the **Properties** tab of the Features window.

Prolog

The prolog section of the implementation file includes header files for all the related classes. You can add additional `#include` files using the `C_` and `CPP::Class/Package::ImpIncludes` property. See the definition provided for the property on the applicable **Properties** tab of the Features window.

You can wrap sections of code with `#ifdef` `#endif` directives, add compiler-specific keywords, and add `#pragma` directives using prolog and epilog properties. For more information, see [Wrapping Code with #ifdef-#endif](#).

The following code shows a sample prolog section:

```
/// package package
/// class class
#include "class.h"
#include <package.h>
#include <rclass1.h>
#include <rclass2.h>
//-----
// class.cpp
//-----
DECLARE_META_PACKAGE(Default)
#define op1_SERIALIZE
```

Constructors and Destructors

A default constructor and destructor are generated for each class. You can explicitly specify additional constructors, or override the default constructor or destructor by explicitly adding them to the model.

For example:

```
class::class(OMThread* thread) {
    NOTIFY_CONSTRUCTOR(class, class(), 0,
        class_SERIALIZE);
    setThread(thread);
    initRelations();
};
```

If you are defining states, use `initStatechart` instead of `initRelations`:

```
class::~~class() {
    NOTIFY_DESTRUCTOR(~class);
    cleanUpRelations();
};
```

Similarly, if you are defining states, use `cleanUpStatechart` instead of `cleanUpRelations`.

Operations

The following code pattern is created for every primitive (user-defined) operation:

```
void class::op1() {
    NOTIFY_OPERATION(op1, op1(), 0, op1_SERIALIZE);
    // Instrumentation
    //#[ operation op1() // Annotation
    // body of the operation as you entered
    //#[
};
```

Accessors and Mutators for Attributes and Relations

Accessors and mutators are automatically generated for each attribute and relation of the class. Their contents and generation can be controlled by setting relation and attribute properties.

For example:

```
attr1Type class::getattr1()const {
    return attr1;
};

void class::setattr1(attr1type p) {
    attr1 = p;
};

OMIteratorrrclass* class::getItsrrclass()const {
    OMIteratorrrclass* iter(itsrrclass);
    return iter;
};

void Referee::_addItsRclass(Class* p_Class) {
    NOTIFY_RELATION_ITEM_ADDED("itsRClass", p_Class,
```

```
        FALSE, FALSE);
    itsRclass->add(p_Class);
};

void class::removeItsRclass(Rclass* p) {
    NOTIFY_RELATION_ITEM_REMOVED();
    rclass.remove(p);
};

void class::clearItsPing() {
};
```

Instrumentation

This section includes instrumentation macros and serialization routines used by animation.

For example:

```
void class::serializeAttributes() const {
    // Serializing the attributes
};

void class::serializeRelations() const {
    // Serializing the relation
};

IMPLEMENT_META(class, FALSE)
IMPLEMENT_GET_CONCEPT(state)
```

State Event Takers

These methods implement state, event, and transition behavior. They are synthesized based on the statechart. If you set the CG::Attribute/Event/File/Generalization/Operation/Relation::Generate property of the class to Cleared, they are not generated.

For example:

```
int class::state1Takeevent1() {
    int res = eventNotConsumed;
    SETPARAMS(hungry);
    NOTIFY_TRANSITION_STARTED("2");
    Transition code
    NOTIFY_TRANSITION_TERMINATED("2");
    res = eventConsumed;
    return res;
};
```

Initialization and Cleanup

These methods implement framework-related initialization and cleanups.

For example:

```
void class::initRelations() {
    state1 = new state1Class(); // creating the states
};

void class::cleanUpRelations() {
};
```

Implementation of State Classes

This section implements a dispatch method and a constructor for each state object. You can change this by choosing the flat implementation strategy, which does not generate state classes.

For example:

```
class_state1::class_state1(class* c, State* p,
    State* cmp): LeafState(p, cmp) {
    // State constructor
};

int class_state1::takeEvent(short id) {
    int res = eventNotConsumed;
    switch(id) {
        case event1_id: {
            res = concept->state1Takeevent1();
            // Dispatching the transition method
            break;
        };
    };
};
```

Changing the Order of Operations/Functions in Generated Code

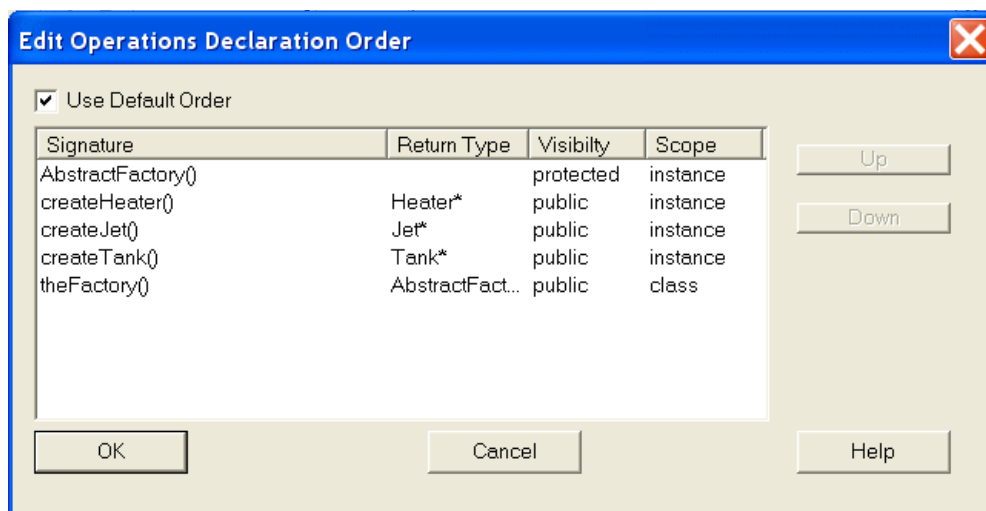
By default, operations appear in the following order in generated code:

1. Constructors and destructors
2. User-defined operations
3. Triggered operations

Within each of these categories, the operations are displayed in the following order: public, protected, private. Within these access subcategories, operations are listed alphabetically.

In some cases, you might want to define a specific order for the operations when the code is generated. To modify the order of appearance in the generated code:

1. Highlight **Operations** or **Functions** in the browser.
2. Right-click and select **Edit Operation Order** (or **Edit Function Order**).
3. In the window that is displayed, highlight the **Signature** that you want to move and use the **Up** and **Down** buttons to modify the order that will be used in code generation.



4. Click **OK**.

Note

If the **Up** and **Down** buttons are disabled (as shown in this example), clear the **Use Default Order** check box at the top of the window.

If you would like to restore the default order used by Rational Rhapsody for code generation, make these changes:

1. Select the **Use Default Order** check box.
2. Click **OK**.

Using Code-Based Documentation Systems

Rational Rhapsody enables you to standardize the way element comments are generated. Using a template, the code generator can take the element description from the model, along with its tag values, and format it as the generated comment inside the code. These generated comments can then be processed by code-based documentation tools such as Doxygen[™].

Template Properties

The following table lists the properties (under `<lang>_CG`) that enable you to standardize the way comments are generated in the code.

Metaclass	Property	Description
File	ImplementationFooter	Specifies the multiline footer to be generated at the end of implementation files.
	ImplementationHeader	Specifies the multiline header that is generated at the beginning of implementation files.
	SpecificationFooter	Specifies the multiline footer to be generated at the end of specification files.
	SpecificationHeader	Specifies the multiline header to be generated at the beginning of specification files.
Configuration	DescriptionBeginLine	Enables you to specify the prefix for the beginning of comment lines in the generated code. This functionality enables you to use a documentation system (such as Doxygen), which looks for a certain prefix to produce the documentation. Note: This property affects only the code generated for descriptions of model elements; other auto-generated comments are not affected.
	DescriptionEndLine	Enables you to specify the prefix for the end of comment lines in the generated code.

Metaclass	Property	Description
Argument Attribute Class Event Operation Package Relation Type	DescriptionTemplate	Specifies how to generate the element description in the code. An empty <code>MultiLine</code> (the default value) tells Rational Rhapsody to use the default description generation rules.

Sample Usage

This section documents a simple model configured to generate Doxygen-compatible descriptions, including the appropriate property settings.

The Model Profile

The `DoxygenDescription` profile defines the description tags and sets the default property values. The following table lists the property values for the `DoxygenDescription` profile.

Property	Value
C_ and CPP_CG::Configuration	
	"*"
DescriptionEndLine	" "
C_ and CPP_CG::File	
SpecificationHeader	<pre>"/** * (C) copyright 2004 * * @file \$FullCodeGeneratedFileName * @author \$FileAuthor * @date \$CodeGeneratedDate * @brief \$FileBrief * * Rhapsody: \$RhapsodyVersion * Component: \$ComponentName * Configuration: \$ConfigurationName * Element: \$ModelElementName * */"</pre>
C_ and CPP_CG::Class	
DescriptionTemplate	<pre>"/** * Class \$(Name) * @brief \$Description * @see \$See * @warning \$Warning */"</pre>

Property	Value
C_ and CPP_CG::Operation	
DescriptionTemplate	"/** * Operation \$Signature * @brief \$Description * \$Arguments * @return \$Return * @see \$See * @warning \$Warning */"
C_ and CPP_CG::Argument	
DescriptionTemplate	"@param \$(Name): [\$Direction] \$Description"

Point Class Definition

The Point class might contain these elements:

- ◆ Point is a 2D point representation
- ◆ Point.create() is a static method to create a new Point
- ◆ Point.create().x is the x coordinate for the new Point
- ◆ Point.create().y is the y coordinate for the new Point

The following table shows the tag values set by the model elements.

Element	Tag Name	Tag Value
Point	FileAuthor	Ford Perfect
Point	FileBrief	The Point class definition
Point	See	Point3D
Point	Warning	NA
Point.create()	Return	a new Point with the specified coordinates
Point.create()	See	NA
Point.create()	Warning	NA

The Generated Code for the Point

The file header for the Point specification file is as follows:

```
/**
 * (C) copyright 2004
 *
 * @file    DefaultComponent\DefaultConfig\Point.h
 * @author   Ford Perfect
 * @date    //! Tue, 16, Mar 2004
 * @brief    The Point class definition
 *
 * Rhapsody:      5.0.1
 * Component:     DefaultComponent
 * Configuration:  DefaultConfig
 * Element:       Point
 */
```

The generated description of the Point class is as follows:

```
/**
 * Class Point
 * @brief A 2D point representation
 * @see Point3D
 * @warning NA
 */
//## class Point
class Point {...
```

The generated description of the create() operation is as follows:

```
/**
 * Operation create(int,int)
 * @brief A static method to create a new Point
 * $Arguments
 * @param x: [in] The new Point x coordinate
 * @param y: [in] The new Point y coordinate
 * @return a new Point with the specified coordinates
 * @see NA
 * @warning NA
 */
//## operation create(int,int)
static PPoint* create(int x, int y);
```

Roundtripping Behavior

Advanced/Full roundtrip does not update element descriptions when the relevant `DescriptionTemplate` properties are not empty.

The following table lists the properties for which roundtripping does not update the element description.

Element	Element Descriptions are Not Updated when these Properties are Not Empty
Argument	The argument description is not updated when the owner (operation or event) description is not updated.
Association end	<code><lang>_CG::Relation::DescriptionTemplate</code>
Attribute	<code><lang>_CG::Attribute::DescriptionTemplate</code>
Class	<code><lang>_CG::Class::DescriptionTemplate</code>
Event	<code><lang>_CG::Argument::DescriptionTemplate</code> <code><lang>_CG::Event::DescriptionTemplate</code> (both at the operation or argument level)
Operation	<code><lang>_CG::Argument::DescriptionTemplate</code> <code><lang>_CG::Operation::DescriptionTemplate</code> (both at the operation or argument level)
Package	<code><lang>_CG::Package::DescriptionTemplate</code>
Type	<code><lang>_CG::Type::DescriptionTemplate</code>

Wrapping Code with #ifdef-#endif

If you need to wrap an operation with an `#ifdef` `#endif` pair, add a compiler-specific keyword, or add a `#pragma` directive, you can set the `SpecificationProlog`, `SpecificationEpilog`, `ImplementationProlog`, and `ImplementationEpilog` properties for the operation.

For example, to specify that an operation is available only when the code is compiled with `_DEBUG`:

- ◆ Set `SpecificationProlog` to `#ifdef _DEBUG <CR>`.
- ◆ Set `SpecificationEpilog` to `#endif`.
- ◆ Set `ImplementationProlog` to `#ifdef _DEBUG <CR>`.
- ◆ Set `ImplementationEpilog` to `#endif`.

The same properties are available for configurations, packages, classes, and attributes. For detailed definitions of these properties, see the property definitions displayed in the Features window.

Overloading Operators

You can overload operators for classes created in Rational Rhapsody. For example, for a `Stack` class, you can overload the “+” operator to automatically perform a `push()` operation, and the “-” operator to automatically perform a `pop()` operation.

All of the overloaded operators (such as `operator+` and `operator-`) can be modeled as member functions, except for the stream output operator `<<`, which is a global function rather than a member function and must be declared a friend function. The overloaded operators that are class members are all defined as primitive operations.

To illustrate operator overloading, consider two classes, `Complex` and `MainClass`, defined as follows:

Class Complex

Attributes:

```
double imag;  
double real;
```

Operations:

```
Complex()                // Simple constructor  
Body:  
{  
    real = imag = 0.0;  
}  
  
Complex(const Complex& c) //Copy constructor  
Arguments: const Complex& c  
Body:  
{
```

```
        real = c.real;
        imag = c.imag;
    }
Complex(double r, double i) // Convert constructor
Arguments: double r
           double i = 0.0
Body:
{
    real = r;
    imag = i;
}

operator-(Complex c) // Subtraction
Return type: Complex
Arguments: Complex c
Body:
{
    return Complex(real - c.real, imag - c.imag);
}
operator[](int index) // Array subscript
Return type: Complex&
Arguments: int index // dummy operator - only
           // for instrumentation
           // check
Body:
{
    return *this;
}
operator+(Complex& c) // Addition by value
Return type: Complex
Arguments: Complex& c
Body:
{
    return Complex(real + c.real, imag + c.imag);
}
operator+(Complex* c) // Addition by reference
Return type: Complex*
Arguments: Complex *c
Body:
{
    cGlobal = new Complex (real + c->real,
                          imag + c->imag);
    return cGlobal;
}
operator++() // Prefix increment
Return type: Complex&
Body:
{
    real += 1.0;
    imag += 1.0;
    return *this;
}
operator=(Complex& c) // Assignment by value
Return type: Complex&
Arguments: Complex& c
Body:
{
    real = c.real;
    imag = c.imag;
    return *this;
}
```

```
operator=(Complex* c)          // Assignment by reference
Return type: Complex*
Arguments: Complex *c
Body:
{
    real = c->real;
    imag = c->imag;
    return this;
}
```

The following are some examples of code generated for these overloaded operators.

This is the code generated for the overloaded prefix increment operator:

```
Complex& Complex::operator++() {
    NOTIFY_OPERATION(operator++, operator++(), 0,
        operator_SERIALIZE);
    //#[ operation operator++()
    real += 1.0;
    imag += 1.0;
    return *this;
    //#[
};
```

This is the code generated for the overloaded + operator:

```
Complex Complex::operator+(Complex& c) {
    NOTIFY_OPERATION(operator+, operator+(Complex&), 1,
        OM_operator_1_SERIALIZE);
    //#[ operation operator+(Complex&)
    return Complex(real + c.real, imag + c.imag);
    //#[
};
```

This is the code generated for the first overloaded assignment operator:

```
Complex& Complex::operator=(Complex& c) {
    NOTIFY_OPERATION(operator=, operator=(Complex&), 1,
        OM_operator_2_SERIALIZE);
    //#[ operation operator=(Complex&)
    real = c.real;
    imag = c.imag;
    return *this;
    //#[
};
```

The browser lists the MainClass, which is a composite that instantiates three Complex classes.

Its attributes are as follows:

```
Complex* c1
Complex* c2
Complex* c3

Body~MainClass()          //Destructor
Body
{
    delete c1;
    delete c2;
    delete c3;
```

```
}  
e()           // Event
```

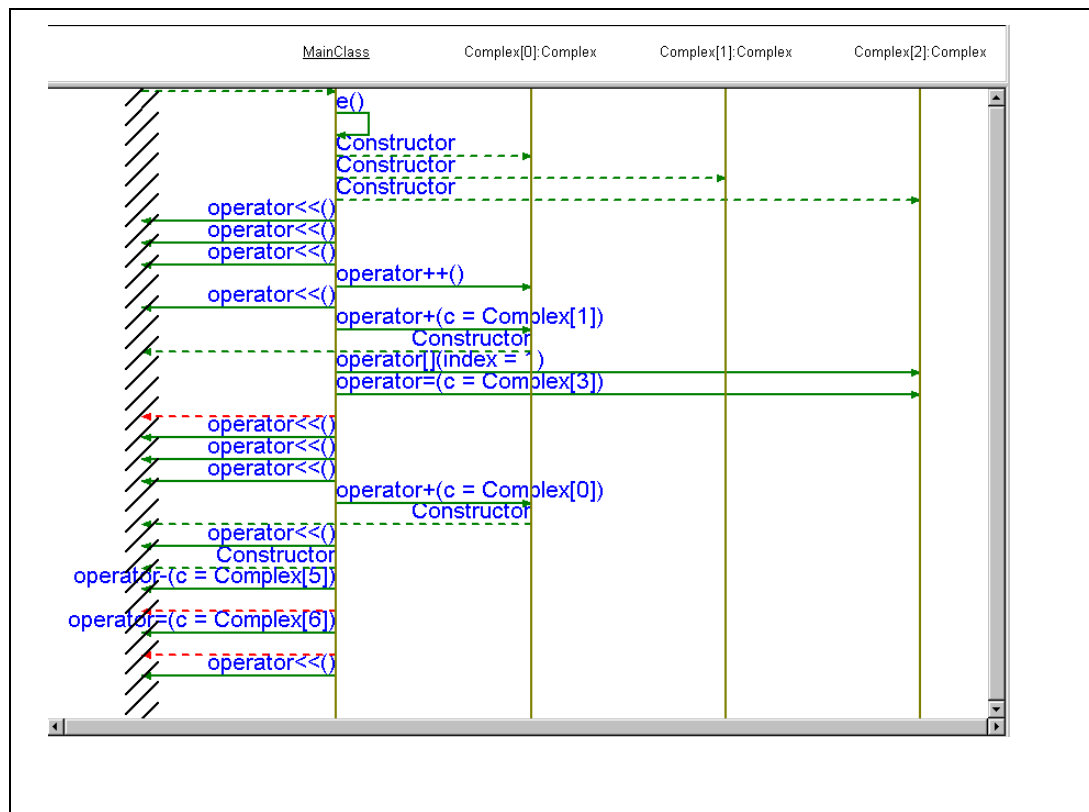
The stream output operator << is a global function that must be declared a friend to classes that want to use it. It is defined as follows:

```
operator<<  
Return type: ostream&  
Arguments:  ostream& s  
            Complex& c  
Body:  
{  
    s << "real part = " << c.real<<  
        "imagine part = " << c.imag << "\n" << flush;  
    return s;  
}
```


To watch the various constructors and overloaded operators as they are being called:

1. Assign animation instrumentation to the project by selecting **Code > Set Configuration > Edit > Setting tab**.
2. Make and run DefaultConfig.exe.
3. Using the animator, create an animated sequence diagram (ASD) that includes the MainClass and instances Complex[0]:Complex, Complex[1]:Complex, and Complex[2]:Complex.
4. Click **Go** on the Animation bar to watch the constructor and overloaded operator messages being passed between the MainClass and its part instances.

The ASD will be similar to the following figure.



Using Anonymous Instances

In Rational Rhapsody, you can create anonymous instances that you manage yourself, or create instances as components of composite instances that are managed by the composite framework.

Creating Anonymous Instances

You can create anonymous instances, apply the C++ new operator as in any conventional C++ program. Rational Rhapsody generates a default constructor (ctor) for every class in the system. You can add additional constructors using the browser.

For example, to create a new instance of class A using the default constructor, enter the following code:

```
A *a = new A();
```

For reactive and composite classes, the default constructor takes a thread parameter that, by default, is set to the main thread for the system. To associate the instance with a specific thread rather than the main thread for the system, you must explicitly pass this parameter to the constructor.

The following example creates an instance of class A and assigns it to a thread T.

```
A *a = new A(T);
```

After creating a new instance, you would probably call its relation mutators to connect it to its peers (see [Using Relations](#)). If the class is reactive, you would probably call its `startBehavior()` method next.

Composite instances manage the creation of components by providing dedicated operations for the creation of new components. For each component, there is an operation `phil` of type `Philosopher`. The new `phil` operation creates a new instance, adds it to the component relation, and passes the thread to the new component.

The following code shows how the composite sys can create a new component `phil`.

```
Philosopher *pPhil = sys->newPhil();
```

After creating the new instance, you would probably call its relation mutators to connect it to its peers. If the class is reactive, you would probably call its `startBehavior()` method next.

Deleting Anonymous Instances

In Rational Rhapsody, you manage anonymous instances yourself. As a result, it is your responsibility to delete them. Components of composite instances are managed by the composite. To delete them, you must make an explicit request of the composite object.

Apply the C++ `delete` operator as in any conventional C++ program. Deletion of instances involves applying the destructor of that instance. Rational Rhapsody generates a default destructor for every class in the system. You can add code to the destructor through the browser.

For example, to delete an instance pointed to by `aB`, use the following call:

```
delete aB;
```

Deleting Components of a Composite

For each component of a composite, there is a dedicated operation that deletes an instance of that component.

For example, deleting an instance pointed to `aB` from a component named `compB` in a composite `C`, use the following call:

```
C->deleteCompB(aB);
```

Using Relations

Relations are the basic means through which you reference other objects. Relations are implemented in the code using container classes. The Rational Rhapsody framework contains a set of container classes used by default. You can change the default, for example to use Standard Template Library (STL)TM containers, using properties. For more information about using the Rational Rhapsody properties, see [Properties](#).

The collections and relation accessor and mutator functions documented in this section refer to the default set provided with Rational Rhapsody.

Note

To quickly see the relations for a class, object, and package, right-click the element in the Rational Rhapsody browser and select **Show Relations in New Diagram** from the menu. For more information about this menu command, see [Showing all relations for a class, object, or package in a diagram](#).

To-One Relations

To-one relations are implemented as simple pointers. Their treatment is very similar to that of attributes; that is, they also have accessor and mutator functions.

If B is a class related to A by the role name `role`, A contains the following data member:

```
B* role;
```

It contains the following methods:

```
B* getRole();  
void setRole(B* p_B);
```

These defaults are modifiable through the properties of the role. For more information about using the Rational Rhapsody properties, see [Properties](#).

To-Many Relations

To-many relations are implemented by collections of pointers using the `OMCollection` template.

If E is a class name multiply related to F by role name `role`, E contains the following data member:

```
OMCollection<F*> role;
```

The following methods are generated in E to manipulate this relation:

- ♦ To iterate through the relation, use the following accessor:
`OMIterator<F*> getRole() const;`

For example, if you want to send event *x* to each of the related *F* objects, use the following code:

```
OMIterator<F*> iter(anE->getRole());
while(*iter)
{
    *iter->GEN();
    iter++;
}
```

In this code, *anE* is an instance of *E*.

- ◆ To add an instance to the collection, use the following call:
`void addRole(F* p_F);`
- ◆ To remove an instance from the collection, use the following call:
`void removeRole(F* p_F);`
- ◆ To clear the collection, use the following call:
`void clearRole();`

These defaults are modifiable through the properties of the role. For more information about using the Rational Rhapsody properties, see [Properties](#).

Ordered To-Many Relations

Ordered to-many relations are implemented by collections of pointers using the `OMList` template.

A to-many relation is made ordered by making the `Ordered` property for the relation to `Checked`.

All the manipulation methods are the same as described in [To-Many Relations](#).

Qualified To-Many Relations

A qualified to-many relation is a to-many relation qualified by an attribute in the related class. The qualified to-many is implemented by a collection of pointers using the `OMMap` template.

All the manipulation methods are the same as described in [To-Many Relations](#). In addition, the following key qualified methods are provided.

```
F* getRole(type key) const;
void addRole(type key, F* p_F);
void removeRole(type key);
```

Random Access To-Many Relations

A random access to-many relation is a to-many relation that has been enhanced to provide random access to the items in the collection.

A to-many relation is made random access by making the `GetAtGenerate` property for the relation to `Checked`. This causes a new accessor to be generated:

```
F* getRole(int i) const;
```

Support for Static Architectures

Static architectures are often used in hard real-time and safety-critical applications with memory constraints. Rational Rhapsody provides support for applications without memory management and those in which non-determinism and memory fragmentation would create problems by completely avoiding the use of the general memory management (or heap) facility during execution (after initialization). This is a typical requirement of safety-critical systems.

Rational Rhapsody can avoid the use of the general heap facility by creating special allocators, or local heaps, for designated classes. The local heap is a preallocated, continuous, bounded chunk of memory that has the capacity to hold a user-defined number of objects. Allocation of local heaps is done via a safe and simple algorithm. Use of local heaps is particularly important for events and triggered operations.

Rational Rhapsody applications implicitly and explicitly result in dynamic memory operations in the following cases:

- ♦ **Event generation (implicit)** means optionally resolved via local heap
- ♦ **Addition of relations** means resolved by implementing with static arrays (dynamic containers remain dynamic)
- ♦ **Explicit creation of application objects via the new operator** means resolved via local heap if the application indeed dynamically creates objects

You can specify whether local heaps apply to all or only some classes, triggered operations, events, and thread event queues.

Properties for Static Memory Allocation

The following table lists some of the properties that enable you to configure static allocations for the generated code. Setting any of these properties at a level higher than an individual instance sets the default for all instances. For example, setting a class property at the component level sets the default for all class instances. In all cases, behavior is undefined if the actual number of instances exceeds the maximum declared number.

Property	Subject and Metaclass	Description
BaseNumberOfInstances	<code><lang>_CG::Class</code> <code>CG::Event</code>	Specifies the size of the local heap memory pool allocated for either: • Instances of the class (<code><lang>_CG::Class</code>) • Instances of the event (<code><lang>_CG::Event</code>) This property provides support for static architectures found in hard real-time and safety-critical systems without memory management capabilities during run time. All instances of events are dynamically allocated during initialization. Once allocated, the event queue for a thread remains static in size. Triggered operations use the properties defined for events. When the memory pool is exhausted, an additional amount, specified by the <code>AdditionalNumberOfInstances</code> property, is allocated. Memory pools for classes can be used only with the Flat statechart implementation scheme.
AdditionalNumberOfInstances	<code><lang>_CG::Class</code> <code>CG::Event</code>	Specifies the number of instances or events for which memory is allocated when the current pool is empty.
ProtectStaticMemoryPool	<code>CG::Class/Event</code>	Determines whether to protect access to the allocated memory pool using an operating system mutex to provide thread safety.

Property	Subject and Metaclass	Description
MaximumPendingEvents	CG::Class	Specifies the maximum number of events that can be simultaneously pending in the event queue of the active class.

See the definition provided for each property on the applicable **Properties** tab of the Features window.

Events generated in an interrupt handler should not rely on rescheduling and therefore should not cause the use of an operating system mutex. In other words, `ProtectStaticMemoryPool` should be `False`. It is up to you to make sure that the memory pool is not thread-safe. The animation framework is not subject to this restriction.

The framework files include reachable code for both dynamic and static allocation. Actual usage is based on the generated code. Use of the `GEN` and `gen` macros is the same for both modes.

Implementation of fixed and bounded relations with static arrays via the `<lang>_CG:Relation::ImplementWithStaticArray` property implicitly uses static allocation apart from initialization.

Static Memory Allocation Algorithm

Rational Rhapsody implements static memory allocation by redefining the `new` and `delete` operators for each event and class that use local heaps. The memory allocator holds enough memory to accommodate n instances of the specific element, where n is the value of the `BaseNumberOfInstances` property. The memory allocation is performed during system construction and uses dynamic memory. The memory allocator uses a LIFO (stack) algorithm, as follows:

- ◆ Claimed memory is popped off the top of the stack, and the top pointer is moved to point to the next item.
- ◆ Returned memory is pushed onto the top of the stack, and the top pointer is moved to point to the returned item.

The generated class (whether a class, event, or triggered operation) is instrumented to introduce additional code needed for use by the memory allocator (specifically the next pointer).

Attempts to instantiate a class whose memory pool is exhausted result in a call to the `OMMemoryPoolIsEmpty()` operation (in which you can set a breakpoint) and in a tracer message. Failure to instantiate results in a tracer message.

Containment by Value via Static Architecture

Rational Rhapsody normally generates containment by reference rather than containment by value for 1-to-MAX relationships, regardless of the relationship type (set in the diagram). However, the static architecture feature enables you achieve the effect of containment by value by defining the maximum number of class instances and event instances via the static architecture properties, and thus avoiding the use of the default, non-deterministic `new()` operator.

Static Memory Allocation Conditions

If the application uses static memory allocation, the checker verifies that the following conditions are met:

- ♦ The maximum number of class instances is non-zero.
- ♦ The statechart implementation is flat.
- ♦ The `new` and `delete` operators are explicitly specified for each event and class using local heaps.

Reports include limits set for the number of instances.

All properties are loaded and saved as part of the element definition in the repository.

Static Memory Allocation Limitations

The following limitations apply to static memory allocation:

- ♦ Support is not yet provided for allocation of arrays. Instances must be allocated one-by-one. This is because of the (unknown) memory overhead associated with allocation of arrays.
- ♦ The `tm()` timeout function is not yet supported.

Using Standard Operations

Standard operations are class operations that are standardized by a certain framework or programming style. Notable examples are serialization methods that are part of distribution frameworks (see [Standard Operations Example](#)), and copy or convert constructors. Such operations generally apply aggregate functions to class features (such as attributes, operations, and superclasses). The following sections describe how to add standard operations to the code generated for classes and events.

Applications for Standard Operations

Standard operations can be used to follow these steps:

- ◆ Create event serialization methods.
- ◆ Support canonical forms of classes.
- ◆ Make a class persistent.
- ◆ Support a particular framework.

Event Serialization

1. An Event class is instantiated, resulting in a pointer to the event.
2. The event is queued by adding the new event pointer to the event queue for the receiver.

Once the event has been instantiated and added to the event queue of the receiver, the event is ready to be “sent.” The success of the send operation relies upon the assumption that the memory address space of the sender and receiver are the same. However, this is not always the case.

For example, the following are some examples of scenarios in which the sender and receiver memory address spaces are most likely different:

- ◆ The event is sent between different processes in the same host.
- ◆ The event is sent between distributed applications.
- ◆ The sender and receiver are mapped to different memory partitions.

One common way to solve this problem is to marshall the information. *Marshalling* means to convert the event into raw data (or serialize it), send it using frameworks such as publish/subscribe, and then convert the raw data back to its original form at the receiving end. High-level solutions, such as CORBA, automatically generate the necessary code, but with low-level solutions, you must take explicit care. Standard operations enable you to specify to a fine level of detail how to marshall, and unmarshall, events and instances.

Canonical Forms of Classes

Many style guidelines and writing conventions specify a list of operations to include in every class. Common examples are copy and convert constructors, and the assignment operator. Rational Rhapsody enables you to define standard operations such as these for every class.

Persistence

Making classes persistent is usually achieved by adding an operation with a predefined signature to every class. This can be done with a standard operation.

Support for Frameworks

Many object-oriented frameworks are used by deriving subclasses from a base class and then overriding the virtual operations. For example, MFC constructs inherit from the `CObject` class and then override operations such as the following example:

```
virtual void Dump(CDumpContext& dc) const;
```

This is another example of something that can be done with standard operations.

You can add framework base classes using the `<lang>_CG::Class::AdditionalBaseClasses` property.

Creating Standard Operations

For every standard operation defined, you must specify an operation declaration and definition. This is done by adding the following two properties to the `site.prp` file to specify the necessary function templates:

- ◆ `<LogicalName>Declaration` specifies a template for the operation declaration
- ◆ `<LogicalName>Definition` specifies a template for the operation implementation

For example, with a logical name of `myOp`, you would add the following property (using the `site.prp` file or the COM API (VBA)):

```
Subject CG
  Metaclass Class
    Property myOpDeclaration MultiLine ""
    Property myOpDefinition MultiLine ""
  end
```

You add all of the properties to be associated with a standard operation to the `site.prp` file under their respective CG subject and metaclasses. All of these properties should have a type of `MultiLine`.

Template Keywords

The template can contain the following keywords:

- ◆ `$Attributes` where for every attribute in the class, this keyword is replaced with the contents of the template specified in the `CG::Attribute::<LogicalName>` property for the attribute. For example, `<lang>_CG::Attribute::myOp`.
- ◆ `$Relations` where for every relation in the class, this keyword is replaced with the contents of the template specified in the `CG::Relation::<LogicalName>` property for the relation. For example, `<lang>_CG::Relation::myOp`.
- ◆ `$Base (visibility)` where for every superclass/event, this keyword is replaced with the content of the template specified in the superclass/event property `CG::Class/Event::<logicalName>Base`. For example, `CG::Class::myOpBase`.
- ◆ `$Arguments` where for every argument in the class, this keyword is replaced with the contents of the template specified in the `CG::Argument::<LogicalName>` property for the argument. For example, `<lang>_CG::Argument::myOp`.

Note

If you do not set the new location (`CG.Argument.<standardOpName>`), the old location (`CG.Event.<standardOpName>Args`) is still valid.

When expanding the properties, you can use the following keywords:

- ◆ `$(Name)` specifies the model name of the class (attribute, relation, operation, or base class), depending on the context of the keyword. If applicable, `$Type` is the type associated with this model element.
- ◆ `$ImplName` specifies the name of the (generated) code. This is usually the data member associated with this model element.

This is useful in C, where the logical name can be replaced in the generated code to a more complex name.

In addition, you can create custom keywords that relate to any property in the context using the convention `$<property name>`. For example, if you added a property `CG::Class::testA`, you can relate to the property content in `CG::Class::myOpDefinition` by `$testA`.

Note that this feature, combined with stereotype-based code generation, lets a power user define a set of standard operations, associate them to stereotypes, and lets other members of the team apply the stereotypes (getting the standard operations without worry over the properties). See [Customize Code Generation with Stereotypes](#).

Standard Operations Example

If you need to implement a serialization of events for all the events in a certain package, the `CG::Event::StandardOperations` property for the package will contain “serialize, unserialize.” If the serialization is done by placing the event argument values inside an `stringstream`, the corresponding properties and their values are as follows:

CG::Event::SerializeDeclaration

```
public:
    stringstream & serializer(stringstream & theStrStream)
    const;
```

CG::Event::SerializeDefinition

```
stringstream & $Name::serialize(stringstream & theStrStream) {
    $Base
    $Arguments
    return theStrStream;
}
```

CG::Event::SerializeArgs

```
theStrStream << $Name;
```

CG::Event::SerializeBase

```
$Name::serialize(theStrStream);
```

If you defined an event VoiceAlarm with two arguments, severity and priority, and VoiceAlarm inherits from Alarm, the resulting code for the template would be as follows:

```
strstream & VoiceAlarm::serialize(strstream & theStrStream) {  
    Alarm::serialize(theStrStream);  
    theStrStream << severity;  
    theStrStream << priority;  
    return theStrStream;  
}
```

The same process is done for the unserialize part.

Customize Code Generation with Stereotypes

Stereotypes allow extension of the UML metamodel by “typing” different model elements. Certain stereotypes are predefined in the UML, others might be user-defined. Rational Rhapsody properties are name-value pairs that allow, among other things, customization of code generation. Property settings can be applied to a stereotype, and when that stereotype is applied to a model element, the properties for the stereotype are transferred onto the model element. For more information on stereotypes, see [Stereotypes](#).

A stereotype is typically defined as part of a profile, which is a special package primarily containing stereotypes and tags. For more information on profiles, see [Profiles](#).

Example of Using Stereotypes to Customize Code Generation

A Rational Rhapsody project has a Whole and Part class and uses an ordered container (OMList) to keep track of the Part objects. For this project, you create a profile called CustomCG and define an stereotype called OrderedContainer. You make the OrderedContainer stereotype only applicable to AssociationEnd elements. For the OrderedContainer stereotype, you set the CG::Relation::Ordered property to Checked. For the AssociationEnd, you set the stereotype to be OrderedContainer. When you generate code, you will see that OMList is used.

Providing Support for Java Initialization Blocks

Java initialization blocks are used to perform initializations before instantiating a class, such as loading a native library used by the class. You can use standard operations to provide a convenient entry point for this language construct, as follows:

1. Open the `site.prp` file and add the following property:

```
Subject CG
  Metaclass Class
    Property StandardOperations String "InitializationBlock"
  end
end
```

2. Open the `siteJava.prp` file and add the following property:

```
Subject JAVA_CG
  Metaclass Class
    Property InitializationBlockDeclaration MultiLine ""
  end
end
```

3. For every class that needs an initialization block, enter the text in the `InitializationBlockDeclaration` property for the class. For example:

```
static {
  System.loadLibrary("MyNativeLib");
}
```

The text is entered below the class declaration.

Statechart Serialization

Rational Rhapsody provides a mechanism for serialization of reactive instances. By setting a number of Rational Rhapsody properties, you can have methods added to the generated code, which you can then use to implement serialization.

Note

This feature is available only for C and C++ code.

Generating Methods for Serialization

The property `[C][CPP]_CG::Statechart::StatechartStateOperations` determines whether the code is generated for this feature. The possible values for this property are:

- ♦ **None** (default value) where code is not generated for the feature
- ♦ **WithoutReactive** where Rational Rhapsody does not generate calls to `OMReactive`
- ♦ **WithReactive** where Rational Rhapsody generates calls to `OMReactive`

Note

In C++, when using inheritance, a hierarchy of classes must use the same value for the property `CPP_CG::Statechart::StatechartStateOperations`.

Serialization Properties

The following properties, listed for C++ and C respectively, are related to the use of the serialization methods:

- ♦ `CPP_CG::Framework::ReactiveGetStateCall`
Default value is `OMReactive::getState()`;
Defines the prototype of the `getState` framework method.
- ♦ `CPP_CG::Framework::ReactiveSetStateCall`
Default value is `OMReactive::setState(p_state)`;
Defines the prototype of the `setState` framework method.
- ♦ `CPP_CG::Framework::ReactiveStateType`
Default value is `unsigned long`
Defines the `oxfstate` type.
- ♦ `C_CG::Framework::ReactiveGetStateCall`

Default value is `RiCReactive_getState(ric_reactive);`

Defines the prototype of the `getState` framework method.

- ◆ `C.CG::Framework::ReactiveSetStateCall`

Default value is `RiCReactive_setState(ric_reactive, p_state);`

Defines the prototype of the `setState` framework method.

- ◆ `C.CG::Framework::ReactiveStateType`

Default value is `long`

Defines the `oxfstate` type.

Methods Provided for Implementing Serialization

When the properties are set to generate the relevant code, the following public virtual member functions are generated for reactive classes:

Note

For the C functions, the prefix `<class name>` refers to the “class” in the model, for which the statechart was created.

- ◆ `virtual int getStatechartSize()` for C++

`int <class name>_getStatechartSize(<class name>* me)` for C

Returns the number of variables that the statechart uses. This function should be used to allocate the state vector that is passed to the function `getStatechartStates`.

- ◆ `virtual void getStatechartStates(int stateVector[], unsigned long& oxfReactiveState) const` for C++

`void <class name>_getStatechartStates(const <class name>* const me, int stateVector[], unsigned long* oxfReactiveState)` for C

Fills `stateVector` with the current statechart state, and sets `oxfReactiveState` based on the `OMReactive` internal state.

The type of `oxfReactiveState` is taken from the property `[C][CPP].CG::Framework::ReactiveStateType`.

The type of `stateVector` is taken from the property `CG::Statechart::FlatStateType` (default value is `int`).

In `withoutReactive` mode, the last argument is eliminated and the function prototypes become:

- virtual void getStatechartStates(int stateVector[])
for C++
 - void <class name>_getStatechartStates(const <class name>* const
me, int stateVector[]) for C
 - ◆ virtual void setStatechartStates(int stateVector[], unsigned long*
oxfReactiveState) for C++
 - void <class name>_setStatechartStates(<class name>* const me, int
stateVector[], unsigned long* oxfReactiveState) for C
- Set the reactive instance states as well as the OMReactive internal state.
- The type of oxfReactiveState is taken from the property
[C][CPP]_CG::Framework::ReactiveStateType.
- The type of stateVector is taken from the property
CG::Statechart::FlatStateType (default value is int).

Note

If setStatechartState is used during animation, then the instance statechart will not display new states. In order to “refresh” the statechart, you can switch to Silent animation mode, run the animation, and then switch back to Watch animation mode.

Generating Classes as Structs in C++

When generating C++ code, Rational Rhapsody generates classes in your model as C++ classes in the code. While this is the default behavior, it is also possible to have Rational Rhapsody generate classes as structs in your C++ code. To have one or more classes generated as structs, modify the value of the property CPP_CG::Class::GenClassAsStruct.

Components-based Development in C

We enable component-based development in Rational Rhapsody Developer for C by introducing code generation support for interfaces and ports.

A class might realize an interface, that is, provide an implementation for the set of services it specifies (that is, operations and event receptions). As in Rational Rhapsody Developer for C++ and Rational Rhapsody in Java, you use a realization relationship to indicate that a class is realizing an interface. In addition, an interface might inherit another interface, meaning that it augments the set of interfaces the superinterface specifies. You can specify interfaces, realize them, and connect to objects via the interfaces.

C users can take advantage of service ports that allows the passing of operations and functions via ports, in addition to passing events. Just like in C++, you can specify ports with provided and required interfaces. In addition, Rational Rhapsody 7.1 provides code generation support for standard UML ports in C and code generation of ports supports the initialization of links via ports. For more information about ports, see [Ports](#).

In this type of development in C, interfaces are treated as a specification of services (that is, operations) and **not** as inheritance of data (attributes). Also, in this type of development in C, realization (as opposed to inheritance) is used to distinguish between realizing an interface and inheriting an interface/class.

As of Rational Rhapsody 7.1, code generation supports realizing interfaces in C. This means interfaces and ports specified in a C model will be implemented by the code generator. This means code generation generates:

- ◆ Code for a C interface (a class with “pure virtual operations”)
 - Virtual tables with function pointers
 - Relay code from the interface to the realizing class according to the virtual table
- ◆ The “realization code” for the realizing class
 - Aggregating the interface
 - Initializing the virtual table
- ◆ Links between objects that instantiate associations to the interface

Action Language for Code Generation

The following syntax is used for C code generation support for interfaces and ports. In these examples, we have an interface x, an operation f, a port p, and a class A.

Calling an operation via a C interface

```
[Interface]_[Operation]([object realizing the interface]  
[, argList])
```

Example: To call operation x_f (object realizing the interface, port number), where the port number is 5, do:

```
x_f(me->itsl, 5);
```

Sending an event via a C interface

```
RiCGEN_[Interface]([object realizing the interface], [event([argList]])]
```

Example: To send event RiCGEN_l(object realizing the interface, port number), do:

```
RiCGEN_l(me->itsl, evt());
```

Calling an operation via a C port

```
[Interface]_[Operation](OUT_PORT([class], [port], [interface])  
[, argList])
```

Example: To call operation x_f (object realizing the port, port number), where the port number is 5, do:

```
x_f(OUT_PORT(A, p, x), 5);
```

Sending an event via a C rapid port

```
RiCGEN_PORT([pointer to port], [event])
```

Example: To send event RiCGEN_PORT(object realizing the port, event), do:

```
RiCGEN_PORT(me->p, evt());
```

Sending an event via a C rapid port using ISR

```
RiCGEN_PORT_ISR([pointer to port], [event])
```

Example: To send event RiCGEN_PORT_ISR, do:

```
RiCGEN_PORT_ISR(me->p, evt());
```

Querying the port through which the event was received

```
RiCIS_PORT([object], [pointer to port])
```

Example: To query port RiCIS, do:

```
RiCIS_PORT(me, me->p);
```

Sending an event via a C non-rapid port

```
RiCGEN_PORT_I([class], [port], [interface], [event([argList]])]
```

Example: To send event RiCGEN_PORT_I(object realizing the port, event), do:

```
RiCGEN_PORT_I(A, p, x, evt());
```

C Optimization

Note the following information:

- ♦ If a port has only provided interfaces, or just required interfaces, code generation generates a single additional inner part.
- ♦ Ports that are purely reactive are implemented as rapid ports. For more information about rapid ports, see [Using rapid ports](#).

Backward Compatibility

To enable interface realization for models made before Rational Rhapsody 7.1, you must set the `C_CG::Class::InterfaceGenerationSupport` property to `Checked`. The choices are as follows:

- ◆ `Checked` means interfaces are generated with virtual tables and can be realized.
- ◆ `Cleared` means the `Interface` stereotype is ignored. Meaning you can turn off the interface generation ability.

See the definition provided for the property on the applicable **Properties** tab of the Features window.

Note the following information:

- ◆ If the `C_CG::Class::InterfaceGenerationSupport` property is set to `Cleared` for at least one of the interfaces in the contract for the port, the port is treated as a rapid port.
- ◆ `C_CG::Class::InterfaceGenerationSupport` is a backward compatibility property only and is not listed in new models.

Limitations

Note the following code generation limitations:

- ◆ There is no general inheritance support.
- ◆ Ports code is generated into separate files (meaning that there are no nested classes in Rational Rhapsody Developer for C).
- ◆ An inheritance from a class to an interface is interpreted as a realization.
- ◆ No code is generated for a `<<friend>>` dependency to the template class.

Customize C code generation

One of the key features of Rational Rhapsody is the ability to generate code based on a Rational Rhapsody model. There are two primary methods you can use to customize code generation:

- ♦ **Modifying the values of various properties in Rational Rhapsody.** This method is available for all Rational Rhapsody versions (C, C++, Java, and Ada). These properties are found under the CG and <lang>_CG subjects (for example, `CG::Package::UseAsExternal` and `JAVA_CG::Dependency::SpecificationEpilog`).
- ♦ **Using rules.** You might want to use this method if you have significant changes in the generated code where it is not enough to use properties and you want to have a starting point that is the out-of-the-box code generation. This method is available only for Rational Rhapsody Developer for C.

Note: You can also write your own code generator with the use of the RulesComposer tool. Note that you must have a valid license to be able to use this tool.

The using-rules method is described in detail in this section, including the conceptual basis for this customization mechanism, as well as specific instructions for customizing code generation.

Note that both methods let you control the content and appearance of the generated code. These two mechanisms co-exist and can be referred to as basic (using properties) and advanced (using rules) customization, respectively.

Code customization concepts

The process of converting a generic UML model into code can be divided into the following phases:

1. **Transformation.** The transformation phase of the original model into a refined model takes into account the elements of the specific programming language in which the code will be generated. In Rational Rhapsody, this is called “Simplification” and the properties related to it begin with the word “Simplify” (for example, `C_CG::ModelElement::SimplifyAnnotations`).
2. **Writing.** This is the conversion of the refined model into valid code in the chosen target language.

Customizing code generation

The customization feature is available only for Rational Rhapsody Developer for C.

When you instruct Rational Rhapsody to generate code, Rational Rhapsody can take a number of different paths, depending on the value of the property `C_CG::Configuration::CodeGeneratorTool`.

If `CodeGeneratorTool` is set to a value other than `Customizable`, Rational Rhapsody starts its standard internal code generation mechanism.

If `CodeGeneratorTool` is set to `Customizable`, Rational Rhapsody:

1. Creates a refined model from the original model. This model is referred to as the simplified model. (This step represents the *transformation* phase described in [Code customization concepts](#).)
2. Opens the external RulesPlayer code writer to create the code itself. (This step represents the *writing* phase described in [Code customization concepts](#).)

Note: You must have a valid license to be able to use the RulesPlayer code writer.

When code generation is running in Rational Rhapsody, you will see the following messages to show that the RulesPlayer is at work:

```
Loading external generator...
Invoking RulesPlayer
Evaluation of RiCWriter.
```

Both of these steps (creation of the simplified model and generation of code from the simplified model) can be customized, as described in [Customize the generation of the simplified model](#) and [Customizing the code writer](#).

Viewing the simplified model

When the property `C_CG::Configuration::CodeGeneratorTool` is set to `Customizable`, a simplified model is created automatically as the first step of the code generation process.

By default, the simplified model is not displayed in Rational Rhapsody. To have the simplified model displayed in the browser, set the property

`C_CG::Configuration::ShowCGSimplifiedModelPackage` property to `Checked`. Once you have done so, the next time you generate code, the simplified model will be added automatically at the top of the project tree in the Rational Rhapsody browser.

Customize the generation of the simplified model

Rational Rhapsody contains a default mapping that determines how model elements are treated when generating a simplified model.

Properties used for simplification

To change the way specific types of elements are handled, you modify the properties that control simplification. For each type of model element, there is a property that determines how it will be handled during the transformation of the model, for example, `SimplifyConstructors` and `SimplifyDestructors`.

Each of these properties can take any of the following values. Note that these values might not appear for every Simplify property.

- ♦ **None.** The element is ignored in the simplified model.
- ♦ **Copy.** The element is just copied from the original to the simplified model. It is not modified in any way.
- ♦ **Default.** Uses the standard simplification for this item, as defined in Rational Rhapsody.
- ♦ **ByUser.** Uses the customized simplification provided by the user. For details, see [User-provided simplification \(ByUser option\)](#).
- ♦ **ByUserPostDefault.** Uses the customized simplification provided by the user, but only after the Rational Rhapsody standard simplification for the element has been applied.

User-provided simplification (ByUser option)

If you have indicated in the property settings that a user-provided simplification is to be used for a given type of element, then the code generation process starts the user-provided code for transforming the model. The basics of this process are as follows:

- ♦ The user-provided transformation code is provided as a Rational Rhapsody plug-in.

- ◆ You add this plug-in information to the `rhapsody.ini` file, or provide the information necessary to have the plug-in run only for a certain profile.
- ◆ During the code generation process, Rational Rhapsody checks whether the user-provided code has implemented the relevant “simplify” interface for the element in question. (These interfaces are defined in the Rational Rhapsody API.)

Note: Rational Rhapsody provides you with sample projects that show the “simplify” interface. Look in the `<Rational Rhapsody installation path>\Samples\CustomCG Samples path`. For example, see the sample projects provided in the `Statechart_Simplifier_Writer` subfolder. You should review the `Readme.txt` file that accompanies each sample project for details about that project.

- ◆ If the user-provided code implements the “simplify” interface, your implementation is called.
- ◆ The user-provided transformation code uses the Rational Rhapsody API to directly modify the way model elements are transformed.

Customizing the code writer

The rules for rules-based code generation are contained in the Rational Rhapsody Developer for C Writer project. You customize the rules with the use of the RulesComposer tool. The RulesPlayer code writer generates code from the simplified model using these rules.

The following topic describes the steps that are required if you want to customize the rules used for generating code from the simplified model.

Note

For detailed information on how to use the RulesComposer, see the tutorial PDF provided with the installation of the product (look in `<Rational Rhapsody installation path>\Rhapsody\Sodius\RulesComposer\help\tutorial`). There is also a changes document for the RulesComposer that you might find useful (look in `<Rational Rhapsody installation path>\Rhapsody\Sodius\RulesComposer\help`).

Customizing the C rules

To customize the rules used to generate code for a simplified model:

1. Open RulesComposer from Rational Rhapsody. Choose **Tools > RulesComposer**.
2. When RulesComposer opens, the Rational Rhapsody Developer for C Writer project should already be open. If not, open the project manually by choosing **File > Import** in RulesComposer and selecting the <Rational Rhapsody installation path>\Share\CodeGenerator\GenerationRules\LangC\RuleSet\RicWriter folder. When you choose this directory, Eclipse will automatically load the RicWriter project that it contains.

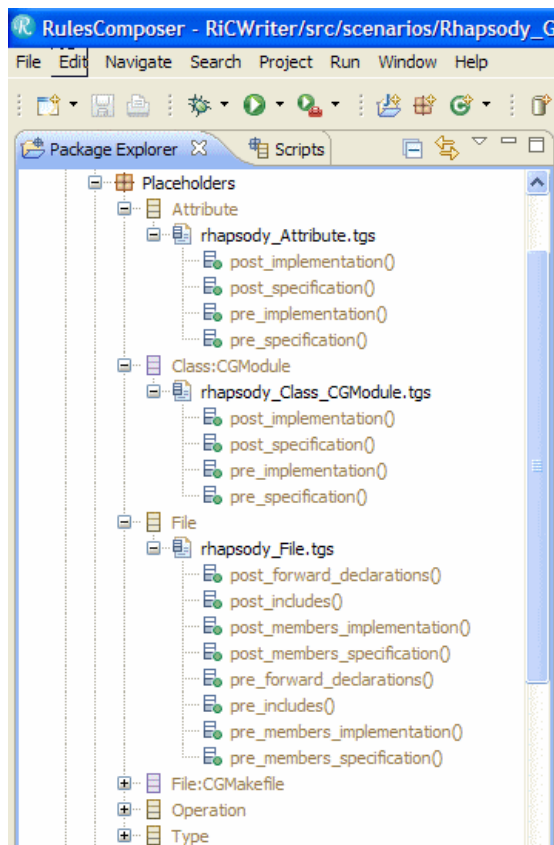
Note: The project is read-only by default. In order to modify the rules, you will need to change the relevant files to read-write.
3. Once the project is open, make your changes to the rules and script files (.java, .tgs). These are located in the src subfolder. Notice the **Placeholders** package. It contains hooks provided in the default rules for user customization. These hooks are empty scripts where you can enter code. These scripts are run from the existing rules at the appropriate time during code generation. For more information, see [Placeholders package](#).
4. Save your changes to the Rational Rhapsody Developer for C Writer project.
5. After saving your changes, you can test them by selecting **Run** in Eclipse. This will take your updated rules and apply them to the model that is currently open in Rational Rhapsody. You can then look at the generated code to verify that the new rules had the intended effect.

Note

The updated rules can only be used to generate code if there is an existing simplified model to which they can be applied. This means that you must have already generated code with Rational Rhapsody at least once for the model with the property CodeGeneratorTool set to Customizable and the property ShowCGSimplifiedModel set to Checked. (When the property ShowCGSimplifiedModel is set to Cleared, the simplified model is deleted after code generation has been completed. So in such a case, you would not have a simplified model to which the updated rules could be applied.)\

Placeholders package

The **Placeholders** package, as *partially* shown in the following figure, contains hooks provided in the default rules for user customization. These hooks are empty scripts where you can enter code. These scripts are run from the existing rules at the appropriate time during code generation. For example, you can insert code in the **post_implementation()** script in the **Attribute** placeholder group. This script runs after implementation of the attribute rules.

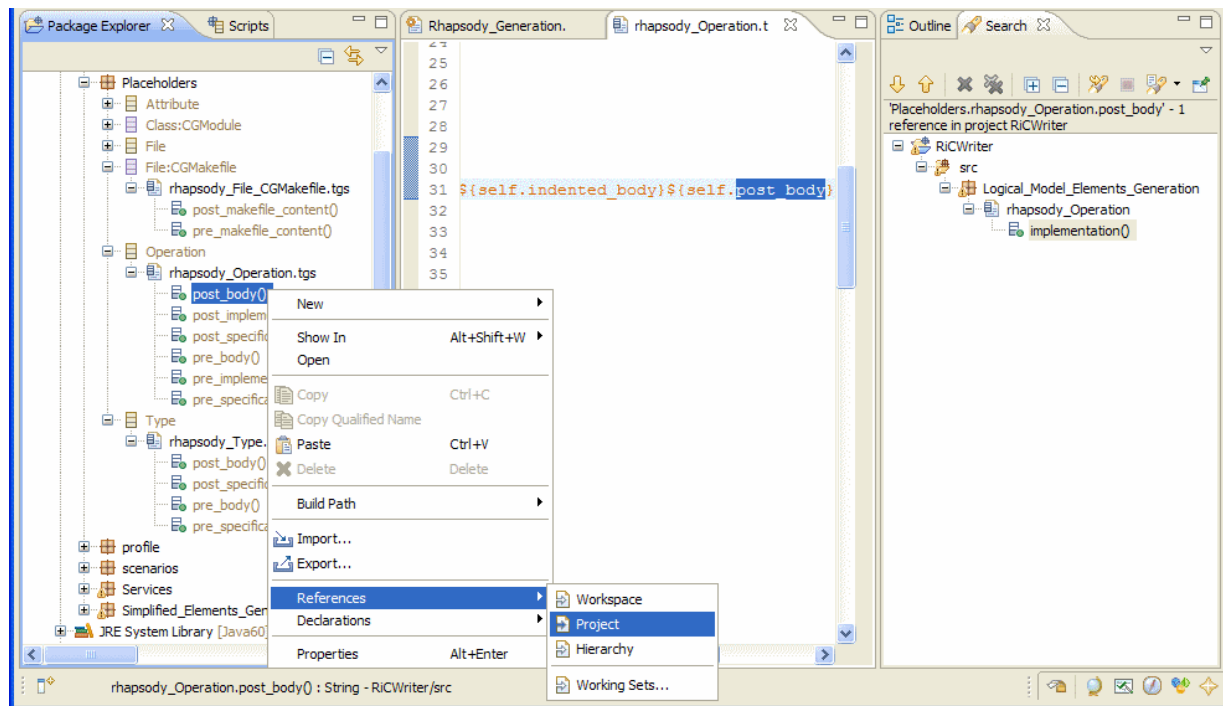


You can go to the reference for a script to find where the script is called.

To find a reference for a particular script:

1. Right-click the script on the RulesComposer Project Explorer, select **References**, and then select **Workspace**, **Project**, or **Hierarchy** (see left third of the following figure), which opens a Search tab in the RulesComposer.
2. Double-click the found script on the **Search** tab (see right third of the figure), which opens the applicable file in the middle of the RulesComposer window (see middle third of the figure).

Note: For illustrative purposes, the following figure shows the results after the above steps have been completed once.



Deploying the changed rules

Once you are satisfied that the changes to the rules have the intended effect on code generation, the last step is to deploy the rules to the .jar file that Rational Rhapsody uses when it performs customized code generation.

To deploy the changed rules:

1. In RulesComposer, select **File > Export**.
2. For the export destination, select **RulesComposer > Deployable RulesComposer configuration**.
3. Click **Next**.
4. For the export directory, select <Rational Rhapsody installation path>\Share\CodeGenerator\GenerationRules\LangC\CompiledRules

Note: By default, the .jar file containing the existing rules (RiCWriter.jar) is read-only. Make sure to change this attribute before attempting to export the updated rules.

5. Under Export Options, select **Deploy JAR file**.
6. Click **Finish**. The new rules will be saved as RiCWriter.jar.

Note: Rational Rhapsody looks for the compiled rules in the filename given for the property C.CG::Configuration::GeneratorRulesSet. The default value for this property is \$OMROOT\CodeGenerator\GenerationRules\LangC\CompiledRules\RiCWriter.classpath.

7. To have Rational Rhapsody implement the rules in the updated .jar file, you must close and then re-open Rational Rhapsody after you have exported the .jar file.

Reverse engineering

Reverse engineering lets you import legacy C, C++, and Java code into a Rational Rhapsody model. It extracts design information from the code constructs found in the source file and builds a model corresponding to the wanted design, as much as possible.

Note

You can also reverse engineer Ada code. In Rational Rhapsody in Ada, choose **Tools > Reverse Engineer Ada Source Files**. For documentation about reverse engineering in Ada, see the documentation provided in <Rational Rhapsody installation path>\Sodius\RiA_CG\AdaRevEng\help.

Once you generate a Rational Rhapsody model from legacy code in the reverse engineering process, further edits to the model or to the code become synchronized in the roundtripping process thereafter. See [Roundtripping](#).

Reverse engineering restrictions

The following restrictions need to be followed when planning and executing a reverse engineering operation:

- ♦ Rational Rhapsody does not import makefiles. Even if a configuration already has an existing makefile, Rational Rhapsody generates another makefile and does not use the original.
- ♦ Files to be imported must contain compilable code with no syntax errors.
- ♦ To change the location of where reverse engineering information is retrieved and saved requires the engineer to close the Reverse Engineering window and change the active configuration for the model in Rational Rhapsody.
- ♦ The tree view does not support displaying files selected from multiple drives. In those cases, Rational Rhapsody does not allow you to switch from tree view to the flat view.

Reverse engineering legacy code

To reverse engineering existing code for use in a Rational Rhapsody project, begin by launching the reverse engineering tool:

1. Open the model into which you want to import legacy code and set which configuration you want to be the active configuration. See [Setting the active configuration](#).
2. Select **Tools > Reverse Engineering**. A display message displays to confirm that reverse engineering settings will be retrieved from/saved to and for which active component/configuration, as shown in the following figure.

Notice that simultaneously the Output window opens in the Rational Rhapsody main window.

3. Click **Continue**. The Reverse Engineering window opens.

Reverse engineering tool features

In the flat view, the Reverse Engineering window contains the following controls:

- ♦ Flat View button  and Tree View button , which you can use to toggle between the two views. A flat view shows folders and files in a list structure while a tree view shows a tree structure.
- ♦ **Add Files** button lets you add to the import file individual files (such as .h files for C and C++ projects) that you want to reverse engineer.
- ♦ **Add Folder** button lets you add to the import file a folder that contains files that you want to reverse engineer.
- ♦ **Remove** button, which is available only when applicable, lets you remove one or more highlighted items on the import list.
- ♦ **Options** group with **Visualization Only (Import as External)** and **Populate Object Model Diagrams** check boxes.

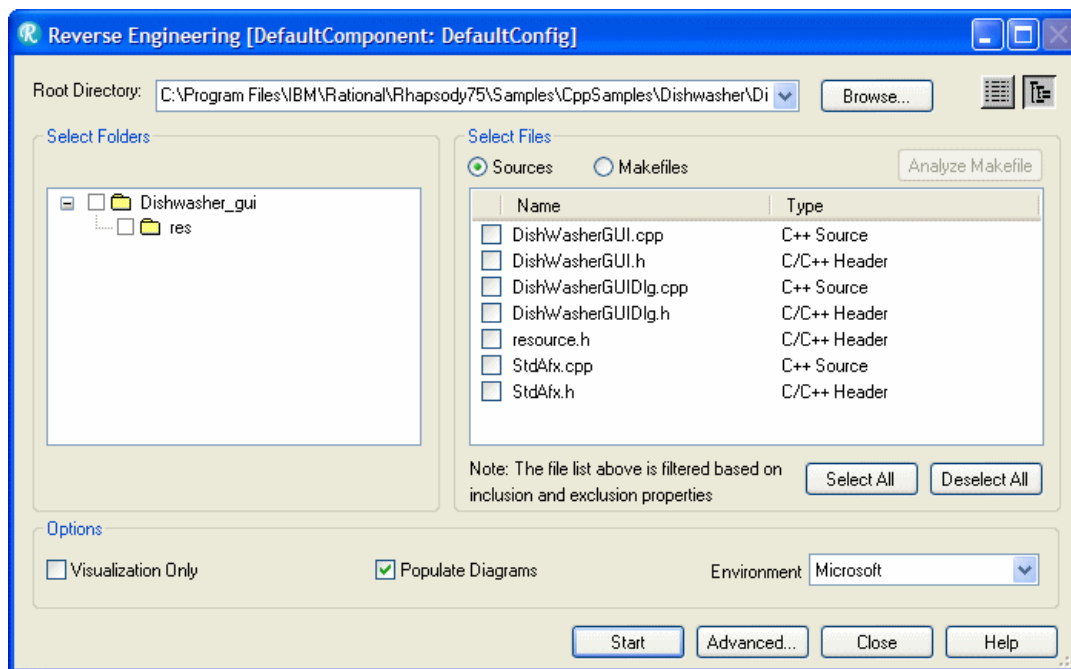
Select **Visualization Only (Import as External)**, when you want Rational Rhapsody to add as external elements all the elements created in Rational Rhapsody during reverse engineering (the CG: :Class/Package/Type: :UseAsExternal property is set to Checked.) This means that while you can see the code using pictures and you can relate to it, the code is still maintained outside Rational Rhapsody and is not generated by Rational Rhapsody. This property is overridden for all packages (but not their contained elements, such as classes, files, types, unless the contained elements are also packages; and in that case they will also have their CG: :Package: :UseAsExternal property set to Checked). Note that when you select or clear the **Visualization Only (Import as External)** check box, this same check box is automatically set the same way on the **Mapping** tab of the Reverse Engineering Options window. See [Mapping classes to types and packages](#).

Select **Populate Object Model Diagrams** when you want imported object model diagrams to be automatically created in your project. Note that when you select or clear the **Populate Object Model Diagrams** check box, this same check box is automatically set the same way on the **Model Updating** tab of the Reverse Engineering Options window. See [Updating existing packages](#).

- ◆ **Start** button lets you start the reverse engineering process. This button changes to **Stop** during the processing.
- ◆ **Advanced** button gives you access to the Reverse Engineering Options window.
- ◆ **Close** button lets you close a window without invoking whatever process is associated with the window.

Displaying files in a tree view

To locate and display the files for the reverse engineering in a tree structure, click the Tree View button  to display the files, as shown here.



1. Click the **Browse** button to open the Browse to Folder window and browse to the folder that has the legacy code you want to reverse engineer, as shown in the following figure, and click **OK**.

2. Select the files you want to reverse engineer. You can select a check box next to one or more individual files. You can also use the **Select All** and **Deselect All** buttons. In the **Select Folders** box, you can select the check box next to a folder to select all the files in that folder. Note that the appearance of the check boxes in the **Select Folders** box have these definitions:

- ☒ All items (files and subfolders) in the folder has been selected
- ☒ Some items have been selected
- ☐ No items have been selected

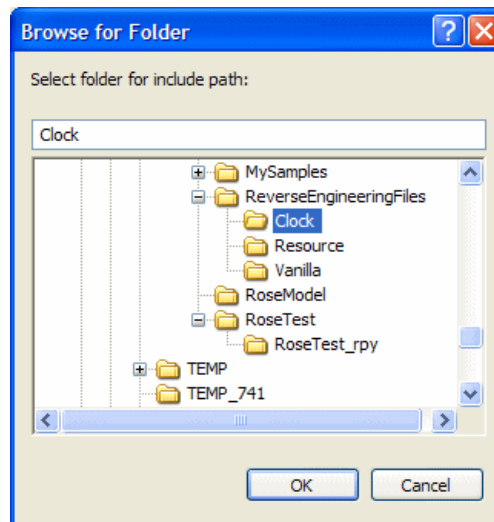
3. Select any of the check boxes in the **Options** group, as applicable.

Note: To specify the filename extensions that should be used to filter files in the Reverse Engineering window, use the `<lang>_ReverseEngineering::Main::ImplementationExtension` and `<lang>_ReverseEngineering::Main::SpecificationExtension`. In addition, note that files that are matching the `ReverseEngineering::Main::ExcludeFilesMatching` property will be filtered out in contrast to the other two properties mentioned previously. For more about the `ReverseEngineering::Main::ExcludeFilesMatching` property, see [Excluding particular files](#). Also, see the definition provided for each property on the applicable **Properties** tab of the Features window.

Displaying files in a flat view

If you are using the flat view of the Reverse Engineering window, to display files in a flat view:

1. To select individual files, click **Add Files** to open the Open window open and browse to the files you want to import. Then click **Open**. Rational Rhapsody displays the files.
2. To include all the files in a folder for reverse engineering, click **Add Folder** to open the Browse for Folder window, as shown in the following figure. This means you want to reverse engineer all the files (that is possible) in the folder. Click **OK**.



The folder is added to the Reverse Engineering window. When you add files or folders to this window, the list maintains itself from session to session so you can maintain the history of the file list.

3. To specify options for this reverse engineering session, click the **Advanced** button to open the Reverse Engineering Options window contains the following tabs:
 - ♦ **Preprocessing** (see [Defining preprocessor symbols](#))
 - ♦ **Input** (see [Analyzing #include files](#))
 - ♦ **Mapping** (see [Mapping classes to types and packages](#))
 - ♦ **Filtering** (see [Specifying reference classes](#))
 - ♦ **Misc** (see [Miscellaneous reverse engineering options](#))
 - ♦ **Process** (see [Reverse engineering error handling](#))
 - ♦ **Model Updating** (see [Updating existing packages](#))
 - ♦ **Log** (see [Reverse engineering message reporting](#))

4. Click **OK** to close the Reverse Engineering Options window.
5. On the Reverse Engineering window, click **Start** to begin the import. Notice that the label for this button changes to **Stop**.
6. Click **OK** to confirm your requested action.
7. Wait while the reverse engineering process completes. When the **Stop** button changes to **Start** again, you can click **Close** to close the Reverse Engineering window.

Note: Depending on the number of files in your folders and the size of your files, the reverse engineering process can take some time. You might want to do a few files first to experience the process before you do whole folders or larger or multiple files at once.

8. You should see information about the reverse engineering in the Output window and in your Rational Rhapsody browser.

Reverse engineering messages in the Output window

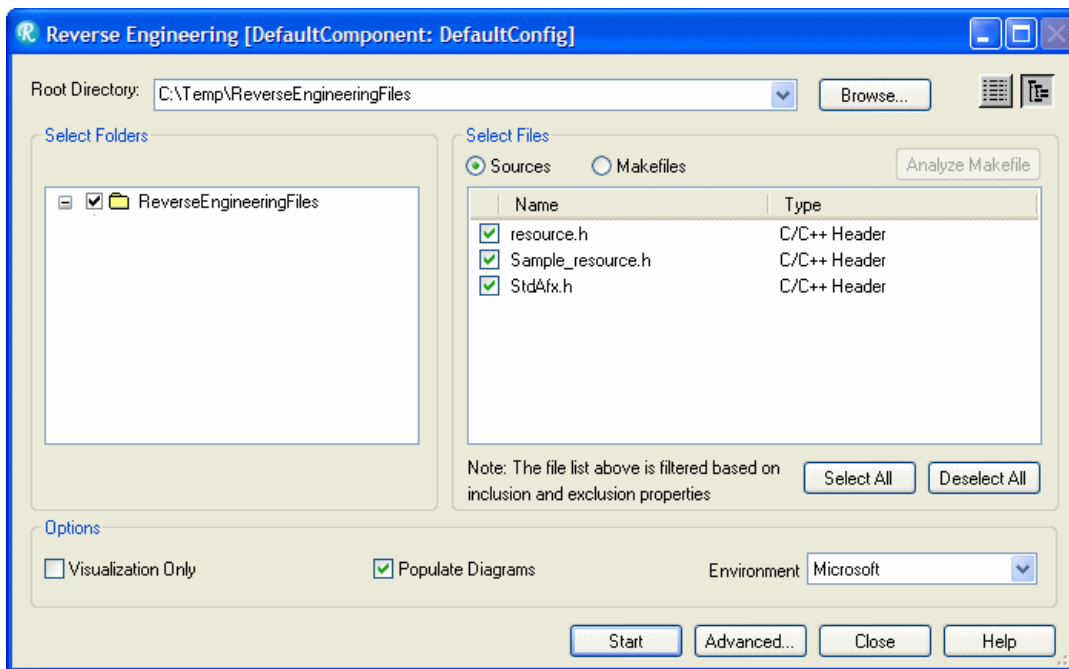
Messages are displayed in the Output window that indicate which files and constructs are being analyzed and which design elements are being added to the model. Note that the reverse engineering tool does not report on every item being parsed, nor does it report on ignored constructs. You can specify how or whether certain events are reported using the Log and Process options. See [Reverse engineering message reporting](#) and [Reverse engineering error handling](#) for more information.

Note that once a file has been successfully imported, Rational Rhapsody does not provide a means to delete information associated with an imported file. If you need to delete imported elements, do it manually from the Rational Rhapsody browser. See [Deleting elements](#).

Initializing the Reverse Engineering window

If the properties for the active configuration already contain information about a reverse engineering configuration (list of files and other options), the Reverse Engineering window will be initialized with this information by default. This initialization includes the following information:

- ◆ The directory tree displays and the selection status is marked according to the list of files saved with the `ReverseEngineering::Main::Files` property. Be sure to include the backslash (for Windows systems) at the end of a path (for example: `C:\TEMP2\ReverseEngineeringFiles\`).
- ◆ When you open the Reverse Engineering tool, the file list will be created and selection status marked accordingly for each item on the directory tree. Notice the three files in the **Select Files** box.



- ◆ The options in the Reverse Engineering window and the Reverse Engineering Options window are initialized accordingly.

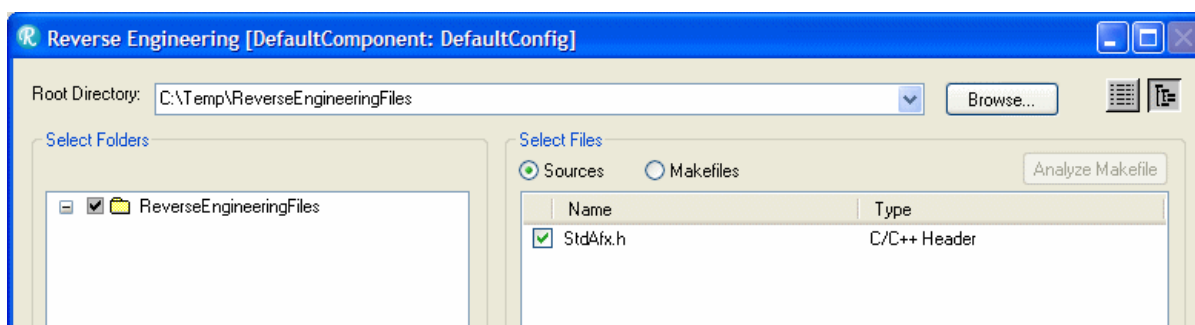
Excluding particular files

If you want to exclude particular files or folder from being reverse engineered, you can use the `ReverseEngineering::Main::ExcludeFilesMatching` property. You can assign one or more comma-separated wildcard expressions to this property, as shown in the following figure. The files or folders that are matched using these wildcard expressions will not be reverse engineered.

Note

This property affects only the tree view of the Reverse Engineering window.

With the `ExcludeFilesMatching` property set to `res*`, when you open the Reverse Engineering tool (**Tools > Reverse Engineering**), the Reverse Engineering window (tree view) looks like the following figure, which shows only one file in the **Select Files** box to be reverse engineered. Compare this with the figure in [Initializing the Reverse Engineering window](#) that shows the same window but without the `ExcludeFilesMatching` property set.



Analyzing makefiles

The Reverse Engineering window requires you to provide the list of files to reverse engineer. In addition, the Advanced Options window allows you to specify other settings for reverse engineering, such as include paths. To facilitate the entry of this information, if you have a makefile for your project, you can just provide Rational Rhapsody with the location of the makefile and it will analyze the makefile in order to retrieve the list of source files and any other settings that are relevant for reverse engineering. To have Rational Rhapsody analyze a makefile:

1. Open the Reverse Engineering window (choose **Tools > Reverse Engineering**).
2. In the Select Files section, select the Makefiles radio button.
3. Use the **Browse** button at the top of the window to locate the root directory to use for reverse engineering. The file list area will then list any makefiles located in this directory (based on the value of the `[lang]_ReverseEngineering:Main:MakefileExtension` property).

4. Mark the check box next to the filename of the makefile in the file list.
5. Use the Environment list to choose the correct environment.
6. Click the Analyze Makefile button.

After the analysis is complete, the source files referenced in the makefile will be displayed in the Select Files section of the window. If you go to the Reverse Engineering Advanced Options window, you will see that Rational Rhapsody has also brought in the other relevant settings from the makefile.

When using this feature, there a number of points that should be kept in mind:

- ◆ While the makefile analysis feature allows you to bring in the settings from the makefile and review them in the Reverse Engineering window (and its Advanced Options window), you can also use the settings from the makefile for reverse engineering without performing the analysis step: simply select the appropriate makefile, and click the Start button to begin reverse engineering.
- ◆ When analyzing makefiles, Rational Rhapsody does not actually parse the makefile. Rather, it opens the makefile and then analyzes the output. Therefore, in order for this feature to work properly, you have to make sure that the value of the property `InvokeMake` is correct.
- ◆ Since the syntax of makefiles varies between environments, Rational Rhapsody uses a set of environment-level properties in order to analyze makefiles. These properties can be found under `[lang]_ReverseEngineering::Makefile[environment name]`, and they are used to define the syntax for standard makefile commands. The values of these properties can be customized for different environments. This set of properties includes:
 - `MakeCommand`
 - `IncludeSwitch`
 - `DefineSwitch`
 - `UndefineSwitch`
 - `CompileNoLinkSwitch`
 - `CompileCommands`
 - `LinkCommands`
 - `ChangeDirectoryCommand`
 - `ChangeDirectorySwitch`

Visualization of external elements

Rational Rhapsody enables you to visualize legacy code or edit external code as *external elements*. This external code is code that is developed and maintained outside of Rational Rhapsody. This code will not be regenerated by Rational Rhapsody, but will participate in code generation of Rational Rhapsody models that interact or interface with this external code so, for example, the appropriate `#include` statement is generated. This functionality provides easy modeling with code written outside of Rational Rhapsody, and a better understanding of a proven system.

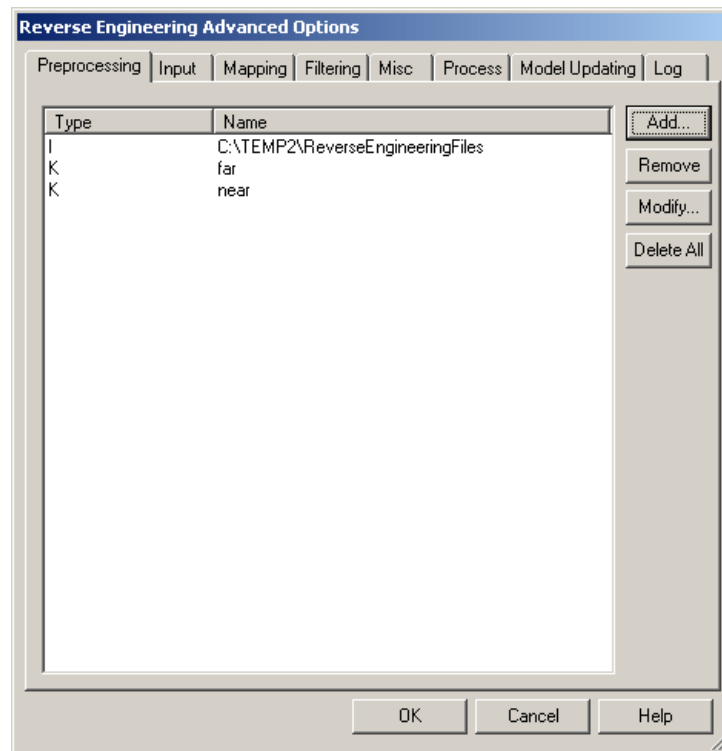
Rational Rhapsody supports the following reverse engineering functionality using external elements:

- ◆ Reverse engineering can import elements as external.
- ◆ Reverse engineering populates the model with enough information to:
 - Model external elements in the model.
 - Enable you to open the source of the external elements, even if the element is not included in the scope of the active component.

For more information, see [External elements](#).

Defining preprocessor symbols

The **Preprocessing** tab, as shown in the following figure, lets you define preprocessor symbols to be uniformly applied to all imported files.



You can define the following types of preprocessor symbols:

- ♦ **D** means symbol defined with `#define`. See [Defined symbols \(C and C++\)](#).
- ♦ **I** means directory added to `#include` search path. See [Include/CLASSPATH paths](#).
- ♦ **K** means additional keywords. See [Additional keywords \(C and C++\)](#).
- ♦ **U** means symbol undefined with `#undef`. See [Undefined symbols \(C and C++\)](#).

The **Preprocessing** tab has the following controls:

- ♦ **Add** button lets you add a preprocessing symbol. See [Adding a preprocessing symbol](#).
- ♦ **Remove** button, which is available only when applicable, lets you remove one or more selected symbols from the list.
- ♦ **Modify** button lets you modify the selected symbol.
- ♦ **Delete All** button lets you delete all preprocessing symbols from the list.

Adding a preprocessing symbol

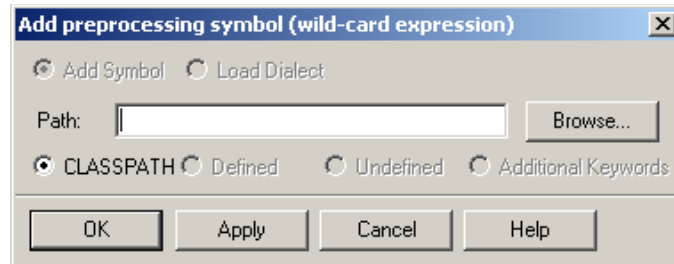
To add a preprocessing symbol, click **Add** on the **Preprocessing** tab of the Reverse Engineering Options window to open the Add Preprocessing Symbol window.

In C and C++, you can add the following symbols:

- ♦ Include symbols. See [Include/CLASSPATH paths](#).
- ♦ Defined symbols. See [Defined symbols \(C and C++\)](#).
- ♦ Undefined symbols. See [Undefined symbols \(C and C++\)](#).
- ♦ Additional keywords. See [Additional keywords \(C and C++\)](#).

In addition, in C++ you can load dialects. See [Dialects \(C++\)](#).

For Rational Rhapsody in Java, the Add Preprocessing Symbol window enables you to specify only the **CLASSPATH** option, as shown in the following figure.



Include/CLASSPATH paths

Include paths tell Rational Rhapsody where to look for header files to be included using `#include` directives found in the source file. This directive has the same effect as the following preprocessing switch:

```
/I<dir>
```

The reverse engineering tool adds constructs declared in header files with `#include` statements to the model, as long as either condition is true:

- ♦ The file is in the same directory as the one being imported.
- ♦ An include path is specified on the **Preprocessing** tab.

To include the path:

1. In the Add Preprocessing Symbol window, select the **Add Symbol** and **Include/CLASSPATH** radio buttons.
2. Use the **Browse** button to locate the appropriate folder.
3. Depending on what you want to do:
 - ♦ Click **Apply** if you want to enter another Include path.
 - ♦ Click **OK** if you are done and to return to the **Preprocessing** tab. Notice that a symbol with type I is added to the list of preprocessing symbols.
4. Click **OK** again to close the Reverse Engineering Options window.
5. Click **Start** to re-import the file.

Defined symbols (C and C++)

The `#define` symbol is a constant and macro that the preprocessor expands before compilation. No storage is allocated for these symbols. They have no type, and the debugger cannot reference them. This definition has the same effect as the following preprocessing switch:

```
/D<name>{=|#}<text>
```

- ◆ Solving parser problems with unknown macros or statements
- ◆ For `#ifdef` inclusion or exclusion of code parts

To define symbols:

1. On the Add Preprocessing Symbol window, select the **Add Symbol** and **Defined** radio buttons.
2. In the **Symbol** box, type the name of the symbol and the value, if it has one, using the following format:

```
<symbol> = <value>
```

For example, to define a preprocessing symbol `ev_H` with the value

```
"$Id: event.h 1.22 1999/02/03 11:12:36 amy Exp $", type:
```

```
ev_H = "$Id: event.h 1.22 1999/02/03 11:12:36 amy Exp $"
```

3. Depending on what you want to do:
 - ◆ Click **Apply** if you want to enter another `#define` symbol.
 - ◆ Click **OK** if you are done and to return to the **Preprocessing** tab. Notice that a symbol with type D is added to the list of preprocessing symbols.
4. Click **OK**.

Using #define

Rational Rhapsody supports #define preprocessor directives that use the following format:

```
#define <identifier> <replacement list>
```

Typically, you declare a #define declarative in a C model is as follows:

- ◆ If it contains parameters, model it as a function with the `C.CG::Operation::Header` property set to `in_header`. For example:

```
#define MAX(X,Y)  
(X)>(Y)?(X):(Y)
```

- ◆ If it does not contain parameters (for example, it defines a constant), model it as a constant variable. For example:

```
#define SIZE 1024
```

- ◆ Otherwise, model it as a type (for example, multi-line #defines).

Typically, to declare a #define declarative in a C++ model is as follows:

- ◆ If the #define declarative in C++ does not contain parameters (for example, it defines a constant), model it as a constant variable and set the `CPP.CG::Attribute::ConstantVariableAsDefine` property to `Checked`. For example:

```
#define SIZE 1024
```

- ◆ Otherwise, model it as a type.

The reverse engineering tool imports the #defines according to the way they are modeled. However, if the comment for the #define is a multi-line, even though the #define itself is one line, the reverse engineering tool imports it as a type. For example:

```
#define SIZE 1024 /* my buffer  
size */
```

To import all #define as a type, set the `<lang>_ReverseEngineering::ImplementationTrait::ImportDefineAsType` property to `True`. See the definition provided for the property on the applicable **Properties** tab of the Features window.

Using #if...#ifdef...#else...#endif

The reverse engineering tool reacts to preprocessor conditions (#if...#ifdef...#else...#endif) just as a compiler does. The preprocessor condition structure is not read by the reverse engineering parser, and the only data received is the data inside valid preprocessor conditions.

Consider the following code in a source file:

```
#ifdef _STDC
#define _A
#else
#define _B
#endif
```

- ◆ If `_STDC` is known by the preprocessor, the result of importing this file will be the creation of a user type named `_A` with the following declaration:

```
#define %s
```

- ◆ If `_STDC` is **not** known by the preprocessor, the result of importing this file will be the creation of a user type named `_B` with the following declaration:

```
#define %s
```

Undefined symbols (C and C++)

The `#undef` preprocessing directive undefines symbols previously defined using the `#define` directive. This has the same effect as the following preprocessing switch:

```
/U<name>
```

To undefine a symbol with `#undef`:

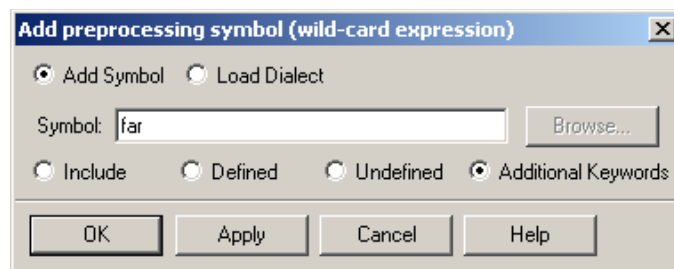
1. In the Add Preprocessing Symbol window, select the **Add Symbol** and **Undefined** radio buttons.
2. In the **Symbol** box, type the name of the symbol you want to undefine.
3. Depending on what you want to do:
 - ◆ Click **Apply** if you want to enter another symbol with `#undef`.
 - ◆ Click **OK** if you are done and to return to the **Preprocessing** tab.
4. Click **OK**.

Additional keywords (C and C++)

To improve keyword support for reverse engineering and roundtripping so that Rational Rhapsody can correctly import and roundtrip declarations that use non-standard or unknown keywords, you can add a list of additional user-defined keywords to the **Preprocessing** tab of the Reverse Engineering Options window.

To add additional keywords:

1. On the Add Preprocessing Symbol window, select the **Add Symbol** and **Additional Keywords** radio buttons.
2. Enter a keyword in the **Symbol** box, as shown in the following figure.



3. Depending on what you want to do:
 - ♦ Click **Apply** if you want to enter more additional keywords.
 - ♦ Click **OK** if you are done and to return to the **Preprocessing** tab.
4. Notice on the **Preprocessing** tab that your keywords with type K are added to the list of preprocessing items.

Note

You can use the `<lang>_ReverseEngineering:Parser:AdditionalKeywords` property to add a list of comma-delimited additional keywords (for example: `far, near`). Note that this property might already have keywords included in it that is provided with Rational Rhapsody.

Additional keywords limitations

Note the following limitations for additional keywords:

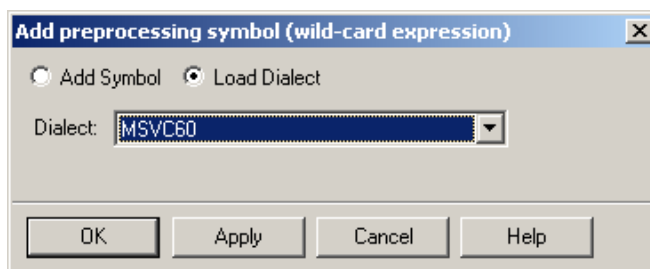
- ◆ Keywords with parameters are not supported.
- ◆ Keywords with more than one word in them are not supported.
- ◆ Keywords cannot be seen in the signature for the element.
- ◆ If the same keyword is used in more than one place (for example, before the type and after the type), the parser will encounter an ambiguity and will fail to indicate the keyword use correctly.

Dialects (C++)

Imported source files must contain C++ code that complies with the ANSI/ISO C++ standard. In practice, there are many dialects of C++ in use, some reflecting various stages in the evolution of the language. The dialect setting determines the default dialect. In addition, it can add a list of dialect-specific preprocessing switches to the user-defined switches. The Reverse Engineering tool understands the Microsoft Visual C++ 6.0 (MSVC60) dialect.

To select this dialect in Rational Rhapsody Developer for C++:

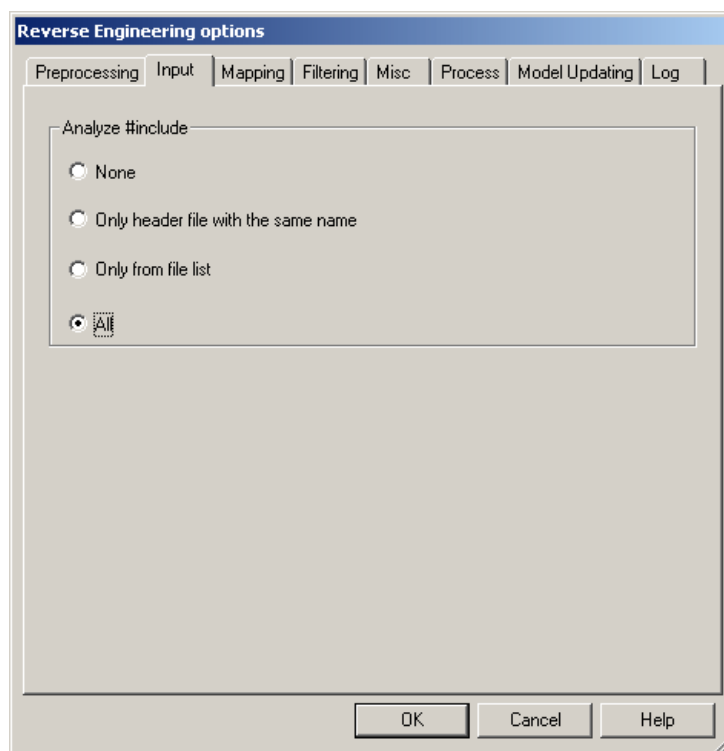
1. In the Add Preprocessing Symbol window, select the **Load Dialect** radio button.
2. From the **Dialect** list, select **MSVC60**, as shown in the following figure.



3. Click **OK**.

Analyzing #include files

The **Input** tab, shown in the following figure, lets you specify which include files should be analyzed in the reverse engineering process.



- ◆ **None** means to analyze only the files or folders specified in the main Reverse Engineering window; all #include statements are ignored.

This mode contributes the least performance drain to the reverse engineering process. Reverse engineering in this most limited mode should only be used when appropriate. This mode imports no implementation information (such as operation bodies) or initialization of static variables, and loses needed information such as dependencies. Keep in mind that implementation files cannot be analyzed without first analyzing their corresponding specification files.

Note: Using **None** will not import the content of .cpp or .c files.

- ◆ **Only header file with the same name** means to analyze only the matched included files. In other words, the corresponding specification file for the analyzed implementation file of the same name. This is known as *logical files mode*.

For example, for the implementation file named `MyClass.cpp`, the reverse engineering

utility will analyze only the include file named `MyClass.h`. All other included files in the list of files you selected are not analyzed. This high-performance mode imports full information about analyzed classes, but might lose dependencies through separated parts of a project. This mode is designed for reverse engineering of large projects (about 1000 files).

Note: If you select this option, nested `#include` statements are not analyzed.

See [Analyzing header files with the same name](#) for an example.

- ♦ **Only from file list** means to analyze only the include files you specified in the main Reverse Engineering window.

For example, suppose you have four files (`one.h`, `two.h`, `three.h` and `one.cpp`) and you select the files named `one.h`, `two.h`, and `one.cpp`. The reverse engineering utility will analyze `one.cpp` and its include files `one.h` and `two.h`. It **will not** analyze `three.h` because you did not select it.

This mode gives you the most control and strategic conservation of performance, eliminating the needless analysis of irrelevant files and redundant information. In addition, it enables you to select files containing important declarations to the analysis without having to add every file within the directory. This mode of reverse engineering imports all the information needed and creates dependencies through whole project. It is designed for middle-sized projects (about 100 files).

See [Analyzing a list of files](#) for an example.

- ♦ **All** means to analyze all included files on all levels. This mode is called *recursive analysis*, and consumes the most performance because it imports all information, even redundant information such as MFC and STL. This is the default value.

The `<lang>_ReverseEngineering::ImplementationTrait::AnalyzeIncludeFiles` property has enumerated values to indicate how the reverse engineering process analyzes include files. See the definition provided for the property on the applicable **Properties** tab of the Features window.

For C++ projects, you can use the `CreateDependencies` property to specify how the reverse engineering feature should handle the creation of dependency elements in the model from code constructs such as `#includes`, forward declarations, friends, and namespace usage. For more details on how to use this property, see the definition provided for the property on the applicable **Properties** tab of the Features window.

Analyzing header files with the same name

Suppose you want to reverse engineer the file `omreactive.cpp`. It has the following included files:

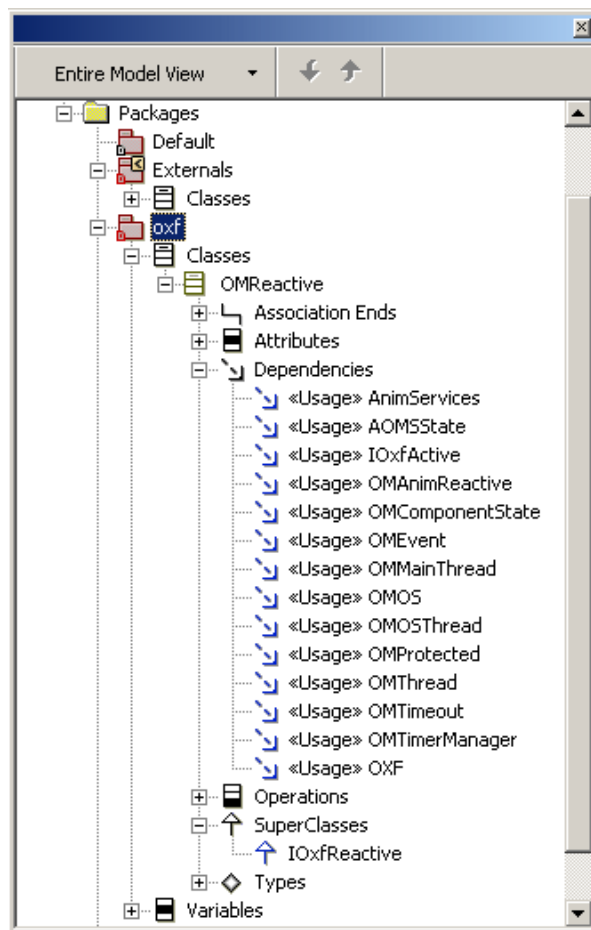
```
#include <oxf.h>
#include <omoutput.h>
#include <omreactive.h>
#include <state.h>
#include <omthread.h>
#include <aommacro.h>
```

To reverse engineer only `omreactive.cpp` and `omreactive.h`:

1. Add the file `omreactive.cpp` to the main Reverse Engineering window.
2. Click **Advanced** to open the Reverse Engineering Options window.
3. On the **Input** tab, select the **Only header file with the same name** radio button.
4. On the **Preprocessing** tab, add the path to the `oxf` folder (for example, `<Rational Rhapsody installation path>\Share\LangCpp`). You need to set this value because the directive `#include <omreactive.h>` says to look for the specification file in `omreactive`, so you need to specify where that is.
5. Click **OK**.

6. Click **Start** on the Reverse Engineering window.

The tool will analyze `omreactive.h` and ignore the other files included in the `omreactive.cpp` file. As you can see from the following figure, the Rational Rhapsody browser shows the **oxf** package with the **OMReactive** class and some of its data members. The **OMReactive** class has Usage dependencies to externally referenced classes and inherits from the **IOxfReactive** superclass.



Analyzing a list of files

Suppose you want to reverse engineer the file `omreactive.cpp`. It has the following included files:

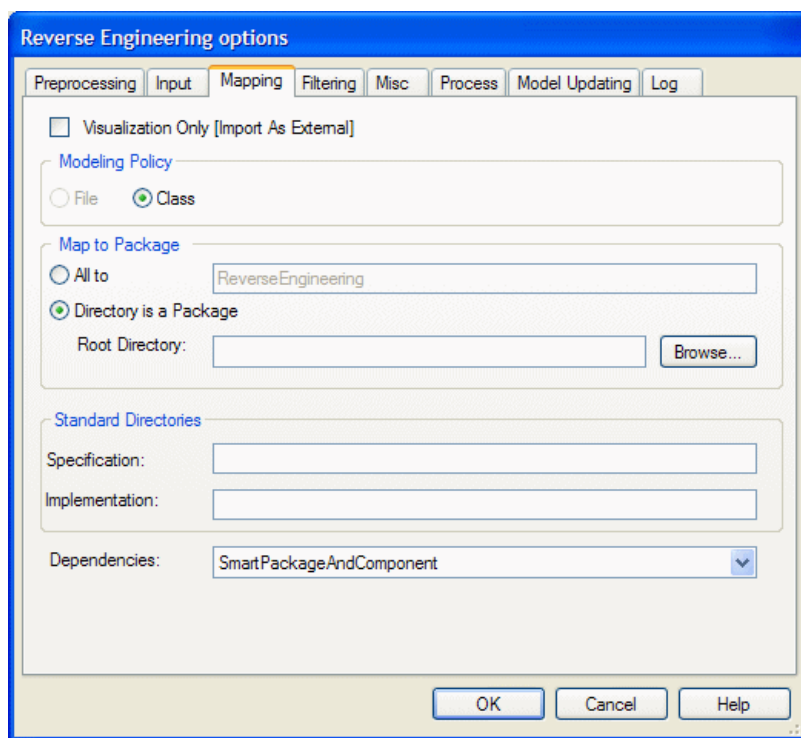
```
#include <oxf.h>
#include <omoutput.h>
#include <omreactive.h>
#include <state.h>
#include <omthread.h>
#include <aommacro.h>
```

If you specified all of these files except for `state.h`, all the include files except for `state.h` will be analyzed by the reverse engineering tool.

Mapping classes to types and packages

The **Mapping** tab, as shown in the following figure, enables you to:

- ♦ Create external files in a given package or component.
- ♦ Set the modeling policy based on a file or a class.
- ♦ Allow files to be imported into one package, or to import the files while emulating the existing directory structure.
- ♦ Set standard directories.
- ♦ Create UML dependencies from include statements.



The **Mapping** tab has the following controls:

- ♦ **Visualization Only (Import as External)** check box, when selected, sets Rational Rhapsody to add as external elements all the elements created in Rational Rhapsody during reverse engineering (their `CG::Class/Package/Type::UseAsExternal` property is set to `Checked`). This property is overridden for all the packages (but not their contained elements, such as classes, files, types, unless the contained elements are also packages; and in that case they will also have their `CG::Package::UseAsExternal` property set to `Checked`). In C++, this control has an effect on the controls available in the **Modeling Policy** group.

Note: When you select or clear the **Visualization Only (Import as External)** check box, this same check box is automatically set the same way on the Reverse Engineering window.

- ♦ **Modeling Policy** group lets you save the original file mapping after reverse engineering. This group has the following controls:

- **File** radio button lets you save files to be imported into separate packages. This is the default value for Rational Rhapsody Developer for C. For C, this means that a model is a file-based (Functional C) model, which is a code-centric model.

This option is only available in C++ when importing the package as “external” (select the **Visualization Only (Import as External)** check box). Importing C++ external files as packages is not advisable if the imported code needs to be reused because code generation for package files is not supported.

This option is unavailable in Java.

- **Class** radio button means that no files are created.

Note: If you want the mapping file saved to a component (so that you can import files as class-based information and replicates the file structure), set the `<lang>_ReverseEngineering::ImplementationTrait::RespectCodeLayout` property to `Mapping`.

The root path of the top-most folder created during reverse engineering is added to the **Include Path** field of the active component; it is later used to compile `#include` statements for the imported files and to open the source code for external elements. If the imported files have several, non-dependent roots, multiple paths are written to the field, separated by commas.

For example, for files `C:\Project1\Subsystem\A.h` and `D:\Project2\Subsystem2\B.h`, the **Include Path** field would contain the following string:

`C:\Project1;D:\Project2`

If this option is used with an empty component (one whose scope is empty or includes only external elements), the first root path is added to the **Directory** field of the component; any additional paths are written to the **Include Path** field.

- ♦ **Map to Package** group lets you specify how to organize imported elements to packages. This group has the following controls:
 - **All to** radio button lets you map all imported elements to the package you specify in the text box next to this radio button. For example, when you select the **All to** radio button, the Reverse Engineering tool automatically fills in the default name of ReverseEngineering in the text box. This means that all imported elements will be mapped to a package called **ReverseEngineering**. You can change this value, for example, to MyREPackage.

Note: It is possible to set a nested package in this field. The syntax is Package1::Package2. Non-existent packages will be created during reverse engineering.

- **Directory is a Package** radio button lets you reverse engineer the files while emulating the existing directory structure so that packages are created for each imported directory. All the files in the directory are imported to the corresponding package. This is the default value. You can set the root directory or have the Reverse Engineering tool calculate it for you. See [Specifying directory structures](#) for more details and an example.

Note: This option sets the CG::Package::GeneratedDirectory property to Checked for the active configuration. For elements, the original hierarchy of directories is restored on code generation for the imported elements, and #include statements are generated for references to them.

- ♦ **Standard Directories** group lets you separate the header and source files to different directories for the reverse engineering of your selected project. This group has the following controls:
 - **Specification** box. Enter the name of the directory for your header files (for example, inc).
 - **Implementation** box. Enter the name of the directory for your source files (for example, src).

Using the **Specification** and **Implementation** boxes provides you with better modeling of your code if you want to separate the header and source files to different directories. Code generation (after reverse engineering) will also generate each header file to a directory called `inc` and each source file to a directory called `src` (or whatever names you designed in the **Specification** and **Implementation** boxes). The following table illustrates if you use these boxes (left column) versus if you do not (right column).

Specification = inc Implementation = src	Specification = [blank] Implementation = [blank]
client "a elements (from a.h and a.c)" "b elements (from b.h and b.c)" "c elements (from c.h and c.c)" server "e elements (from e.h and e.c)" "f elements (from f.h and f.c)" "g elements (from g.h and g.c)"	client inc "a elements from a.h" "b elements from b.h" "c elements from c.h" src "a elements from a.c" "b elements from b.c" "c elements from c.c" server inc "e elements from e.h" "f elements from f.h" "g elements from g.h" src "e elements from e.c" "f elements from f.c" "g elements from g.c"

Note: You cannot view code using the Active Code View window nor can you edit code if you generate to an external directory.

- ♦ **Dependencies** list lets you set Rational Rhapsody to create dependencies from the include files during reverse engineering. This means that `#include` between files can create dependencies between the component files and/or between classes. In addition, forward declarations of elements (variables, functions, classes, and so on) will create dependencies from the component file to the element. The possible values are:
 - **ComponentOnly** means to create dependencies between component files, but not between model classes
 - **None** means do not create dependencies on reverse engineering
 - **PackageAndComponent** means to create dependencies between model classes and component files
 - **PackageOnly** means to create dependencies between model classes, but not between component files
 - **SmartPackageAndComponent** means to create only necessary dependencies to reflect the code

Note: The above values for **Dependencies** are all that are available in the current Rational Rhapsody version. Note the following information:

Not all these values will appear for every language of Rational Rhapsody. For example, all of the above values might be available for Rational Rhapsody in C++, but only two might be available for Rational Rhapsody in Java.

You can set the default for **Dependencies** in the `<lang>_ReverseEngineering::ImplementationTrait::CreateDependencies` property. You might notice in the property definition that there is a `DependenciesOnly` value, but it does not appear in the **Dependencies** list. This value is only for backward compatibility purposes and you cannot set this value directly. In a case where you might have an old model that uses `DependenciesOnly`, the current Rational Rhapsody automatically sets the value to `PackageOnly`.

The list of values and backward compatibility behavior are different among the languages.

This operation is successful if the reverse engineering utility analyzes both the included file and the source; and the source and included files contain class declarations for creating the dependencies between them. If there is not enough information, the includes are not converted into dependencies. This can happen in the following cases:

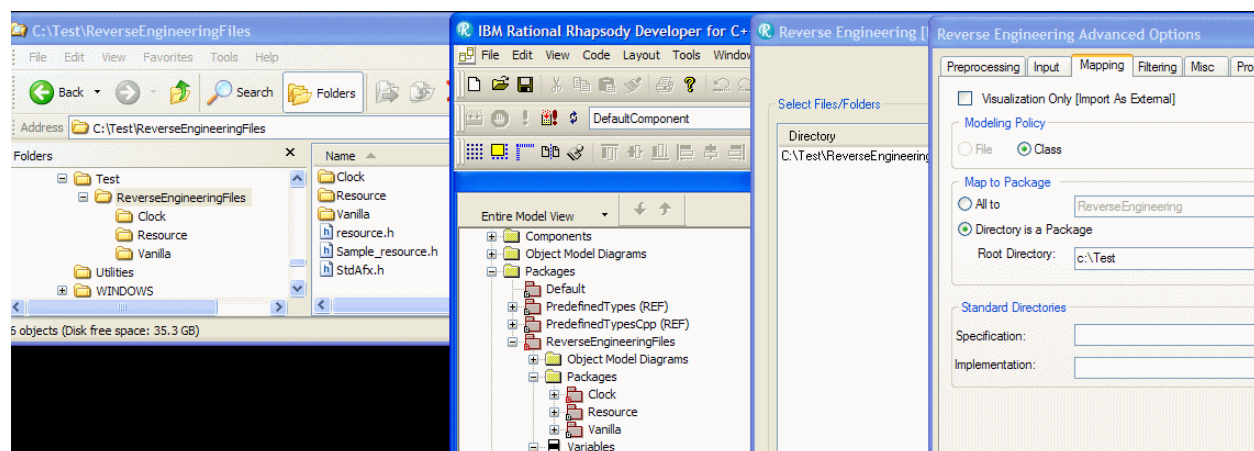
- The include file was not found, or is not in the scope of the settings on the **Input** tab.
- A class is not defined in the include file or source file, so the dependency could not be created.

If the dependency is not created successfully, the include files that were not converted to dependencies are imported to the `<lang>_CG::Class::SpecIncludes` or `<lang>_CG::Class::ImpIncludes` properties so you do not have to re-create them manually. If the include file is in the specification file, the information is imported to the `SpecIncludes` property; if it is in the implementation file, the information is imported to the `ImpIncludes` property.

If a file contains several classes, include information is imported for all the classes in the file.

Specifying directory structures

The **Directory is a Package** radio button lets you import files into one package, or to import the files while emulating the existing directory structure. When you select the **Directory is a Package** radio button, the file hierarchy is preserved during import, and the tool creates nested packages from the folders (if any), as shown in the following figure where the left-most image shows the path of the source files, the middle image of the Rational Rhapsody browser shows the results of the reverse engineering, and the right-most image shows the **Mapping** tab settings for the reverse engineering.



You can use the `<lang>_ReverseEngineering::ImplementationTrait::RootDirectory` property to designate the root directory that contains all the folders that should become package. In this way, Rational Rhapsody builds the package hierarchy according to the folder tree from the specified path.

With the `RootDirectory` property set, the root directory path shows on the **Mapping** tab when you access it. You can change the path for the particular reverse engineering session you are in by changing the value in the **Root Directory** box on the **Mapping** tab.

The `RootDirectory` property is necessary for code centric modeling as it determines the component directory and supports the package-to-directory connection.

When applicable, the reverse engineering tool calculates the directory that creates the minimum number of packages and suggests (through a message window) this directory as the root directory during the reverse engineering process. Use the `<lang>_ReverseEngineering::ImplementationTrait::UseCalculatedRootDirectory` property to set whether the root directory should be calculated, that the calculated root directory should always override the value in the `RootDirectory` property, or that you should be automatically asked if you want the calculated root directory to override the value in that property.

Note

The root directory feature is available for Rational Rhapsody in C, C++, and Java.

Note the following information:

- ♦ If you are doing reverse engineering with merge option, there will be no calculated root directory and the current value will be used.
- ♦ For Rational Rhapsody in Java, if there is no value in the `RootDirectory` property and you are using the tree view version of the Reverse Engineering window (click the Tree View button ) , the root directory from the tree view will be used.
- ♦ When you export an Eclipse project to Rational Rhapsody, the root directory is set according to the Eclipse project. The `UseCalculatedRootDirectory` property is ignored.
- ♦ If you use batch or the COM API to do reverse engineering, a No Reply is used if the `UseCalculatedRootDirectory` property is set to Auto. To avoid the calculate root directory message box, you might want to set the `UseCalculatedRootDirectory` property to Always or Never.

C++ namespaces

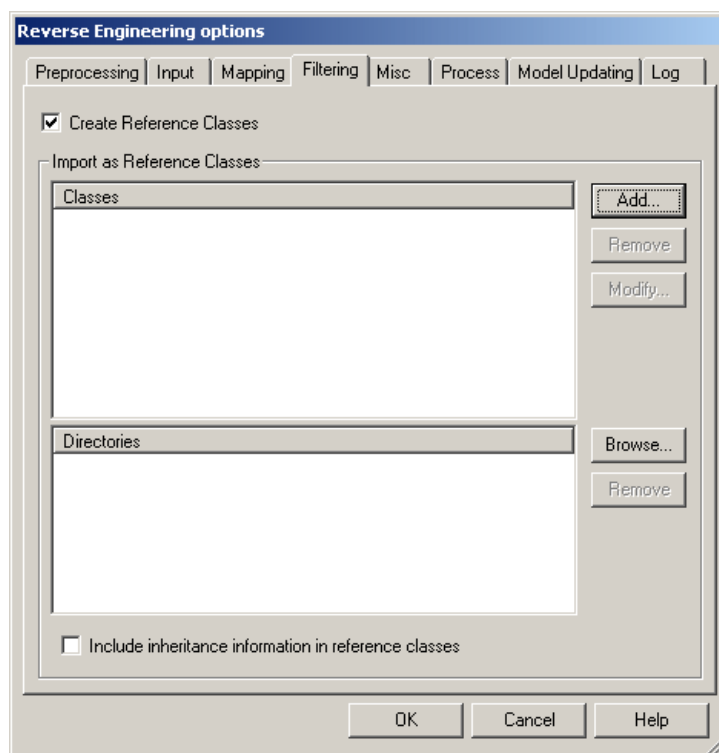
The reverse engineering tool fully supports C++ namespaces. Namespaces are always converted to packages.

For example, the following code will be imported correctly:

```
namespace XXX {  
    class CCC {  
        int i;  
    };  
}
```

Specifying reference classes

The **Filtering** tab, shown in the following figure, enables you to specify classes that should be imported as reference classes (the equivalent of `CG::Class::UseAsExternal` set to `Checked`). You can select individual classes to model as reference classes, or specify an entire directory of reference classes. Reference classes are imported without attributes and operations; they are used for referencing only.



The tab contains the following controls:

- ♦ **Create Reference Classes** check box lets you specify whether to create external classes for undefined classes that result from forward declarations and inheritance. By default, reference classes are created (as in previous versions of Rational Rhapsody).

If the incomplete class cannot be resolved, the tool deletes the incomplete class if the `<lang>_ReverseEngineering::Filtering::CreateReferenceClasses` property is set to `Cleared`.

Note: In some cases, the class cannot be deleted (for example, a class referenced by a typedef type).

- ♦ **Import as Reference Classes** group has the following controls:
 - **Classes** box lists the classes that should be imported as reference classes and their directories. The following buttons control this list:
 - Add** lets you add a reference class.
 - Remove** lets you remove one or more selected reference classes from the list.
 - Modify** lets you modify the specified reference class.
 - **Directories** box lists the directories for the imported classes. The following buttons control this list:
 - Browse** button lets you to browse to the correct directory.
 - Remove** lets you remove one or more selected directories from the list.

To analyze these elements during reverse engineering, set the following properties (under `<lang>_ReverseEngineering::Filtering`) to `Checked`:

- ♦ `AnalyzeGlobalFunctions`
- ♦ `AnalyzeGlobalVariables`
- ♦ `AnalyzeGlobalTypes`

See the definition provided for a property on the applicable **Properties** tab of the Features window.

Reference classes

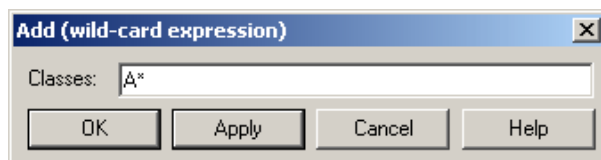
Reference classes are imported into the model as placeholders without members or relations. A typical example is the MFC classes, which are not of interest for elaboration but can be listed simply to show that they are acting as superclasses or peer classes to other classes in the model. Wildcard expressions are permissible for reference class names.

Adding a reference class

To add a reference class:

1. In the **Filtering** tab on the Reverse Engineering Options window, click **Add** to open the Add window.
2. Type a wildcard expression in the **Classes** box to match the reference class names, as shown in the following figure. For example, to match all class names that begin with the letter A, enter the following wildcard expression:

A*



3. Click **OK**.

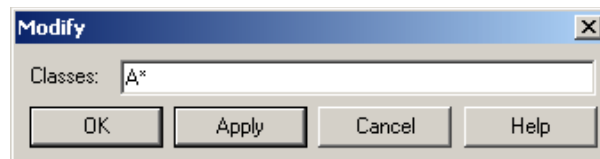
Deleting a reference class

To remove a reference class, select the class in the **Classes** list and click **Remove**.

Modifying a reference class

To modify a reference class:

1. In the **Classes** list, select the reference class and click **Modify**. The Modify window opens, as shown in the following figure.



2. Modify the wildcard expression.
3. Click **OK**.

Locating a directory that contains reference classes

The **Directories** list under the **Classes** list on the **Filtering** tab specifies the directories that the reverse engineering tool should search for reference classes, including their subdirectories.

To locate a directory that contains reference classes:

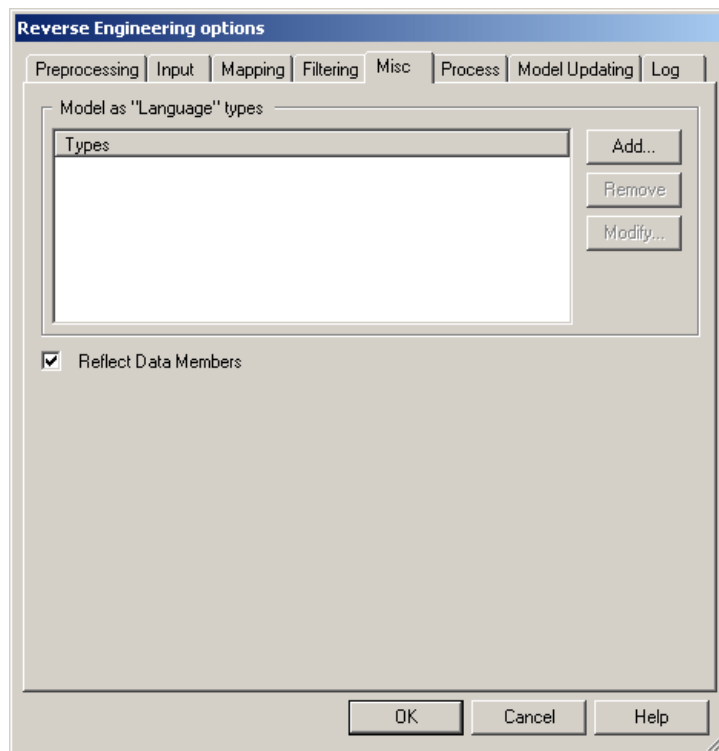
1. On the **Filtering** tab, click **Browse** to open the Browse for Folder window.
2. Select the appropriate directory and click **OK**.

To remove a directory, select the reference class directory and click **Remove**.

Miscellaneous reverse engineering options

The **Misc** tab, shown in the following figure, enables you to:

- ♦ Import specific classes as Rational Rhapsody types
- ♦ Reflect data members



The **Misc** tab has the following controls:

- ♦ **Model as “Language” types** box lets you specify which classes should be modeled as types. This box has the following controls:
 - **Add** button lets you add a class to be modeled as a type.
 - **Remove** button, which is available when applicable, lets you remove the selected type from the list.
 - **Modify** button, which is available when applicable, lets you modify the selected type.

See [Modeling classes as Rational Rhapsody types](#).

- ◆ **Reflect Data Members** means if this check box is not selected (the check box is cleared), the access level of data members is imported to the `Visibility` property for attributes and the `DataMemberVisibility` property for relations. The visibility in the Features window is always **Public**. This is the behavior of previous versions of Rational Rhapsody.

If this check box is selected, the access level of data members is displayed in the Features window. The `Visibility` property is always set to `fromAttribute` (as `VisibilityOnly`). For relations, the access level is imported to the `DataMemberVisibility` property. In addition, generation of helper functions is disabled on the class properties level.

By default, this checked box is selected.

The following table lists the property values that will be set if the **Reflect Data Members** check box is selected.

Subject and Metaclass	Property	Value
For attributes		
<lang>_CG::Attribute	AccessorGenerate	Cleared
	MutatorGenerate	Never
For relations		
<lang>_CG::Relation	GetAtGenerate	Cleared
	GetKeyGenerate	Cleared
	RemoveKeyGenerate	Cleared
CG::Relation	AddComponentHelpersGenerate	Cleared
	AddGenerate	Cleared
	AddHelpersGenerate	False
	ClearGenerate	Cleared
	ClearHelpersGenerate	Cleared
	CreateComponentGenerate	Cleared
	DeleteComponentGenerate	Cleared
	FindGenerate	Cleared
	GetEndGenerate	Cleared
	GetGenerate	Cleared
	RemoveComponentHelpersGenerate	Cleared
	RemoveGenerate	Cleared
	RemoveHelpersGenerate	False
	SetComponentHelpersGenerate	Cleared
	SetGenerate	Cleared
	SetHelpersGenerate	From Modifier
For classes		
CG::Class	InitCleanUpRelations	Cleared

Note: The setting of this check box is equivalent to the <lang>_ReverseEngineering::ImplementationTrait::ReflectDataMembers property. See the definition provided for the property on the applicable **Properties** tab of the Features window.

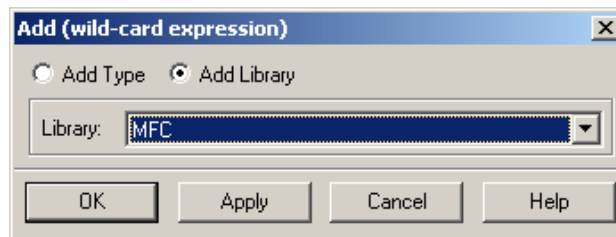
See [Reflect data members](#) for an example.

Modeling classes as Rational Rhapsody types

Rational Rhapsody can either model certain classes as types or use the MFC type library.

1. On the **Misc** tab of the Reverse Engineering Advanced Options window, click **Add** to open the Add (wild card expression) window for types.
2. Select the appropriate radio button:
 - ♦ **Add Type** to add the class as a type
 - ♦ **Add Library** to add the class as an MFC type definition
3. If you selected the **Add Type** radio button, type the name of a single class or a wildcard expression to match the names of several classes to be imported in the **Type** box. For example, `OM*` specifies that all classes that start with “OM” should be types.

Or, if you selected the **Add Library** radio button (applicable to Rational Rhapsody Developer for C++ only), select **MFC** from the **Library** list, as shown in the following figure.



Note: Currently, Rational Rhapsody Developer for C++ recognizes only one predefined type library (MFC) with only one class (CString). You can add more classes in the `CPP_ReverseEngineering::MFC::DataTypes` property of the current configuration. Alternatively, you can create new library metaclasses, like MFC, in the `factory.prp` and `site.prp` files.

Modeling typedefs as user-defined types

Typedefs are read in as user-defined types under the corresponding package or class in the model.

Typedef example 1

Consider a source file that contains the following typedef:

```
typedef unsigned char CHAR;
```

The resultant type will have the name CHAR and have the following form:

```
typedef unsigned char %s
```

Typedef example 2

Consider a source file that contains the following enumerated type:

```
typedef enum {GOOD, INVALID, SWAPPED}image_file_status;
```

The resultant type will have the name image_file_status and have the following form:

```
typedef enum {GOOD, INVALID, SWAPPED} %s
```


Modeling structures as types instead of classes

To have Rational Rhapsody model structures as types instead of classes, click the **Add** button on the **Misc** tab of the Reverse Engineering Options window so that you can enter the names of the structures or use a wildcard to apply the mapping to all structures.

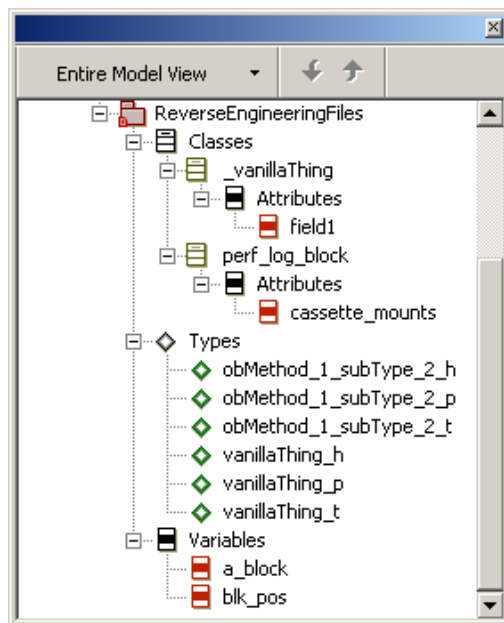
For example, consider the following source file:

```
struct perf_log_block
{
    int cassette_mounts;
};
struct perf_log_block blk_pos[ FIVE ];
another_block a_block[ FIVE ];

typedef struct _vanillaThing
{
    char    field1;
} vanillaThing_t, *vanillaThing_p, **vanillaThing_h;

typedef struct
{
    int     field4;
} obMethod_1_subType_2_t, *obMethod_1_subType_2_p,
  **obMethod_1_subType_2_h;
```

If you do not specify anything on the **Misc** tab, the structures are not modeled as types, as shown in the following figure.

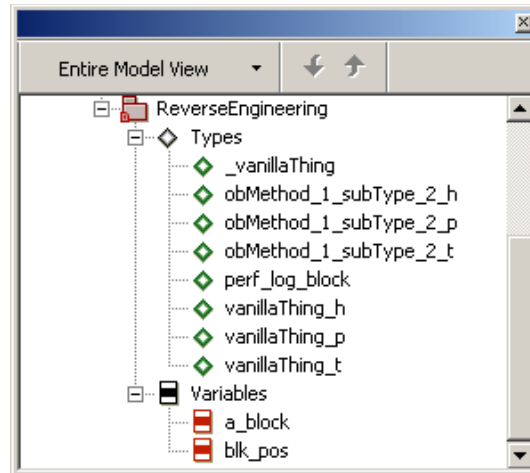


Note the following information:

- ♦ The first structure type is modeled as a class named `perf_log_block` with an attribute named `cassette_mounts`.
- ♦ The array of structures is modeled as an instance of type `perf_log_block` with a multiplicity of FIVE.
- ♦ The array is modeled as a variable named `a_block`, with the following form:

```
another_block %s[ FIVE ]
```
- ♦ The structure typedef is modeled as a class named `_vanillaThing` with an attribute named `field1`.
- ♦ The types `vanillaThing_t`, `*vanillaThing_p`, and `**vanillaThing_h` are modeled as types, as follows:
 - `typedef struct _vanillaThing %s`
 - `typedef struct _vanillaThing * %s`
 - `typedef struct _vanillaThing * * %s`
- ♦ The types `obMethod_1_subType_2_t`, `*obMethod_1_subType_2_p`, `**obMethod_1_subType_2_h` are modeled types.

If you add the wildcard symbol (*) to the **Types** list so all structures are mapped to types, the results are as follows:



Note the following information:

- ♦ The first structure type is modeled as a type named `perf_log_block` with the following declaration:


```
struct %s
{
    int cassette_mounts;
};
```
- ♦ The array of structures is modeled as a variable named `blk_pos`, of type `perf_log_block` with a multiplicity of FIVE, as follows:


```
perf_log_block %s[ FIVE]
```
- ♦ The array is modeled as a variable named `a_block`. Its declaration is as follows:


```
another_block %s[ FIVE ]
```
- ♦ The structure typedef is modeled as a type named `_vanillaThing`. Its declaration is as follows:


```
struct %s
{
    char field1;
};
```
- ♦ The types `vanillaThing_t`, `*vanillaThing_p`, and `**vanillaThing_h` are modeled as types, as follows:


```
- typedef struct _vanillaThing %s
- typedef struct _vanillaThing * %s
- typedef struct _vanillaThing * * %s
```

Reflect data members

When the **Reflect Data Members** check box is selected on the **Misc** tab of the Reverse Engineering Options window, the Reverse Engineering tool imports all code data members as private. The access level of data members in the code is imported into the **Visibility** property of attributes.

When this option is not selected:

- ◆ Reverse engineering imports code data members as attributes with public visibility. All attributes are listed as **Public** in the Features window. In generated code, they have the correct visibility.
- ◆ Accessors and mutators are generated, as are the original, user operations.

When this option is selected:

- ◆ Attributes in the Features window have the “real” visibility, matching the imported code.
- ◆ Accessors and mutators are not generated.

For example, consider the following file, `clock.h`:

```
#ifndef CLOCK_H
#define CLOCK_H

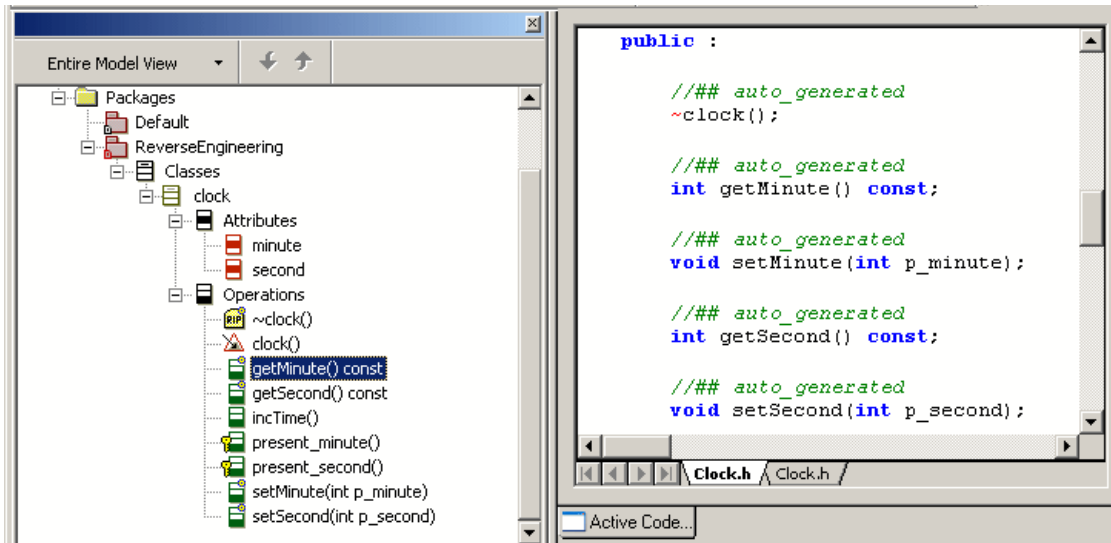
#include <stdio.h>
class clock
{
    int second;
    int minute;

    public:
        clock();
        void incTime(void);
    protected:
        int present_second(void) {return second;}
        int present_minute(void) {return minute;}
};
#endif
```

The file `clock.cpp` contains the following code:

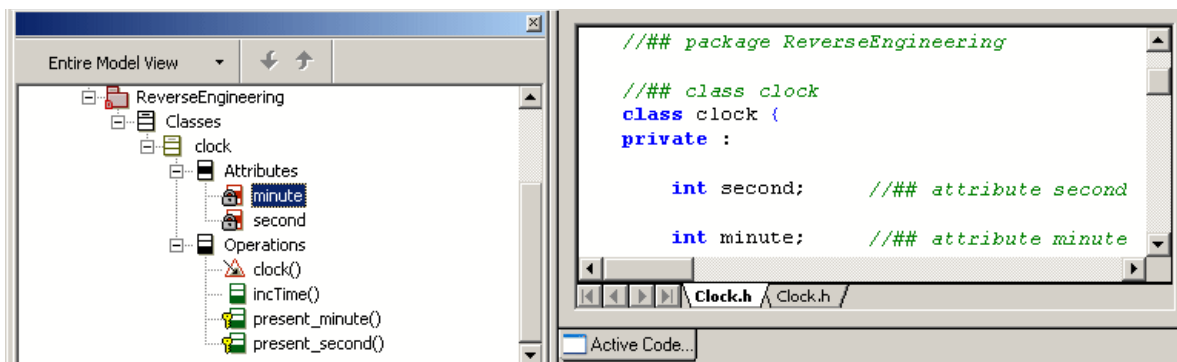
```
clock.cpp
#include "clock.h"
clock::clock() : minute(0),second(0)
{
}
void clock::incTime(void)
{
    if (second == 59)
    {
        second = 0;
        minute ++;
    }
    else
    {
        second++;
    }
    cout << minute << ":" << second << endl;
}
```

If you reverse engineer these files with the **Reflect Data Members** check box cleared (the equivalent of setting the `<lang>ReverseEngineering::ImplementationTrait::ReflectDataMembers` property to `None`) and the input option **Only from file list** on the **Input** tab of the Reverse Engineering Options window, the results are as shown in the following figure.



Note that the accessors and mutators are shown as public in the browser, but the actual visibility of the attributes is private.

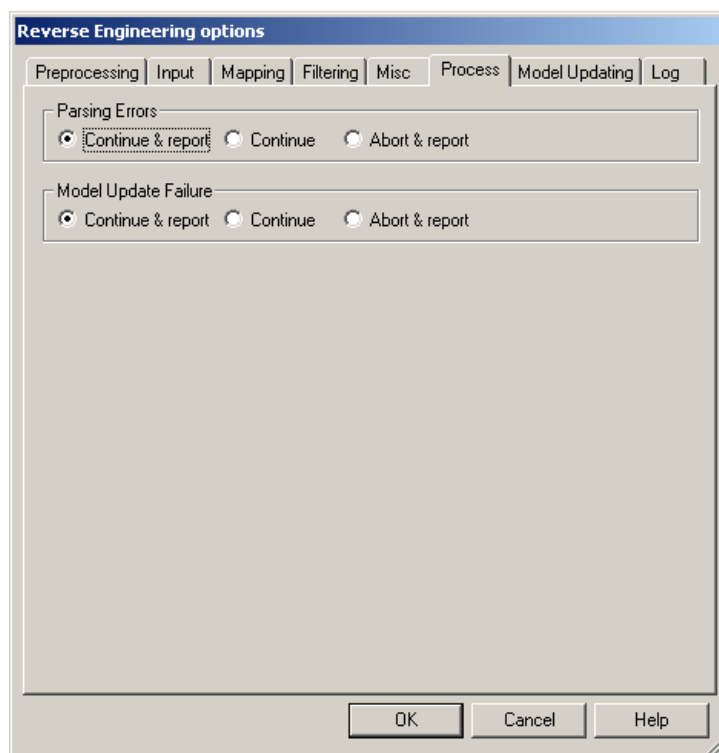
If you select the **Reflect Data Members** check box and repeat the reverse engineering process, the attributes are private and the accessors and mutators are not generated, as shown in the following figure.



In this case, legacy code that already has these operations uses them instead of the Rational Rhapsody default ones.

Reverse engineering error handling

The **Process** tab, shown in the following figure, enables you to specify what to do when the reverse engineering process encounters certain error conditions.



- ♦ **Parsing Errors** are errors encountered while parsing of the source file.
- ♦ **Model Update Failure** are failures that occur when Rational Rhapsody cannot update a model.

For each error condition, there are possible actions:

- ♦ **Continue & report** means to continue importing the next construct and report the error condition.
- ♦ **Continue** means to continue importing but do not report the condition.
- ♦ **Abort & report** means to stop importing and report the condition.

Creating flow charts during reverse engineering

Rational Rhapsody provides an option of automatically creating flow charts for operations during the reverse engineering of code. This feature can be used to create flow charts for all operations in the code or for a subset of operations based on a number of possible criteria.

To have Rational Rhapsody automatically create flow charts during reverse engineering:

1. Open the Reverse Engineering window (choose **Tools > Reverse Engineering**).
2. If you want Rational Rhapsody to create flowcharts as well as other diagrams, select the **Populate Diagrams** check box.
3. If you want Rational Rhapsody to create flow charts but not other diagrams, or you want to tweak the criterion that Rational Rhapsody uses to determine which operations to create flow charts for:
 - a. Click the **Advanced** button.
 - b. When the Reverse Engineering Advanced Options window is displayed, go to the Model Updating tab and select the Create Flowcharts option.
 - c. Use the controls provided with the Create Flowcharts option to specify the criterion for flowchart creation, such as number of control structures or number of lines of code.

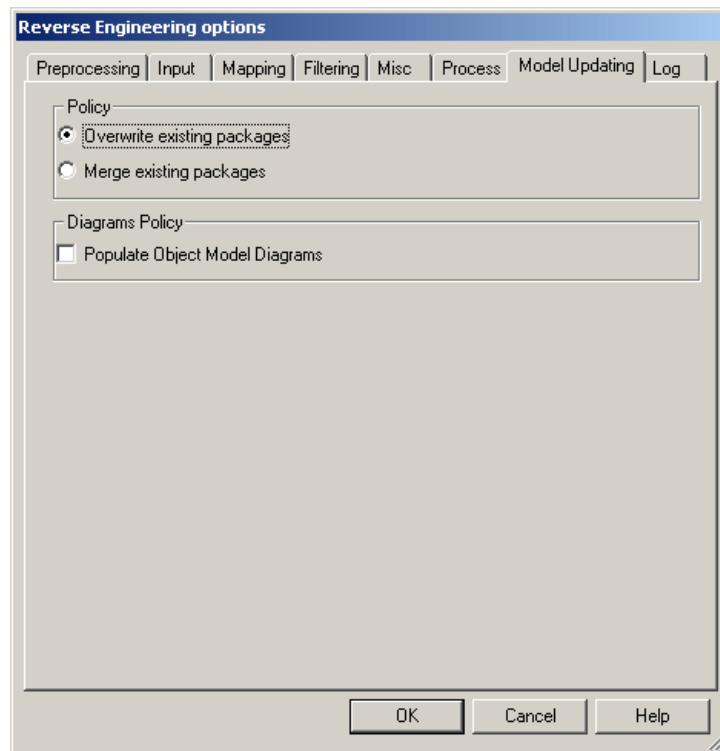
With these options set, when you click the Start button to initiate reverse engineering, flow charts will be created as the code is imported.

The following properties correspond to the flow chart creation option and the criteria that can be used for inclusion:

- ◆ CreateFlowcharts
- ◆ FlowchartCreationCriterion
- ◆ FlowchartMinControlStructures
- ◆ FlowchartMaxControlStructures
- ◆ FlowchartMinLOC
- ◆ FlowchartMaxLOC

Updating existing packages

The **Model Updating** tab, shown in the following figure, enables you to specify how to update existing packages with imported constructs.



The tab contains the following controls:

- ♦ **Overwrite existing packages** means to replace existing packages (including diagrams) with imported ones. For example, if an imported package contains one class and the existing package contains three different classes, the single class being imported replaces the three existing classes.
- ♦ **Merge existing packages** means to merge imported constructs into existing packages. All existing diagrams will be overridden and new diagrams will be created according to the imported packages. For example, if an imported package has a class that contains different attributes than the same class in the existing package, the two classes are merged. The existing package also retains any other classes it had prior to the import.

- ♦ **Populate Object Model Diagrams** means to create object model diagrams (if there are any) when importing. Note that you use **Populate Object Model Diagrams** in conjunction with **Overwrite existing packages** and **Merge existing packages**. Note also that when you select or clear the **Populate Object Model Diagrams** check box, this same check box is automatically set the same way on the Reverse Engineering window.

Note: Your selection is stored in the
ReverseEngineering::Update::CreateDiagramsAfterRE property.

Command-line interface for populate object model diagrams

If you have set the ReverseEngineering::Update::CreateDiagramsAfterRE property to Checked, when you run reverse engineering through the Rational Rhapsody command-line interface, any object model diagrams being imported will be created in your Rational Rhapsody model.

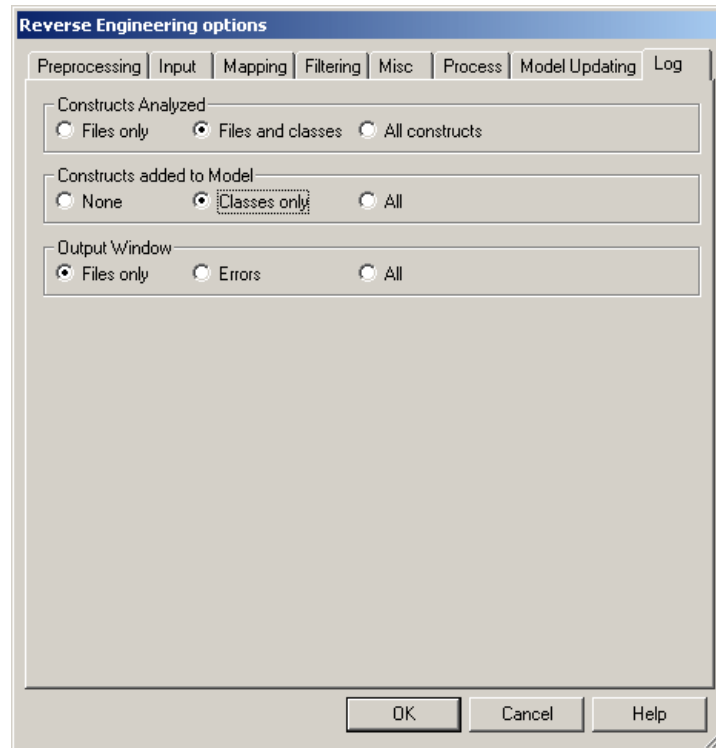
Populate object model diagrams limitations

Note the following limitations:

- ♦ You cannot merge an existing diagram with a new one.
- ♦ Diagrams can include a maximum of 256 elements.

Reverse engineering message reporting

The **Log** tab, shown in the following figure, enables you to specify which kinds of constructs the reverse engineering utility should report on during the import. These options directly impact the performance of the reverse engineering process. The more options you select, the slower the process.



The tab contains the controls:

- ♦ **Constructs Analyzed** specifies which constructs to report. The possible choices are as follows:
 - **Files only** means to report only the files being analyzed.
 - **Files and classes** means to report only the files and classes being analyzed.
 - **All constructs** means to report all constructs being analyzed.
- ♦ **Constructs added to Model** specifies which constructs added to the model are reported. The possible choices are as follows:
 - **None** means no constructs.
 - **Classes only** means to report only the classes being added.

- **All** means to report all the constructs being added.
- ♦ **Output Window** specifies which output to display in the Output window. The possible choices are as follows:
 - **Files only** means to display file information only.
 - **Errors** means to display errors only.
 - **All** means to display all information.

To improve performance when you are reverse engineering large amounts of legacy code, you can hide the Output window. To do this, set the following environment variable in your `rhapsody.ini` file:

```
NO_OUTPUT_WINDOW=TRUE
```

Code respect and reverse engineering for Rational Rhapsody Developer for C and C++

For Rational Rhapsody in C and C++ you can reverse engineer code into the Rational Rhapsody model in a manner that “respects” the structure of the code and preserves this structure when code is regenerated from the Rational Rhapsody model. Meaning that code generated in Rational Rhapsody resembles the original. This means you have complete flexibility for using manually written code or auto-generated code while receiving all the benefits of modeling. You can reverse engineer code into a model in a manner that the model respects the order, location, and dependencies of the global elements in the original code.

For more details about code respect and on how to activate it, see [Code respect](#).

Reverse engineering for C++

You can reverse engineer C++ templates.

Reverse engineering for Rational Rhapsody in Java

There is JDK 1.5 support for generics, enumerations, and type-safe containers.

For more specific information about reverse engineering for Rational Rhapsody in Java, see the following topics:

- ◆ [Reverse engineering and Java 5 annotations](#)
- ◆ [Javadoc handling in reverse engineering and roundtripping](#)
- ◆ [Reverse engineering/roundtripping and static import statements](#)
- ◆ [Reverse engineering/roundtripping and static blocks](#)

Reverse engineering other constructs

This section describes how to reverse engineer these constructs: [Unions](#), [Enumerated types](#), and [Comments](#)

Unions

Unions are read in as user types under the corresponding package in the model.

Consider the following source file:

```
union bb32
{
    int x;
    char y;
}
```

The resultant type will be named bb32 and have the following declaration:

```
union %s
{
    int x;
    char y;
};
```

Enumerated types

Enumerated types are read in as user types under the corresponding package in the model.

Consider the following enum:

```
enum cc_buffer_modes
{
    WRITE_MODE,
    READ_MODE_FORWARD,
    READ_MODE_BACKWARD
};
```

The resultant type will be named cc_buffer_modes and have the following declaration:

```
enum %s
{
    WRITE_MODE,
    READ_MODE_FORWARD,
    READ_MODE_BACKWARD
};
```

Comments

During reverse engineering, a comment that comes immediately before the code for an element is considered a comment for that element, and the comment text will be brought into Rational Rhapsody as the description for that element.

You can use the `<lang>_ReverseEngineering::ImplementationTrait::PreCommentSensibility` property to specify the maximum number of lines by which a comment can precede the code for an element and still be considered a comment for that element. Any comment that precedes an element by more than the number of lines specified will be considered a floating comment. For example, a value of 1 means that a comment must appear on the line prior to the code for an element to be considered a comment for that element. The default is 2.

If a C or C++ project has been reverse engineered, the comments are imported as text elements in the relevant SourceArtifacts and are read in as whole blocks. (The comments that are not become a description of some element that is imported.) Then when the code is generated or roundtripped, the comment/text element is placed in its correct place based on its original location.

The following properties are set by default for this feature:

- ◆ `<lang>_ReverseEngineering::ImplementationTrait::RespectCodeLayout` property is set to `Ordering`.
- ◆ `<lang>_CG::Configuration:CodeGeneratorTool` property is set to `Advanced`.
- ◆ `<lang>_Roundtrip::General::RoundtripScheme` property is set to `Respect`.

For information about respect and SourceArtifacts, see [Code respect](#).

Note the following information:

- ◆ If a function has one comment in a .h file and another comment in a .cpp file then the comment in the .cpp file is imported as a floating comment.
- ◆ When any reversed engineered file has a comment as its first element, then any file header comment is turned off. The same is true for any file footer comment.
- ◆ Reverse engineering imports the first/last comment of the file as a regular comment (as a text fragment). Reverse engineering disables generation of the auto-generated header/footer by setting these properties to empty string values.
 - `<lang>_CG::File::ImplementationHeader`
 - `<lang>_CG::File::SpecificationHeader`
 - `<lang>_CG::File::ImplementationFooter`
 - `<lang>_CG::File::SpecificationFooter`

Limitations for comments

Note the following limitations for comments:

- ♦ The following situation might cause comments to not get imported from open `#IFDEF` branches: When an `.h` file is processed more than once and the `#IFDEF` is at one time “open” and at another time “closed,” some comments inside the `#IFDEF` might be lost.
- ♦ Some comments, usually inside an element declaration (like comments on arguments), change their place after code generation. They are generated below the element.

Macro collection

Note

The macro collection feature applies to C and C++.

Macro collecting allows Rational Rhapsody to automatically understand macros in code that will be reverse engineered. This enhances the process for re-using legacy C and C++ code within Rational Rhapsody, providing an easier adoption of *Model-driven Development (MDD)* while enabling a more code-centric workflow.

During reverse engineering, Rational Rhapsody imports “include” files according to the options selected on the **Input** tab of the Reverse Engineering Options window. “Include” files that do not satisfy the specified criteria are not imported into the model.

This can lead to problems if there are files that use macros from “include” files that not will not be imported into the model according to the reverse engineering options selected. To prevent any such problems, Rational Rhapsody goes through all “include” files and collects any macros defined in them.

Note

During reverse engineering, macro collection takes place before import of the files so the macros are taken into account when the model is built.

Collected macro file

The collected macros become part of the model. They are stored in a controlled file called `CollectedMacros.h` which displays in the browser under the configuration used.

Within this controlled file, macros are grouped by their file of origin.

Macros can be modified in or deleted from this file.

Code Generation

Generated code is similar to the original code.

- ♦ Imported elements are generated into the original files.
- ♦ Order of elements is preserved.

When code is generated from the model, the content of the collected macros will be reflected in the generated code (meaning the generated code will not contain references to the macros).

Controlling macro collection

The way that Rational Rhapsody collects these macros can be controlled using the `<lang>_ReverseEngineering::ImplementationTrait::CollectMode` property, which is set at the Configuration level.

This property can take the following values:

- ♦ **None** means that macros will not be collected from include files that are not on the reverse engineering list. This is the default.
- ♦ **Once** means that macros will be collected only if the model does not yet include a controlled file of collected macros.
- ♦ **Always** means that macros will be collected each time reverse engineering is carried out. The controlled file that stores the macros will be replaced each time.

Code generation of imported macros

Note

This feature applies to C and C++.

Rational Rhapsody (through reverse engineering) imports macros unexpanded so that imported macros can behave as calls to macros by default. This means that in subsequent code generation a macro will be generated as is.

The following properties are set by default for this feature:

- ◆ `<lang>_ReverseEngineering::ImplementationTrait::RespectCodeLayout` property is set to `Ordering`.
- ◆ `<lang>_ReverseEngineering::ImplementationTrait::MacroExpansion` property is set to `Cleared`.
- ◆ `<lang>_Roundtrip::General::RoundtripScheme` property is set to `Respect`.

Note that the contents of a macro are not be shown in the model (meaning you will not see its contents in the Rational Rhapsody browser).

For information about respect, see [Code respect](#).

Limitations for imported macros

Note the following limitations for imported macros:

- ◆ The `#define` of a macro needs to be known (as if the file was being compiled).
- ◆ The macro call must occupy a line by itself, meaning that it does not have any other text before or after it, as shown in the following example:

```
// a macro to declare a list of int's  
DECLARE_LIST_OF_TYPE(listName, int)
```

If there is text before or after a macro, as shown in the following example, the macro call will be expanded, meaning that it will not be imported as a macro call:

```
DECLARE_LIST_OF_TYPE(listName, int) // a macro to declare a list of int's
```

Backward compatibility issues

When you open a model created before Rational Rhapsody 7.2, the product, by default, does not import a macro as a call to the macro itself. Instead Rational Rhapsody imports the expanded elements of a defined macro.

Results of reverse engineering

The results of reverse engineering are as follows:

- ◆ Recognized and supported constructs are added to the model.
- ◆ Existing features in a model are updated from the source file to match the source file definition. For example, if the type of an attribute differs in an existing model and a source file being imported, it is changed in the model to match the source file.
- ◆ With the code respect ability, the reverse engineered code in the Rational Rhapsody model respects the structure of the original code and preserved this structure when code is regenerated from the Rational Rhapsody model. The reverse engineered C++ code respects the order, location, and dependencies of the global elements in the original code.
- ◆ Macro collecting allows Rational Rhapsody to automatically understand macros in code that will be reverse engineered.
- ◆ Unresolved elements that are not resolved by the import process remain unresolved.
- ◆ New diagrams or statecharts are not synthesized using imported elements.
- ◆ New model elements found in a source file are added to the browser, but not to existing diagrams.
- ◆ If you selected the **Overwrite existing packages** option on the **Model Updating** tab of the Reverse Engineering Options window existing model elements not found in the file being imported are deleted from the model.

Lost constructs

Some design information might be lost during import if it cannot be represented internally by Rational Rhapsody. Rational Rhapsody can approximate some information, such as non-public inheritance, in which case the construct can be saved. However, if approximation is turned off for a particular construct or if Rational Rhapsody cannot approximate it, the construct will be lost. Subsequent code generation might cause compilation errors.

The following table lists the constructs that are lost on import.

C++ Construct	Description
Anonymous types with members	Enum, class.
Unions	Mapped to an uninterpreted type rather than a special kind of class.
Namespaces	Will be lost if they are not mapped to packages.
Anonymous types with no instances	
Comments that cannot be mapped to code constructs	The last comment, where comments are specified as above the construct; the first comment, where comments are specified as below the construct.
Vendor-specific language extensions	MS DevStudio PASCAL.
Qualifiers	const is shown in the browser as a C++ declaration (volatile).
Storage classes	Auto, register, static, extern, mutable.
Function specifiers	Inline definitions that are part of a function declaration are marked as such, but definitions that are separate from the declaration (even within the same file) are not explicit.
Ellipses in function declarations	

Roundtripping

Roundtripping is an on-the-fly method used to update the model quickly with small changes entered to previously generated UML code. You can activate a roundtrip in batch mode by updating the code in the file system and then explicitly synchronizing the model. You can also activate a roundtrip in an online, on-the-fly, mode by changing a model within one of the Rational Rhapsody designated views. However, roundtripping should not be used for major changes in the model that would require the model to be rebuilt.

With Rational Rhapsody Developer for C and C++ you can roundtrip code into the Rational Rhapsody model in a manner that “respects” the structure of the code and preserves this structure when code is roundtripped in the Rational Rhapsody model. This means the order of elements in the original code can be preserved during code generation and you can freely change the order of class members and globals and Rational Rhapsody respects the change.

When you have changed the order of elements in C and C++, roundtripping in “respect” mode preserves the order of the following elements for the next code generation:

- ◆ Global elements
- ◆ Class elements
- ◆ #includes and forward declarations
- ◆ Auto-generated operations (excluding statechart and instrumentation code)

For more details about code respect and on how to activate it, see [Code respect](#).

Note

Rational Rhapsody has properties that restrict or control how roundtripping changes the model. See [Roundtripping properties](#).

Supported elements

In general, you can roundtrip the following model elements:

- ♦ Classes and class aggregations (template class, types, nested classes)
- ♦ Class elements (attributes, bodies of primitive operations, arguments, types)
- ♦ Class member functions (name, return type, arguments)
- ♦ Operations (comments, name, arguments type/name, arguments addition/removal, return type, visibility)
- ♦ Global elements (functions, variables, types, template functions)
- ♦ Relations (comments, name, other class association, multiplicity, visibility, static-property)
- ♦ Events (comments, name, default argument value, arguments type, arguments addition/removal, return type)
- ♦ Actions placed on transition labels
- ♦ Actions placed inside a state: on entry, on exit, or as a static reaction.
- ♦ Code within annotations
- ♦ `#define-s`.

Roundtripping limitations

Roundtripping is not supported for changes that are made to the following nor to data that are not supported by Rational Rhapsody modifiers (for example, mutable or volatile):

- ♦ Stereotypes
- ♦ States
- ♦ Transitions
- ♦ Precompiled directives (`#pragma`, macros)
- ♦ Header/footer for files
- ♦ Text element for files, not supported by Rational Rhapsody modifiers
- ♦ Component / Configuration information (file mapping)
- ♦ Inline operations generated as `#define`

Dynamic Model-code Associativity (DMCA)

Dynamic model-code associativity (DMCA) changes the code of a model to correspond to the changes you make to a model in the Rational Rhapsody UI. Conversely, when you edit the code of a model directly, DMCA redraws the model to correspond to your edits. In this way, Rational Rhapsody maintains a tight relationship and traceability between the model and the code; the code represents another view of the model. The following **Code > Dynamic Model Code Associativity** menu commands control the DMCA in the model:

- ♦ **Code > Dynamic Model Code Associativity > Bidirectional** changes are automatically put through in both directions. That is, changes that are made to the model cause new, updated code to be generated, and edits made directly to the code are automatically added to the Rational Rhapsody model.
- ♦ **Code > Dynamic Model Code Associativity > Roundtrip** changes made directly to the code are brought into the Rational Rhapsody model, but changes made to the model do not automatically generate new code. You can roundtrip only code that has been previously generated by Rational Rhapsody; that is, you can only edit code that has been previously generated by Rational Rhapsody and incorporate those changes back to the model. See [The roundtripping process](#) for more information.
- ♦ **Code > Dynamic Model Code Associativity > Code Generation** changes made to the model automatically generate new code, but editing code does not automatically change the model.
- ♦ **Code > Dynamic Model Code Associativity > None** means DMCA is disabled. The online code view windows become simple text editors

You can also use the following two settings to control DMCA:

- ♦ `ModelCodeAssociativityMode` in the `rhapsody.ini` file
- ♦ The `General::Model::ModelCodeAssociativityFineTune` property allows you to change the default DMCA mode. However, this is usually set through the menu commands listed previously. (Default = `Bidirectional`)

Note

Dynamic model-code associativity is applicable to all versions of Rational Rhapsody (meaning, Rational Rhapsody Developer for C, C++, Java, and Ada).

The roundtripping process

If you modify source files directly, then select **Code > Generate Code**, Rational Rhapsody prompts you to roundtrip the code. This section specifies your options within the roundtripping procedure.

Automatic and forced roundtripping

If you select automatic roundtripping (the **Bidirectional** or **Roundtrip** setting), the model is automatically updated with code changes if one of the following occurs:

- ◆ You change the window in focus, away from the Code View window.
- ◆ You save the file that you are editing.
- ◆ You close the Code View window.

To force roundtripping, do one of the following actions:

- ◆ Select **Code > Roundtrip**. Code is roundtripped for modified elements.
- ◆ Select **Code > Force Roundtrip**. Code is roundtripped for all elements.

You can force roundtripping at any time. If you have set DMCA to **None** or **Code Generation**, the only way to roundtrip modified code back into the model is by a forced roundtrip.

Roundtripping classes

1. In an OMD, left-click the classes.
2. Select **Code > Roundtrip > Selected classes**, or right-click the classes and select **Roundtrip**.

Rational Rhapsody responds by listing messages of the form “Roundtripping class x” for every class that is roundtripped. Note that if you attempt to roundtrip a class for which code was not generated (or for which the implementation file was erased), Rational Rhapsody responds with an appropriate error message and that class remains unchanged.

Modifying code segments for roundtripping

You can modify code segments for roundtripping that are procedural behaviors entered as operations and actions in statecharts.

Every segment in the implementation code has the following format:

```
... generated code
//[ segment-type segment-identifier
C++ code you entered
//]
... generated code continues
```

All these code segments are in the implementation files.

The only segments you can modify without losing the ability to roundtrip are as follows:

- ◆ Static reactions

```
//[ reaction reaction-id
// you can modify this code
someReactionCode();
// do not modify beyond the //[ sign
//]
```
- ◆ Exit action

```
//[ exitAction ROOT.idle.(Entry)
someExitAction();
//]
```
- ◆ Entry action

```
//[ entryAction ROOT.idle.(Entry)
someEntryAction();
//]
```
- ◆ Transition actions

```
//[ transition transition-id
someTransitionCode();
//]
```
- ◆ Primitive operations (procedural behaviors)

```
//[ operation doit()
someOperationCode();
someMoreOperationCode();
//]
```

Recovering lost roundtrip annotations

To roundtrip your code, Rational Rhapsody uses special annotations inserted by the code generator into the implementation file. The symbols are as follows:

Language	Annotation Symbols
Ada	Element: --++ <ElementType> <ElementName> Body: --+[<ElementType> <ElementName> --+]
C	Element: /*## <ElementType> <ElementName> */ Body: /*#[<ElementType> <ElementName> */ /*#]*/
C++ and Java	Element: //## <ElementType> <ElementName> Body: //#[<ElementType> <ElementName> //#]

Note

If you edit or delete these annotations, Rational Rhapsody cannot trace your code back to the model.

To recover corrupted roundtrip annotations:

1. Rename the damaged file.
2. Regenerate code for this class. This should produce a new file with the correct annotations.
3. Copy your changes from the damaged file into the newly generated file.
4. Try to roundtrip again.

If you have modified any of the files, the following message is displayed:

File <filename> has been modified externally. Do you want to roundtrip?

If you modified the file contents, you must roundtrip to add the modifications to the model. Choose **Yes** to confirm the roundtrip. Rational Rhapsody updates the model and the generated code reflects your manual modifications.

If you choose **No**, Rational Rhapsody overwrites the modified files and your changes will be lost.

Roundtripping classes

The following table lists the modifications that can be roundtripped in a class implementation file.

Element	Change
Constructors and operations	• Change the name of an argument. • When the name is changed, the argument description is lost. • You cannot change the argument type. • Modify bodies between the <code>//#[</code> and <code>//#]</code> or <code>--+[</code> and <code>--+]</code> delimiters.
State actions	Modify state actions (transition, entry, exit, and reactions in state) between the delimiters.
State-action actions	Modify the state-action actions (in activity diagrams) between the delimiters.
Static attributes	Add or modify (but not remove) the initial value.

Note

Actions might appear in the code multiple times.

Rational Rhapsody roundtrips the first occurrence of the action code. If two or more occurrences are modified, the first modified occurrence is roundtripped. One technique is to call an operation in state, state action, and transition actions, thereby eliminating duplication of the action code and possible roundtrip ambiguity.

The following table lists the modifications that can be roundtripped in a class specification file.

Element	Change
Arguments	Add, remove, and change the type of constructors, operations, and triggered operation arguments. Note that changes to argument descriptions in the class specification file are not roundtripped.
Association	Add or remove association, directed association, or aggregation. You must set the property <code>CPP_</code> or <code>JAVA_Roundtrip::Update::AcceptChanges</code> property value to <code>All</code>.
Attributes	- Modify the descriptions. If there is a blank line at the end of the description, the description is lost. - Add or remove attributes. You must set the property <code>CPP_</code> or <code>JAVA_Roundtrip::Update::AcceptChanges</code> property value to <code>All</code>. - Modify the name, type, or access of existing attributes.
Classes	- Modify the descriptions. If there is a blank line at the end of the description, the description is lost. - Modify the class name. In the next code generation, the modified class will be generated to new files, such as <code><new_name>.h</code> and <code><new_name>.cpp</code>. When using DMCA, you must close and reopen the class file to reassociate the class text with the proper class in the model. - Add a new class. The addition will be reflected under the associated package in the model.
Constructors and operations	- Modify the descriptions. If there is a blank line at the end of the description, the description is lost. - Add or remove constructors or operations. You must set the property <code>CPP_</code> or <code>JAVA_Roundtrip::Update::AcceptChanges</code> property value to <code>All</code>. - For the specification file, modify types for existing operation or constructor arguments, but not their names. For specification and implementation files, you can modify both the type and the name. However, if the change is only in the implementation file, you can only change the name and not the type. - Modify return types for existing operations.
Destructors	Modify the descriptions. If there is a blank line at the end of the description, the description is lost.
Nested classes	Add, remove, or modify a nested class.

Element	Change
Relations	- Modify the descriptions. If there is a blank line at the end of the description, the description is lost. - Modify the role name for an existing relation. Given a relation "Class_1* itsClass_1", you can modify the role name itsClass_1. For directed associations, you can also modify the related class Class_1 (for bidirectional association and aggregation, you cannot modify the related class).
Standard operations	Modify standard operations to inline, by adding "inline" to the declaration. Note that the definition is generated automatically. The property <code><lang>_CG::Operation::Inline</code> is set to <code>in_source</code>. As a result, the implementation of the function stays in the implementation file. (The "inline" keyword is added to both, specification and implementation files.)
Triggered operations	Modify the descriptions. If there is a blank line at the end of the description, the description is lost.
User-defined types	Add, remove, or modify user-defined types.

Roundtripping packages

Rational Rhapsody roundtrips function argument name changes in the package implementation file; however, changes to argument types are not roundtripped. When the name is changed, the argument description is lost.

Rational Rhapsody does not roundtrip changes to the initial values of variables.

The following table lists the modifications can be roundtripped in a package specification file.

Element	Change
Event	- Modify the description. Changes to argument descriptions are not roundtripped. - Add or remove event "arguments." Event arguments are actually attributes of the corresponding event class. - Modify an event "argument" type and name. When the name is changed, the argument description is lost.
Function	- Modify the description. Changes to argument descriptions are not roundtripped. - Add or remove a function. - Modify the return type for an existing function.

Element	Change
Function argument	• Add or remove a function argument. • Modify an argument type for an existing function. Changes to argument names are not roundtripped.
Instance	• Add or remove an instance. • Modify a name or class type for an instance.
Variable	• Modify description. Changes to argument descriptions are not roundtripped. • Modify a variable type or name. • Add or remove a variable.

To remove a function, variable, or instance with DMCA active:

1. Remove the element from the .h or .cpp file.
2. Switch focus to the .cpp or .h file while pressing the **Shift** key.
3. Remove the element from the second file.

The **Shift** key prevents DMCA from firing before you have made the changes to the second file.

To remove a function, variable, or instance with DMCA set to **None**:

1. Remove the element from the .h or .cpp file, then save the file.
2. Remove the element from the .cpp or .h file, then save the file.

Roundtripping deletion of elements from the code

Note

This feature applies to C and C++ in Respect mode and Java in Advanced mode.

You can manually delete elements in code and use roundtripping to update your Rational Rhapsody model. You can delete variables, functions, types (Struct, Union, Enum, typedef), types' members, attributes, operations, #define-s, #include-s, forward declarations, and associations. Note that you cannot delete auto-generated #include statements to the Rational Rhapsody framework files.

The roundtripping deletion of elements from the code feature involves the following properties:

1. Depending on whether you have Rational Rhapsody Developer for C, C++, or Java:
 - ♦ For Rational Rhapsody Developer for C and C++: Set the `<lang>_Roundtrip::General::RoundtripScheme` property (for example, `CPP_Roundtrip::General::RoundtripScheme`) to Respect to turn on code respect, which is required for this feature. See [Activating the code respect feature](#).
 - ♦ For Rational Rhapsody in Java: Set the `Java_Roundtrip::General::RoundtripScheme` property to Advanced.
2. For C, C++, and Java: Because the `<lang>_Roundtrip::Update::AcceptChanges` property is, by default, set to Default, the feature to roundtrip hand-edited deletion of elements is available.

Note: Be aware of the following when the `<lang>_Roundtrip::Update::AcceptChanges` property is set to Default:

- ♦ Deletion of the elements Classes, Actors, and Objects is disabled. In addition, deletion of elements is disabled when Rational Rhapsody finds parser errors in the roundtripped code.

Note: You can enable deletion of all elements (no exceptions) and even if there are parser errors during roundtripping. To do so, set the `<lang>_Roundtrip::Update::AcceptChanges` property to All. You should consider the consequences of using the All value. For more information about this property, see [Update::AcceptChanges](#).

- ♦ Deletion of an element that has a prolog and/or epilog is disabled. (You enter values for the prolog and/or epilog in the following properties: `ImplementationProlog`, `SpecificationProlog`, `ImplementationEpilog`, `SpecificationEpilog`.)

Roundtripping for C++

The following details apply to roundtripping in C++ only:

- ◆ You can perform a Advanced (Full) roundtrip for language types in Rational Rhapsody Developer for C++.
- ◆ There is support of #includes and forward declarations.
- ◆ Roundtripping can convert auto-generated operations to user operations on modifying in code through either of the following methods:
 - By setting the `CG::CGGeneral::GeneratedCodeInBrowser` property to Checked.

This works for all auto-generated operations shown in the browser except constructors and destructors.
 - If the above property is not used because there are no auto-generated operations in the browser, then you can remove the “`///
auto_generated`” annotation of the operation so that user operations will be added to the model.
- ◆ Roundtripping takes into account code changes made for all user-defined types.
- ◆ If you change the order of elements, the “code respect” option preserves the order of the following elements for the next code generation:
 - Global elements
 - Class elements
 - #includes and forward declarations
 - Auto-generated operations (excluding statechart and instrumentation code)
- ◆ Position of `<<friend>>` dependency is preserved by roundtripping in code respect mode.
- ◆ You can roundtrip C++ templates.

Roundtripping for Java

The following details apply to roundtripping in Java only:

- ◆ Advanced (Full) roundtrip is supported for Java.
- ◆ You can add “import” statement in the code, which creates a dependency in the model.
- ◆ There is JDK 1.5 support for generics, enumerations, and type-safe containers.

For more information about roundtripping and Java, see the following topics:

- ◆ [Javadoc handling in reverse engineering and roundtripping](#)
- ◆ [Reverse engineering/roundtripping and static import statements](#)
- ◆ [Reverse engineering/roundtripping and static blocks](#)

Roundtripping properties

Rational Rhapsody includes many properties to control roundtripping. They are specified in `<lang>_Roundtrip`, where `<lang>` is the programming language. For example, in Rational Rhapsody in C, these properties are in `C_Roundtrip`; in Rational Rhapsody Developer for C++, they are in `CPP_Roundtrip`.

A definition for each property is provided on the applicable **Properties** tab of the Features window. The following table lists the properties that control roundtripping.

Property	Description
<code>General::NotifyOnInvalidatedModel</code>	Determines whether a warning window is displayed during roundtrip. This warning is displayed when information might get lost because the model was changed between the last code generation and the roundtrip operation. This property is available only in Rational Rhapsody Developer for C and C++.
<code>General::ParserErrors</code>	Specifies the behavior of roundtrip when a parser error is encountered.
<code>General::PredefineIncludes</code>	Specifies the predefined include path for roundtripping. This property is available only in Rational Rhapsody Developer for C, C++, and Java.
<code>General::PredefineMacros</code>	Specifies the predefined macros for roundtripping. This property is available only in Rational Rhapsody Developer for C and C++.
<code>General::ReportChanges</code>	Defines which changes are reported (and displayed) by the roundtrip operation. This property is available only in Rational Rhapsody Developer for C, C++, and Java.

Property	Description
<code>General::RestrictedMode</code>	The <code>RestrictedMode</code> property is a Boolean value (<code>Checked</code> or <code>Cleared</code>) that specifies whether restricted-mode roundtripping is available. This property can be modified on the configuration level. (Default = <code>Cleared</code>) Restricted mode of Advanced (Full) roundtrip enables you to roundtrip unusual usage of Rational Rhapsody elements, such as a class declaration in a user-defined type. Restricted mode has more limitations, but preserves the model from unexpected changes. The additional <i>limitations</i> for restricted mode are as follows: • User-defined types cannot be removed or changed on roundtrip because Rational Rhapsody code generation adds the "Ignore" annotation for a user-defined type declaration. • Relations cannot be removed or changed on roundtrip. • New classes are not added to the model. This property is available only in Rational Rhapsody Developer for C and C++.
<code>General::RoundtripScheme</code>	Specifies whether to perform a <code>Basic</code>, <code>Advanced</code> (for C, C++, and Java only), or <code>Respect</code> (for C and C++ only) roundtrip. <code>Basic</code> is the default for Ada, <code>Advanced</code> for Java, and <code>Respect</code> for C and C++.

Property	Description
Update::AcceptChanges	The AcceptChanges property is an enumerated type that specifies which changes are applied to each code generation element (attribute, operation, type, class, or package). You can apply separate properties to each type of code generation element. The possible values are as follows: • Default means that all the changes can be applied to the model element, including deletion. However, note that deletion is disabled for classes, actors, and objects. In addition, deletion is disabled if Rational Rhapsody finds parser errors in the roundtripped code. This is the default value. • All means all of the changes can be applied to the model element. There are no exceptions (as there are for the Default value). • NoDelete means all the changes except deletion can be applied to the model element. This setting prevents accidental removal of operations, constructors, attributes, relations, variables, instances, and functions. • AddOnly means to apply only the addition of an aggregate to the model element. You cannot delete or change elements. • NoChanges means do not apply any changes to the model element. Note that the value of the property is propagated to all the aggregates of an element. Therefore, if a package has the property value NoChanges, no elements in that package will be changed. This property is available only in Rational Rhapsody Developer for C, C++, and Java.

Code respect

In Rational Rhapsody, code respect means that the order of elements in the original code is preserved during code generation. This means that you can freely change the order of class members and globals and Rational Rhapsody “respects” those changes. Code respect has these additional features:

- ◆ Code generation regenerates text fragments to the correct place in file.
- ◆ Reverse engineering imports `#ifdef`-s to the model as a verbatim text.
 - The branches of `#ifdef`-s that are seen by the compiler are modeled as logical elements.
 - The branches of `#ifdef`-s that **are not** seen by the compiler are modeled as a verbatim text.

This means that code generated in Rational Rhapsody is similar to the original. This gives you complete flexibility for using manually written code or auto-generated code while receiving all the benefits of modeling. You can reverse engineer C++ and C code into a model in a manner that the model respects the order, location, and dependencies of the global elements in the original code. See [Reverse engineering](#).

Note

The code respect feature applies to Rational Rhapsody in C and Rational Rhapsody in C++, and the reverse engineering and roundtripping features in these products. As of Rational Rhapsody 7.2, any new project you create has the code respect featured activated by default. To activate code respect for an old project, see [Activating the code respect feature](#).

In addition, you can set up Rational Rhapsody Developer for C++ and C so that you can roundtrip code into the Rational Rhapsody model that respects the structure of the code and preserves this structure when code is roundtripped in the Rational Rhapsody model.

When you have changed the order of elements in C++ and C, roundtripping in respect mode preserves the order of the following elements for the next code generation:

- ◆ Global elements
- ◆ Class elements
- ◆ `#includes` and forward declarations
- ◆ Auto-generated operations (excluding statechart and instrumentation code)

See [Roundtripping](#).

Activating the code respect feature

To activate the code respect feature for Rational Rhapsody in C++ and Rational Rhapsody in C:

1. Open the Features window.
2. On the **Properties** tab, select the **View** arrow and select **All**.
3. Expand **<lang>_Roundtrip** and then expand **General**.
4. For the **RoundtripScheme** property, select **Respect**.
5. Click **OK**.

Note that the code respect function is based on elaborating SourceArtifact files (previously known as component files).

Note

As of Rational Rhapsody 7.2, any new project you create has the code respect featured activated by default. Any old projects opened in Rational Rhapsody 7.2 or higher retain their original roundtrip scheme.

Where code respect information is defined

Code respect information (such as mapping, ordering, and code snippets) of an element is defined in a *SourceArtifact* element, which is typically created by reverse engineering or roundtripping. The code respect feature applies to Rational Rhapsody C++ and Rational Rhapsody C.

Note that previous to Rational Rhapsody version 7.2, a *SourceArtifact* was referred to as a component file. While component files still exist, they now refer to elements under the **Components** category in Rational Rhapsody. When component files are located under packages or classes, and so on, they are referred to as *SourceArtifacts*.

Note

For existing component files under a component (for example, created by a user or in old models), their locations are not changed.

Because a *SourceArtifact* (for example, a .h file) is located under its applicable class/package/object/block, a configuration management operation of the element includes any *SourceArtifact*.

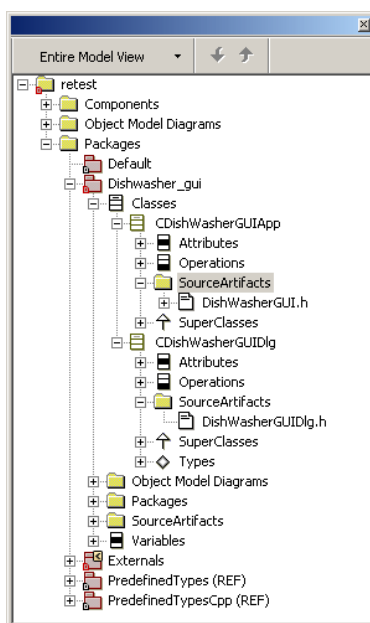
By default the code respect feature is available. When active, the following properties have the following values:

- ◆ <lang>_ReverseEngineering::ImplementationTrait::
LocalizeRespectInformation property set to Checked.
- ◆ <lang>_ReverseEngineering::ImplementationTrait::
RespectCodeLayout property set to Ordering.
- ◆ <lang>_Roundtrip::General::RoundtripScheme property (for example, CPP_Roundtrip::General::RoundtripScheme) set to Respect. See [Activating the code respect feature](#).

Making SourceArtifacts display in the browser

By default, SourceArtifacts do not appear on the Rational Rhapsody browser as only advanced Rational Rhapsody users might want to view or work with them in Rational Rhapsody. Letting Rational Rhapsody and the reverse engineering/roundtripping process manipulate (that is, create and edit) SourceArtifacts is typical.

To show SourceArtifacts on the Rational Rhapsody browser (after you have reverse engineered code into Rational Rhapsody), choose **View > Browser Display Options > Show Source Artifacts**. SourceArtifacts are filed in a **SourceArtifacts** folder, as shown in the following figure where the folder displays under its class.



Manually adding a SourceArtifact

A SourceArtifact is created when you reverse engineer or roundtrip code in Rational Rhapsody, which is the typical way to do so. Advanced users might also want to manually add a SourceArtifact, which you can do through the Rational Rhapsody browser.

To manually add a SourceArtifact:

1. To enable the display of SourceArtifacts on the Rational Rhapsody browser, choose **View > Browser Display Options > Show Source Artifacts**. See [Making SourceArtifacts display in the browser](#).
2. To add a SourceArtifact, right-click a class/package/object/block and select **Add New > SourceArtifact**.

Reverse engineering and SourceArtifacts

Note the following rules for reverse engineering and locating a SourceArtifact under a class:

- ♦ If only one class is mapped to the artifact, the artifact is located under the class it is mapped to.
- ♦ If more than one class is mapped, the artifact is located under the first class. Priority is given to a class with the same of the file.
- ♦ If no class is mapped to the artifact, the artifact is located under the package.
- ♦ External files are imported under the component.

Roundtripping and SourceArtifacts

Note the following rules for roundtripping and locating a SourceArtifact, which are similar as the ones for reverse engineering:

- ♦ New SourceArtifacts are added under classes.
- ♦ The order of elements is updated.
- ♦ For existing component files under a component (for example, created by a user or in old models), their locations are not changed.

Code generation and SourceArtifacts

Note the following rules for code generation and locating a SourceArtifact:

- ◆ Only advanced code generation supports SourceArtifacts.
- ◆ If a class/package belongs to a scope of component, the artifacts under the class are generated according to their code respect information, the same as other component files.
- ◆ If more than one class is mapped to a SourceArtifact, all classes are generated.
- ◆ Other elements mapped to the SourceArtifact are also generated.

Location of the generated files

Note the following rules in relation to the location of the generated files:

- ◆ According to folder hierarchy under component. If a class displays in the scope of the component as an element of the folder, the class and its SourceArtifacts aggregates will be generated according to the path of the folder.
- ◆ According to the “package as directory” policy, the package hierarchy determines the folder hierarchy.

Configuration management and SourceArtifacts

Note the following configuration management considerations:

- ◆ A SourceArtifact under a class/object/block cannot be saved as a unit. Checking in/out the class leads to automatically checking in/out its code respect information.
- ◆ The **Corresponding Component File** check box (on the Add to Archive Options window for configuration management) will not appear unless the configuration management operation is done for a class that is mapped to other’s class SourceArtifact. The same is true for other the other configuration management operations (check in, check out, and so on).

Code-centric mode

For projects that use a model-driven development approach, the basis of the software design is the information in the model. When code is generated from Rational Rhapsody, this code simply reflects the design information stored in the model.

Rational Rhapsody can also be used by projects that use a code-centric approach to development. In this code-centric mode, Rational Rhapsody works on the assumption that the code serves as the blueprint for the software, and that the visual modeling capabilities of Rational Rhapsody are being used primarily to visualize the code.

This assumption that the code takes precedence over the model leads to different behavior on the part of Rational Rhapsody with regard to its code-related features: code generation, reverse engineering / roundtripping, animation.

Note

In terms of programming languages, code-centric mode can be used with C or C++.

When working in C, code-centric mode can only be used with file-based modeling, not object-based modeling.

Entering code-centric mode

There are a number of ways to enter code-centric mode:

- ♦ Create a new model by reverse engineering your existing code.
- ♦ Creating a model from scratch after linking the new model to Rhapsody's code-centric settings.
- ♦ Creating a single component in a project that will use the code-centric settings.

If you are using Rational Rhapsody to visualize existing code:

1. Open the Rational Rhapsody Reverse Engineering window (choose **Tools > Reverse Engineering**).
2. Select the files that should be reverse engineered.

3. Start the reverse engineering process.

When reverse engineering is completed, Rational Rhapsody will automatically enter code-centric mode.

If you are creating a new model and do not plan to import any existing code, but still want to take advantage of the code-centric approach:

1. Open the New Project window (choose **File > New**).
2. Using the Project Settings drop-down list, select the code-centric settings.

If you want to create a single component in a project that will use the code-centric settings:

1. Use **File > Add to Model** to add the code-centric settings to the model (*CodeCentric75Cpp.sbs* or *CodeCentric75C.sbs*)
2. Create a new component, and add a dependency from the component to the code-centric settings.
3. Apply the *AppliedProfile* stereotype to the dependency.
4. Generate code using the new component. This code will then serve as the blueprint for your model.

Regardless which of these methods you used to enter code-centric mode, you will notice the following in the Rational Rhapsody browser:

- ◆ The code-centric settings will be displayed under the Settings category.
- ◆ The entire project or individual components will have a dependency upon the code-centric settings with the *AppliedProfile* stereotype applied to the dependency.

Note

The dependency will be at the project level only if you entered code-centric mode by selecting the code-centric settings in the New Project window.

Leaving code-centric mode

To switch from code-centric to model-centric mode:

1. Delete any dependencies upon the code-centric settings.
2. Highlight the code-centric settings in the browser, and open the pop-up menu.

3. Select **Delete from Model**.
4. To prevent all your existing code from being overwritten the next time code is generated, open the Features window for the active component and enter a different path for the **Directory** field on the **General** tab.

Roundtripping in code-centric mode

In the most extreme code-centric scenario, you could use Rational Rhapsody only to visualize your code and not add any non-code-related elements to your model. In such a scenario, you could theoretically use Rational Rhapsody to reverse engineer your entire code periodically and there would be no reason to save this model in between such reverse engineering sessions.

The Rational Rhapsody code-centric mode is designed to combine this code-based focus with the ability to add non-code-related elements to your model.

In code-centric mode, Rational Rhapsody will roundtrip into your model any changes you make to your code, regardless of how drastic these changes are. At the same time, Rational Rhapsody allows you to add non-code-related elements to your model, for example, requirements, and keeps this information in your model permanently. The roundtripping of the changes to your code does not affect in any way the non-code-related elements you have included in your model.

There are a number of ways to initiate the roundtripping of your code changes into your model:

- ♦ If you are using the DMCA feature, then when you change the focus from the code editor to the browser, any changes you have made to your source files will be brought into your model.
- ♦ **Code > Roundtrip** - Rational Rhapsody looks for changes only in the files modified since the last time the model was updated from the code.
- ♦ **Code > Force Roundtrip** - Rational Rhapsody looks for changes in all the files, even those not modified since the last update.

The most important distinction between roundtripping in code-centric mode and roundtripping in model-centric mode is the following: *In model-centric mode, code generation is always carried out after the manual code changes have been imported into the model (hence the term "roundtripping"). In code-centric mode, however, Rational Rhapsody never regenerates the code after the changes have been imported into the model (it is only a one-way process).*

If you create a new file in the folder that contains your source code, Rational Rhapsody will import the contained elements into the model the next time you roundtrip.

Note

When you add a new file to the folder containing your source code, the contained elements will be imported into the model only if you selected the folder at some point in the Reverse Engineering window, as opposed to specifying individual files.

When you roundtrip code changes into the model, object model diagrams will be updated accordingly. To turn off this default behavior, modify the value of the property `ObjectModelGe::AutoPopulate::EnabledOnUpdateModel`.

Code generation in code-centric mode

When working in the Rational Rhapsody model-centric mode, if you select **Code > Generate**, Rational Rhapsody generates completely new files for the files/classes you have specified, whether it is selected classes, a specific configuration, or the entire project.

In code-centric mode, the code-generation behavior of Rational Rhapsody is based on the premise that if you add any code-related elements to your model, you would prefer that Rational Rhapsody make as few changes as possible to your code. So if you use the **Generate** option in code-centric mode, Rational Rhapsody does not regenerate the entire file. Rather, it generates only the code segments that represent the new elements that were added and inserts them in the appropriate location in your code. The rest of your code remains exactly as it was.

This principle of minimal intrusion into your code results in a number of other differences in code generation behavior, relative to code generation in model-centric mode:

- ◆ The generated code does not include Rational Rhapsody annotations.
- ◆ Auto-generated code, such as getters/setters and default constructors/destructors, is not generated.
- ◆ If your code contains code elements that cannot be imported into a Rational Rhapsody model, this code will remain even after you have used the Rational Rhapsody code generation feature.
- ◆ In order to keep formatting as consistent as possible, the indentation used for Rational Rhapsody-generated code elements is based on the indentation of the code preceding the code that is being added.

Other code-generation behavior in code-centric mode:

- ◆ If you add a file in C or a class in C++ to your model, Rational Rhapsody will generate new files for them when you use the **Generate** option.
- ◆ If you change an element name in the model, all references to it in your code will be updated the next time you generate code.
- ◆ When you add a new element to a class, it will be added to the code following the last element with the same visibility. If there are not elements with the same visibility, it will be added at the end of the code for the class.

Note

Code-centric mode's selective code updating is only available when using the Rational Rhapsody *Advanced* code generation setting. This means that code-centric mode cannot be used with older models unless you change the `CodeGeneratorTool` property from *Basic* to *Advanced*.

Diagrams for which code not generated

When working in code-centric mode, code is not generated for statecharts or activity diagrams.

Code regeneration in code-centric mode

The **Generate** menu option starts the selective code generation process. However, the **Regenerate** option will regenerate the entire file. If you use the **Regenerate** option to generate an entire file, The Rational Rhapsody selective code update feature will not be used until you have roundtripped the file.

Because source code files might contain elements that cannot be brought into a Rational Rhapsody model, if you decide to delete your code and regenerate all the code from your model, the code will not look the same as your original code and might not even compile. The same is true if you set a new Directory for your component - in such a case Rational Rhapsody will regenerate all the code and this code will not necessarily include everything that was in your original code, leaving open the possibility that the code might not even compile.

Animation in code-centric mode

The Rational Rhapsody animation feature is made possible due to instrumentation code that Rational Rhapsody inserts when it generates code for configurations where Instrumentation Mode has been set to **Animation**.

Because the underlying approach in code-centric mode is to minimize the intrusion into your code, there is a difference in the way animation code is generated in code-centric mode, compared to animation code generated in model-centric mode:

In code-centric mode, when code is generated for animation, only the files that contain animated elements are generated, rather than all the files as is the case in model-centric mode.

As is the case with model-centric mode, animation code that is generated is framed within `#ifdef _OMINSTRUMENT` blocks. If you make any changes to the code within these blocks, roundtripping will ignore these changes.

Because code is not generated for diagrams such as statecharts and activity diagrams in code-centric mode, the only type of diagram that can be animated is a sequence diagram.

To further minimize the intrusion into your code as the result of animation, you can use the following Rational Rhapsody features when animating in code-centric mode:

- ♦ The **Animate** option in the context menu for Sequence Diagrams can be used to specify that only certain sequence diagrams should be animated. (When you use this option, all classes included in the diagram will be animated.)
- ♦ The Advanced Instrumentation Settings window (accessed from the **Advanced** button on the Settings tab of the Features window for configurations) can be used to specify that instrumentation code should only be generated for certain type of elements, such as operations.
- ♦ The Advanced Instrumentation Settings window can be used to specify that instrumentation code should only be generated for specific diagrams and/or classes.

While the instrumentation code is less intrusive in code-centric mode, it is important to keep in mind the following when using animation:

- ♦ In ordinary code-centric code generation, code is generated only for modified elements within a file. When using animation, however, files that contain any animated elements will be regenerated in their entirety.
- ♦ If you are working with a single Rational Rhapsody configuration and change the Instrumentation Mode to Animation, then the files generated by Rational Rhapsody will overwrite those currently in your output directory.
- ♦ If a file has been generated for animation purposes, it will be generated in full each time you generate code (selective code updating is not used) until the next time you roundtrip

the code. This means that to restore selective code update behavior for files that contained instrumentation code, you must:

- change the animation settings so that instrumentation code is not generated
 - regenerate the code
 - roundtrip the generated file
- ♦ Even though *auto-generated* code is not usually generated in code-centric mode, if you are using animation there are cases where Rational Rhapsody might generate some *auto-generated* elements, for example, generating a constructor if your code does not contain one.

Scope for code-centric models

The scope defined for a component determines both the code generation scope and roundtripping scope. Code is generated for modified elements only if the files that contain them are included in the component scope, and changes to files are roundtripped into the model only if the modified files are included in the component scope.

When you first reverse engineer files, all of these files are added to the scope. If you specified a folder in the Reverse Engineering window, then any files you create in that folder will automatically be added to the scope. Similarly, if you delete any files from that folder, the corresponding elements will be removed from the model when you roundtrip.

If you add a file (in C) / class (in C++) to the model and update the code, the generated file will be added to the scope. This is true even if you generate the file to a folder that is not defined as part of the scope.

You can manually modify the scope by adding files/folders in the Roundtrip Settings window (**Code > Roundtrip/Forced Roundtrip > Settings**).

When modifying the scope, it is important to keep in mind that in code-centric mode the package structure in Rational Rhapsody following roundtripping will always be identical to the directory structure of your source code. This rule has a number of implications:

- ♦ If you would like to reverse engineer files in increments, make sure that all the directories containing your code are underneath a single directory that can be specified as the Root Directory on the Mapping tab of the Reverse Engineering Advanced Options window. You can then import any directories under this root directory, and Rational Rhapsody will create a package structure that matches your directory structure. If you are using this type of incremental approach, you should set the value of the property `UseCalculatedRootDirectory` to *Never* prior to carrying out reverse engineering.
- ♦ If you change the root directory, the package structure in your model will be different following roundtrip. It is possible that some packages will be removed from the model and then recreated. If one of these packages contains elements not reflected in your code,

for example requirements, then these elements will no longer exist in the model following roundtripping since they do not have any representation in the code that was roundtripped.

Note: These consequences of changes to the package structure in Rational Rhapsody apply also when you are just manually changing the hierarchy of the folders that contain your source code. Before roundtripping code changes after such folder adjustments, you should make similar changes to the package structure in your Rational Rhapsody model to ensure that no code-less elements disappear.

This “remove and recreate” approach is also used if you rename individual elements in your code such as classes. Since classes in your model may contain significant information that is not represented in your code, make sure that such renaming is not going to result in loss of information from your model.

- ♦ If you are importing code from directories that do not have a common ancestor directory directly above them, you will have to import them into different Rational Rhapsody components.

Properties modified by code-centric settings

The different code-generation and roundtripping behavior in code-centric mode is the result of new properties defined in the code-centric settings or the inclusion of property values that differ from the default values used in model-centric mode. These properties / property overrides come from the code-centric settings (sbs) file, and they include:

- ♦ `[lang]_ReverseEngineering::ImplementationTrait::VisualizationUpdate` - exists only in code-centric settings, set to True, responsible for code-centric behavior of reverse engineering / roundtripping
- ♦ `[lang]_Roundtrip::Update::AcceptChanges` - in code-centric mode, the code changes that roundtrip imports are broader than in model-centric mode. In code-centric mode, the value of `AcceptChanges` is All, meaning any additions, changes, or deletions will be reflected in the model. In model-centric, Rational Rhapsody does not roundtrip the deletion of classes or deletions that result in parser errors.
- ♦ `[lang]_Roundtrip::Update::MergePolicy` - this property determines the merge policy Rational Rhapsody uses during roundtripping when comparing the model based on the latest code to the saved model. In the code-centric settings, this property is set to `CodeDriven`. This value means that Rational Rhapsody imports certain types of code elements that it does not import when working in model-centric mode. This property differs from the property `AcceptChanges` in that `AcceptChanges` deals with changes to model elements (adding, deleting, modifying) while `MergePolicy` is used to indicate to Rational Rhapsody that the code, rather than the model, should be given precedence when it comes to merging changes.
- ♦ `[lang]_CG::Configuration::CodeUpdate` - exists only in code-centric settings, set to True, responsible for the selective code generation used in code-centric mode

- ◆ `[lang]_CG::ModelElement::SimplifyAnnotations` - value is set to `CodeUpdateAnnotations` in order to minimize the Rational Rhapsody annotations generated in code, limiting them to special cases such as animation.

In addition, a number of code generation property values are overridden to prevent generation of *auto-generated* code. These include:

- ◆ for classes - properties such as `CreateImplicitDependencies`, `GenerateImplicitConstructors`, `ImplementStatechart`, `ImplicitDependencyToPackage`, `GenerateDestructor`
- ◆ for relations - properties such as `AddComponentHelpersGenerate`, `AddGenerate`, `ClearGenerate`, `CreateComponentGenerate`, `DeleteComponentGenerate`, `GetEndGenerate`, `GetGenerate`, `RemoveComponentHelpersGenerate`, `RemoveGenerate`, `RemoveHelpersGenerate`, `SetComponentHelpersGenerate`, `SetGenerate`, `SetHelpersGenerate`
- ◆ for attributes - properties such as `AccessorGenerate` and `MutatorGenerate`
- ◆ for events - `Generate`
- ◆ for ports - `Generate`
- ◆ `MainGenerationScheme`, which controls initialization code
- ◆ properties that control generation of headers and footers
- ◆ properties used for the inclusion of framework header files (such as `IncludeHeaderFile`), for example, `oxf.h`

Animation

Rational Rhapsody animation is a key technology that enables model validation. You can validate the analysis and design model by tracing and stimulate the executable model. In addition, the Rational Rhapsody animator helps you debug your system at the design level rather than the source code level by actually executing the model and animating the various UML design diagrams. Rather than merely simulating the application and viewing values of variables and pointers, you see actual values of instances of states and relations.

The animator enables you to juxtapose different views of an application while it is running. You can watch the animated model executing in any of the following views:

- ◆ Sequence diagrams
- ◆ Statecharts
- ◆ Activity diagrams
- ◆ Browser
- ◆ Event Queue window
- ◆ Call Stack window
- ◆ Output window

Simultaneously viewing animated sequence diagrams, animated statecharts, animated activity diagrams, and the animated browser in adjacent windows as the model is executing enables you to verify that the design behaves as wanted. Highlighting in the animated diagrams helps you to pinpoint the current state of execution.

While the model is running, you can use the **Animation** toolbar to step through the program, set and clear breakpoints, and inject events to observe how the system reacts in quasi-real time. You can observe the operation for the system either in the animated views or by generating an output trace.

Animation Overview

The animator is a *design-level* debugger, as well as a model validator. In other words, the animator supports the standard functionality of a programming language debugger at the design level. The objects you follow are design-level objects; that is, objects that are modeled in Rational Rhapsody.

Animation Features

During an animation session, you can perform the following activities:

- ♦ Inspect and modify the current status of the model:
 - View current instances and the relationships between them.
 - View the current state for reactive objects.
 - View animated sequence diagrams depicting events and operations actually sent or called.
 - Generate events.
- ♦ Open or close animated views.
- ♦ Set breakpoints.
- ♦ Advance execution using the Go buttons on the **Animation** toolbar.

Preparing for Animation - General Procedure

The following procedure lists the general steps to prepare for and run animation, with references to the more specific procedures.

1. If necessary, create a component. See [Create a Component](#).
2. If necessary, create a configuration. See [Creating a Configuration](#).
3. Set the Instrumentation mode for the configuration to **Animation**. See [Setting the Instrumentation Mode](#).
4. Set the active configuration. The active configuration is the one generated when you generate code. See [Setting the active configuration](#).
5. Generate code for the configuration. See [Generating Code](#).
6. Build the animated component. See [Building the Target](#).
7. Run the animated component. See [Running the Animated Model](#).

Create a Component

A *component* is a physical subsystem in the form of a library or executable program. It plays an important role in the modeling of large systems that contain several libraries and executables. Each component contains configuration and file specification categories, which are used to generate, build, and run the executable model.

Each project contains a default component, named `DefaultComponent`. You can use the default component (you can rename it) or create a new component.

Creating a component

To rename the default component:

1. In the Rational Rhapsody browser, expand the **Components** category.
2. Double-click **DefaultComponent** to open the Features window.
3. In the **Name** box, replace the name `DefaultComponent` with another name.
4. Click **Apply**.
5. To set the features for this component, see [Setting the Component Features](#)

To create a new component:

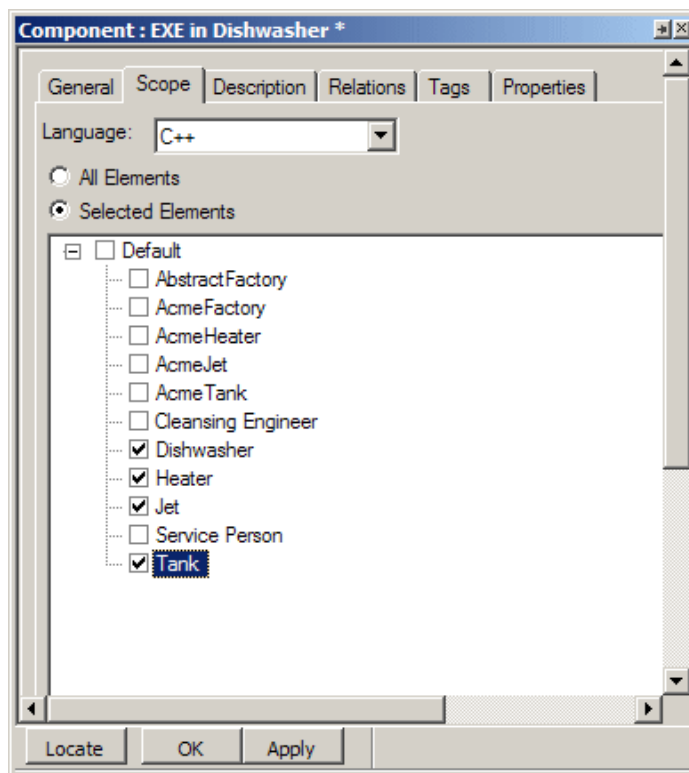
1. In the Rational Rhapsody browser, right-click the **Components** category and select **Add New Component**.
2. Type a name for your new component and press **Enter**.
3. To set the features for this component, double-click the new component to open its Features window and see [Setting the Component Features](#).

Setting the Component Features

Once you have created the component, you must set its features.

To set the component features:

1. With the Features window open for your component, on the **General** tab, in the **Type** group, select the **Executable** radio button if it is not already selected.
2. On the **Scope** tab, specify which model elements to include in the component.
 - ♦ **All Elements.** Select this radio button if you want to select all available elements.
 - ♦ **Selected Elements.** Select this radio button if you want to select only certain elements and then use the check boxes next to each element to indicate which model elements to include in the component. Notice that if you select check box next to the parent element, all sub-elements are selected. If you only want certain sub-elements, select the check boxes next to those sub-elements instead of the parent element, as shown in the following figure:



3. Click **OK**.

Creating a Configuration

A component can contain many configurations. A *configuration* specifies how the component is to be produced.

Each component contains a default configuration, named `DefaultConfig`. You can use the default configuration (you can rename it) or create a new one.

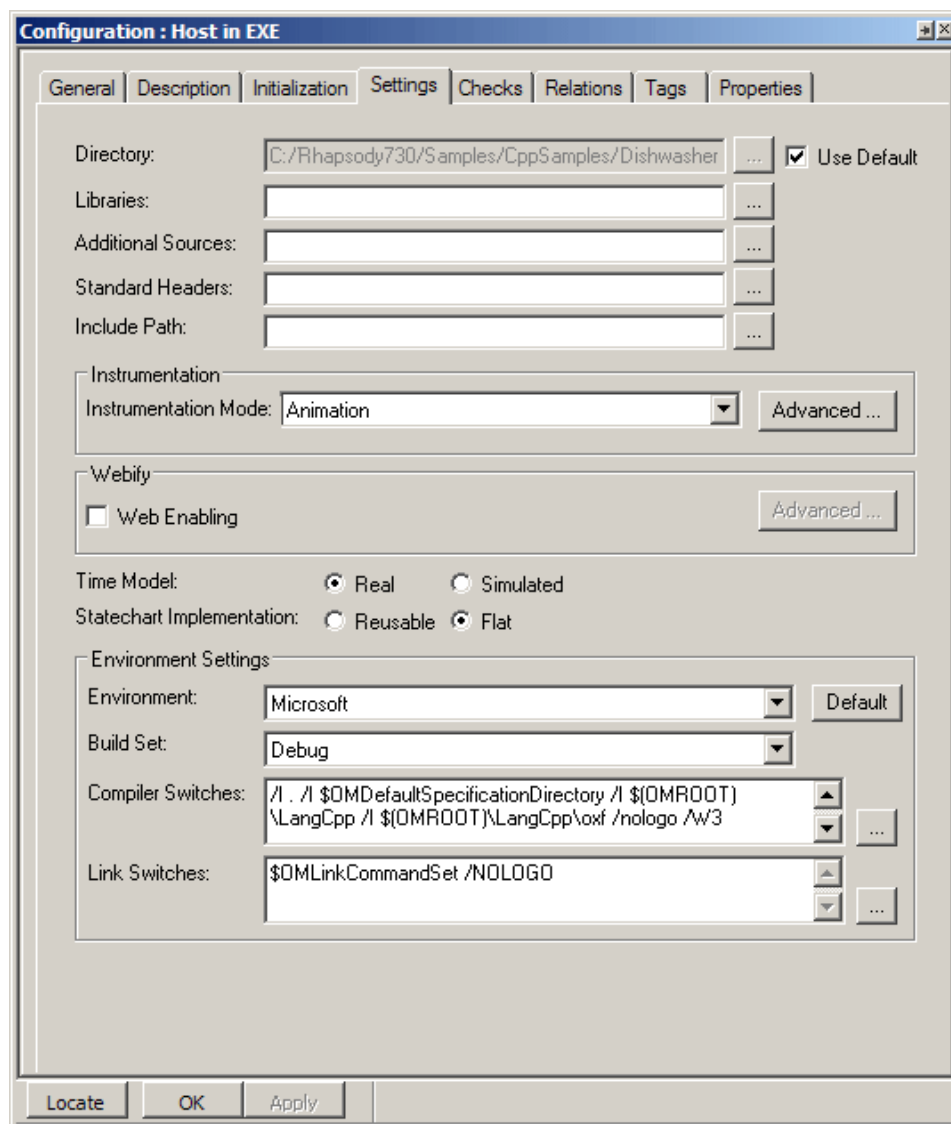
To rename the default configuration:

1. In the Rational Rhapsody browser, expand the applicable component and its **Configurations** category.
2. Double-click **DefaultConfig** to open the Features window.
3. In the **Name** box, replace `DefaultConfig` with another name.
4. Click **Apply**.
5. To enable animation, you must set the instrumentation mode. See [Setting the Instrumentation Mode](#).

Setting the Instrumentation Mode

To set the Instrumentation mode for a configuration:

1. With the Features window open for your configuration, on the **Settings** tab, set the **Instrumentation Mode** box to **Animation**, as shown in the following figure. This adds instrumentation code, which makes it possible to animate the model.



2. Optionally, to instrument operations and set a finer scope on the instrumentation, click the **Advanced** button to open the Advanced Instrumentation Settings window. For more information, see [Using selective instrumentation](#).
3. Click **OK**.

Running the Animated Model

When running the animated model, you can use any of these animation methods:

- ♦ Run an executable application on the same machine as Rational Rhapsody (see [Running on the Host](#)).
- ♦ Run an executable application on a different machine than Rational Rhapsody (see [Running on a Remote Target](#) and [Testing an Application on a Remote Target](#)).
- ♦ Test a library built with Rational Rhapsody with a GUI built outside of Rational Rhapsody (see [Testing a Library](#)).

For all of these animation methods, the process follows these steps:

1. The application auto-connects to Rational Rhapsody to run the animation.
2. The message “Initializing animation...” is displayed.
3. An operating system window opens to display console output from the application. You can minimize this window, if wanted.
4. The **Animation** toolbar displays (see [Animation Toolbar](#)).

Running on the Host

To run the application on the host:

1. Open a command prompt window.
2. Type the DOS command `ipconfig`.
3. Copy the IP address of the host into the command and press **Enter**.
4. Open your Rational Rhapsody project.
5. Highlight your EXE for the host in the browser and right-click to display the Features window.
6. Set the `<Language>_CG::<Compiler>::UseRemoteHost` property value to **Checked**.
7. Also in the properties list, locate the `<Language>_CG::<Compiler>::RemoteHost` property and paste in the IP address of the host for its value. Click **OK**.
8. Generate and make the executable.

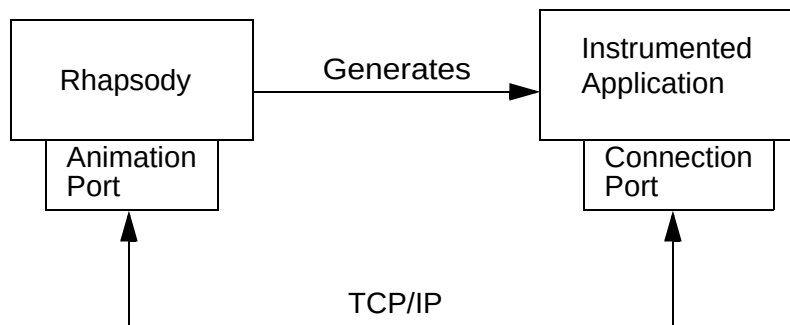
Running on a Remote Target

You can inspect or debug animated code running on a remote target through these steps:

1. Copy the executable from the host to the target.
2. Run the executable.
3. Check to be certain that the animation is running on the host.

The animation views, shown on the host, are at the design level. The animation server communicates with the target via a TCP/IP connection. Each animation connection requires its own unique port number, which is set in the `rhapsody.ini` file. The framework inserts the same port number into the connection port of the instrumented application.

The instrumented application can run on either the same machine as Rational Rhapsody (the host) or a remote target. The following illustration shows the relationship of these components to each other.



Opening a Port Automatically

If you want Rational Rhapsody to locate an open port automatically for animation, change the values of the following variables in the [General] section of the `rhapsody.ini` file:

- ♦ **AnimationPortNumber** defines the port to try first
- ♦ **AnimationPortRange** specifies the number of ports to test, in addition to AnimationPortNumber, before giving up

These changes set up the search to start with the AnimationPortNumber and increment by one until either an open port is found or until the number of tries equals the value of AnimationPortRange + 1.

For example, if AnimationPortNumber = 5000 and AnimationPortRange = 100, Rational Rhapsody first attempts to establish a connection on port 5000. If that fails, then it tries port 5001. If that fails, then it tries port 5002. This search continues until either an open port is found or until it finally tests port 5100. If port 5100 fails, then no animation can be performed for that instance of Rational Rhapsody.

In addition to the animation ports, another `rhapsody.ini` file value needs to be manually set. In the [General] section, change the EnableMultipleAnimation from FALSE (default) to TRUE.

Testing an Application on a Remote Target

To test an application running on a remote target:

1. Open a command window and change to the directory where the application is located.
2. At the command prompt, enter the command to run the application with the `-hostname` option indicating the machine running Rational Rhapsody. For example, if your application is named `myapp.exe` and Rational Rhapsody is running on a machine named Julius, run your application using the following command:

```
myapp -hostname Julius
```


Testing a Library

To test a Rational Rhapsody library:

1. Build the library inside Rational Rhapsody.
2. Build the application outside Rational Rhapsody and include the library built in Rational Rhapsody.
3. Open a command window and change to the directory where the application is located.
4. At the command prompt, enter the command to run the application. For example, if your application is named `myapp.exe`, use the following command:

```
myapp
```

Partially Animating a Model (C/C++)

Rational Rhapsody enables you to partially animate a model, to test selected elements without the overhead of animating the entire project. This feature also enables you to animate projects that contain non-animated components.

All elements in the model are generated and built, but only the animated elements are displayed in the animation environment during run time. Animated and nonanimated elements, listed below, are able to pass instances and events to one another, but only animated events can be generated using the animation environment.

- ◆ Package
- ◆ Class (including nested classes)
- ◆ Statechart
- ◆ Activity diagram
- ◆ Relation
- ◆ Attribute
- ◆ Actor
- ◆ Operation
- ◆ Event

Note

Partial animation also applies to trace operations.

Setting Elements for Partial Animation

1. Set the instrumentation of the active component to **Animation**.
2. By default, all elements are animated. Therefore, partial animation is a process of elimination.

For each element that you do not want animated, set the
`<lang>_CG::<type_of_element>::Animate` property to `False`.

The `<type_of_element>` metaclass can be `Package`, `Class`, `Statechart`, `Relation`, `Attribute`, `Operation`, `Event`, or `Actor`.

3. Save the model.
4. Generate, make, and run the project.

Note: You can specify partial animation per configuration using the Advanced Instrumentation Settings window. For more information, see [Using selective instrumentation](#).

Partial Animation Considerations

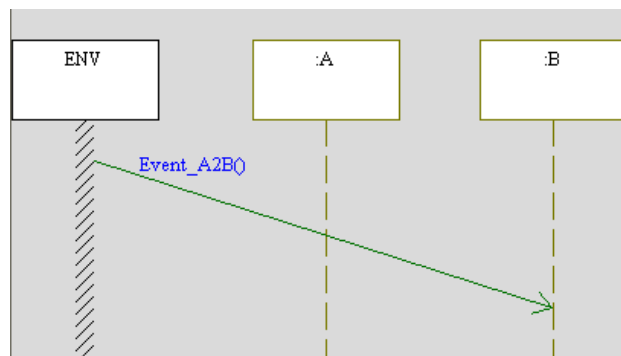
- ◆ Animation for components should be set at the configuration level.
- ◆ If any element in the executable is animated, the instrumentation should be set to **Animation** in the Configuration setting for the executable component. The component, which creates the executable, is linked with the instrumented OXF libraries if at least one part of the model (aggregate, library, and so on) is animated.
- ◆ If an operation is animated, its arguments are also animated. When setting animation for arguments, it is active or unavailable for all arguments. Arguments cannot be animated individually.
- ◆ The animation setting of a class affects all of its instances, even if the package they belong to is not animated.
- ◆ Only animated events can be injected using the animation environment.
- ◆ Only animated events appear in the event queue.
- ◆ Elements that are not animated are not shown in any animated view.
- ◆ Only animated attributes or relations are shown for an animated instance.
- ◆ Only animated operations, timeouts, and events are displayed in the animated sequence diagram and the call stack.
- ◆ If a model contains an active class that is not animated, its thread is considered a foreign thread. This thread displays in the thread list, but is named by its operating system handle (not by the name of the active object).

- ◆ When using Rational Rhapsody-generated DLLs, the OXF libraries should also be a DLL. This DLL should be animated if any of the Rational Rhapsody-generated DLLs are animated.

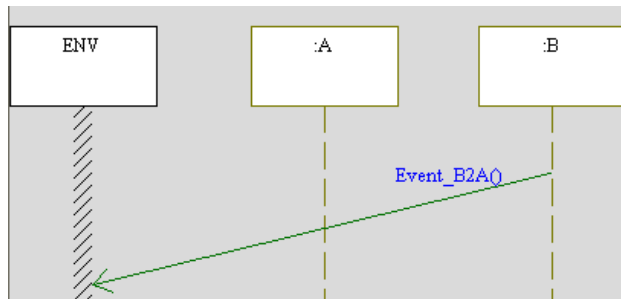
Partially Animated Sequence Diagrams

Instances that are not animated are not displayed in animated views. However, Rational Rhapsody does display messages passed between animated and nonanimated instances.

If a nonanimated instance (A) sends a message, it is shown originating from the system border.



If an animated instance (B) sends a message to a nonanimated instance (A), it is shown being sent to the system border.



A message sent from a nonanimated class is shown coming from its call stack predecessor, if present. Messages sent from a nonanimated class whose call stack predecessor is absent are shown originating from the system.

Messages sent from one nonanimated class to another are not shown.

Ending an Animation Session

To end an animation session, use any of the following methods:

- ♦ In the **Animation** toolbar, click the **Quit Animation** tool.
- ♦ Select **Code > Stop**.
- ♦ Allow the application to terminate.

Animation Toolbar

When you run an executable model with instrumentation set to **Animation**, Rational Rhapsody displays the **Animation** toolbar. This toolbar automatically appears during an animation session. To display or hide this toolbar during an animation session, select **View > Toolbars > Animation**.

The **Animation** toolbar contains the following tools.

Tool Button	Button Name	Description
	Go Step	Advances the application a single step (until the next Rational Rhapsody-level occurrence, such as the calling of an operation).
	Go	Advances the application until it terminates or reaches a breakpoint.
	Go Idle	Advances the application until the next timeout or event for the focus thread. If there are no timeouts or events waiting in the event queue, nothing happens.
	Go Event	Advances the application until the next event is dispatched or the executable reaches an idle state.
	Animation Break	Interrupts a model that is executing and suspends the clock.
	Command Prompt	Opens the Animation Command window where you can type animation and trace commands.
	Quit Animation	Ends the animation session and terminates the executable.
	Thread	Opens the thread view.
	Breakpoints	Opens the Breakpoints window, where you can control breakpoints. When a breakpoint is encountered in any thread, the animator switches control from the application to you. The last thread to reach a breakpoint becomes the focus thread.
	Event Generator	Opens the Events window, where you can generate events using the Event Generator.
	Operation Calls	Opens the Operations window. Lets you start operation calls during animation and tracing to validate parts of the design model.

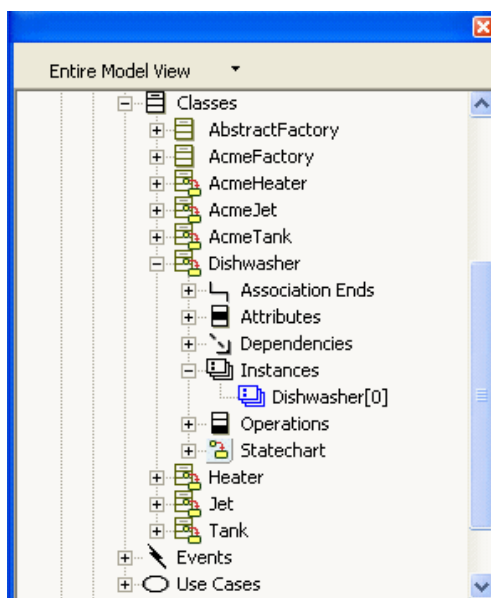
Tool Button	Button Name	Description
	Silent/Watch	Toggles between Silent and Watch mode at any break or when the executable is idle. Silent mode enables you to perform design-level debugging and display animation information only at breakpoints. In contrast, Watch mode enables you to continually update animation information in normal step-by-step operation via the Go buttons.

Note that all the **Go** commands cause the tracer to execute immediately, even if it is in the middle of reading commands from a file. The subsequent (unread) lines are used as commands the next time the tracer stops execution and searches for commands. If the model reaches a breakpoint, control can return to you before a **Go** command is complete.

Creating Initial Instances

It is a good idea to issue a **Go Idle** command immediately after starting an executable model so all initial instances are created.

To create instances, click **Go Idle** after starting the model. The initial instances are created (as well as any instances created by those instances) and are listed under the Instances category for the class in the browser.



The instance name is in the format `class[n]`, where *n* is the number of instances, beginning with 0, that have been created since the model began executing. It is not possible to change this number.

Break Command

To interrupt a model that is executing, click the **Break** tool.

The **Break** command enables you to regain control immediately (or as soon as possible). Issuing a **Break** command also suspends the clock, which resumes with the next **Go** command.

Note

For simple applications, there might be a backlog of notifications. Although the model stops executing immediately, the animator can accept further input only after it has cleared this backlog and displayed any pending notifications.

The **Break** command cannot stop an infinite loop that resides within a single operation. For example, issuing a **Break** cannot stop the following `while()` loop:

```
while(TRUE) j++;
```

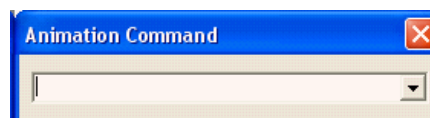
However, it can stop the following code if `increaseJ()` is an operation defined within Rational Rhapsody:

```
while(TRUE) increaseJ();
```

Command Prompt

To issue an animation command, for example, to manually inject an event into the model, click the **Command Prompt** tool.

The Animation Command bar displays, as shown in the following figure:



You can also generate events using the **Event Generator** tool. See [Event Generator](#).

Generating Events Using the Animation Command Bar

Before you can inject an event into the model, the instance to which you are sending the event must already exist. See [Event Generator](#).

To generate an event:

1. In the Animation Command bar, type a `GEN()` command using either of the following formats:

- ◆ `instance->GEN(event)`
- ◆ `instance->GEN(event())`

2. Press **Enter**.

If the event is one that the instance is able to receive, the event is entered into the call stack (see [Call Stack](#)) in the following format:

```
instance->event()
```

If the instance is not able to receive the event, or no such event is defined in the package, the message "Invalid event name or non-existing event class <event>" is displayed.

3. To process the event, click one of the **Go** tools.

Events with Arguments

If the event has arguments, the GEN command is as follows:

```
instance->GEN(event(parameter[, parameter]*))
```

In this command:

- ◆ `instance` specifies the name of the instance to which you are sending the event (using the format `class[n]`)
- ◆ `event` specifies the name of the event you want to generate
- ◆ `parameter` specifies the values of the actual arguments to be passed to the event

When the event is generated, the actual argument names and their values appear in the call stack in the following format:

```
instance->event(argument = parameter[,  
argument = parameter]*))
```

If an event has arguments, you should provide the GEN command with the correct number of parameters and the correct types. For example, if event `X` is defined as `X(int, B*, char*)` where `B` is a class defined in Rational Rhapsody, to generate an event you can enter either of the following commands:

```
A[1]->GEN(X(3, B[5], "now"))  
A[1]->GEN(X(1, NULL, "later"))
```

Event arguments must be either pointers to classes defined in Rational Rhapsody, or of a type that can be read from a string, such as `int` or `char*`. If you want to generate an event of a user-defined type that you have defined either inside or outside of Rational Rhapsody, you must either overload its I/O stream operator `>>(istream&)`, or instantiate the template `string2X(T& t)` so Rational Rhapsody can interpret the characters entered.

The command `A[1]->GEN(Y(1))` works because the `>>` operator automatically converts the character "1" to the integer 1. On the other hand, the command `A[1]->GEN(Y(one))` would not work because the `>>` operator cannot convert the characters "one" to an integer.

Note

If you pass complex parameters (such as `structs`) and use animation, you must override the `>>` operator. Otherwise, Rational Rhapsody generates compilation errors.

Generating Events Using the Command History List

The Animation Command bar has a list of commands previously issued in the same animation session. You can select a previous command from this list, rather than retyping it.

To select a previous command from the history list:

1. Click the down arrow to the right of the command-entry box.
2. From the list, select the command you want to reissue.

Threads

Classes that are both active and reactive run on their own threads. In animated applications with multiple threads, the thread view enables you to control the execution of threads. The animator always maintains one thread in focus. All thread-related operations are performed with respect to the focus thread:

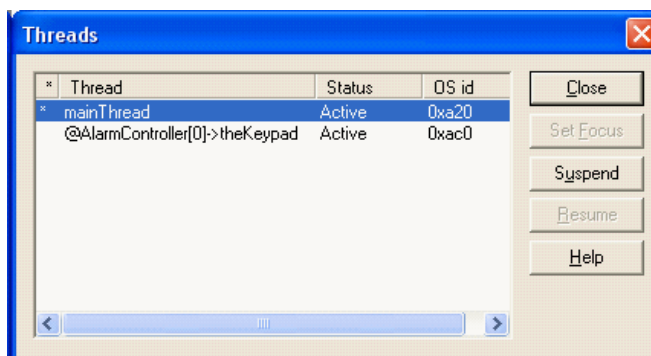
- ♦ The call stack view displays the call stack of the focus thread (see [Call Stack](#)).
- ♦ The event queue view displays the event queue of the focus thread (see [Event Queue](#)).
- ♦ The **Go** buttons on the **Animation** toolbar affect the execution only of the focus thread.

A **Go Step** advances the focus thread one step. While the focus thread is executing, all other threads execute as much as they can until the moment the focus thread finishes executing this step. It is impossible to predict how far non-focus threads will advance, because their behavior depends on how the operating system scheduler behaves during run time.

Thread View

To start the thread view, click the **Thread** tool.

The Threads window opens, listing all threads that currently exist in the model, their status (Active or Suspended), and their operating system IDs (in hex).



Setting the Thread Focus

The focus thread has an asterisk in the first column. In the figure, the main thread has focus. This means that the call stack and event queue views will display information for this thread only.

1. Select a thread that does not currently have focus. The **Set Focus** button becomes active.
2. Click **Set Focus**.

To suspend an active thread, select an active thread and click **Suspend**.

Note

The Animator cannot advance the application if the focus thread is suspended.

To resume a suspended thread, select a suspended thread and click **Resume**.

Names of Threads

The first thread of your application is called the `mainThread`. The `mainThread` is also the system thread, and objects that have sequential concurrency run on this thread.

Threads associated with active objects are named according to their object names:

- ♦ Threads associated with active objects are denoted by `@objectName`. For example, if A is an active class, `@A[0]` is the name of a thread on which an instance of A might run.
- ♦ Threads associated with active objects that are targets of relations are denoted by `@objectName->roleName`. For example, if A is an active class that is related to B as `itsA`, `@B[3]->itsA` is the name of a thread on which an instance of A might run.

You can register external threads that you have manually created (using an operating system API call rather than the Rational Rhapsody `OMThread` wrapper) by assigning your own names to them. The registering of an external thread introduces it to Rational Rhapsody and adds it to the list of threads associated with active objects displayed in the Threads window.

During the course of execution, additional external threads can appear that interact with Rational Rhapsody-defined objects. These external threads are denoted by an ID that the operating system automatically assigns to them.

Notes on Multiple Threads

Go commands speak explicitly to the focus thread and send an implicit **Go** to all other threads. For example, in a multithreaded environment with three threads named `@T1`, `@T2`, `@T3`, and `@T2` having focus, a **Go Step** command would advance `@T2` a single step. During this time, threads `@T1` and `@T3` might advance one or more steps depending on the scheduling policy of the underlying operating system. In any case, when control returns to you, all three threads have executed a whole number of steps (execution does not stop in the middle of a step).

Only *active* (not suspended) threads are advanced in a **Go** command. If the focus thread is suspended, the execution does not advance and you are prompted to either set the focus to another thread or resume the focus thread. If the focus thread dies during a **Go Step**, **Go Event**, or **Go Idle** command, the application immediately stops.

Active Thread Properties

For the VxWorks and pSOSystem environments, you can assign a name to an active thread by modifying the `CG::Class::ActiveThreadName` property for the class. The default value of this property is the empty string. The active thread name is used only by the external source debugger, (for example, the Tornado debugger), and is not the same as the thread name that is used for Rational Rhapsody animation.

For all environments, you can set the thread priority by modifying the `ActiveThreadPriority` property for the class. The default priority for all threads is taken from the `DefaultThreadPriority` static class member of the `OMOSThread` framework class (defined in `Share\oxf\os.h`).

For all environments, you can set the initial size of the thread stack by modifying the `ActiveStackSize` property for the class. This property helps provide support for static memory architectures. The default size for thread stacks is taken from the `DefaultStackSize` static class member of the `OMOSThread` framework class.

Creating Breakpoints

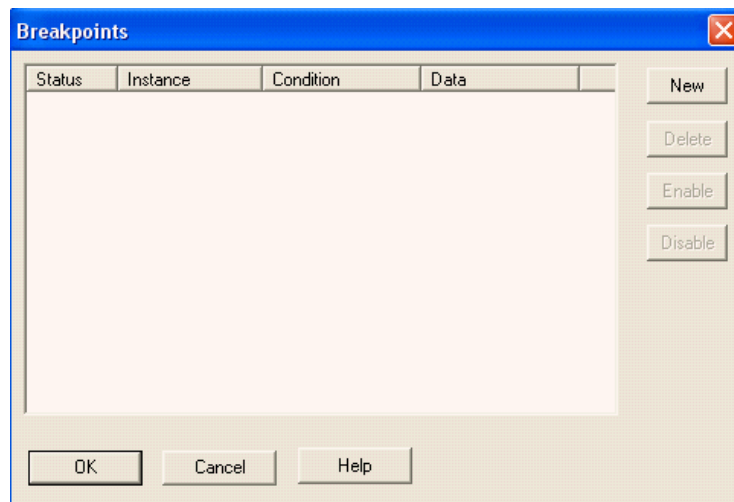
When a breakpoint is encountered in any thread, the animator switches control from the application to you. The last thread to reach a breakpoint becomes the focus thread.

You can control breakpoints by either issuing breakpoint commands in the Animation Command bar or using the Breakpoints tool.

1. Click the **Command Prompt** tool in the **Animation** toolbar.
2. In the Animation Command bar, enter **Show #Breakpoints** (not case-sensitive).

Note: Alternatively, you can control breakpoints using the Breakpoints window. To activate this window, click the **Breakpoints** tool in the **Animation** toolbar.

The Breakpoints window opens, as shown in the following figure:

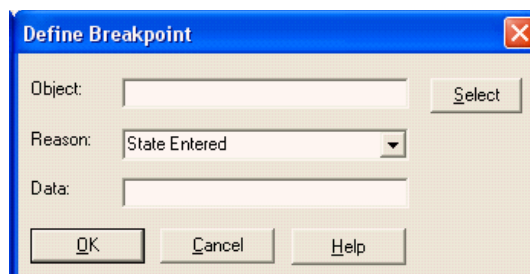


Using this window, you can define, delete, enable, and disable breakpoints.

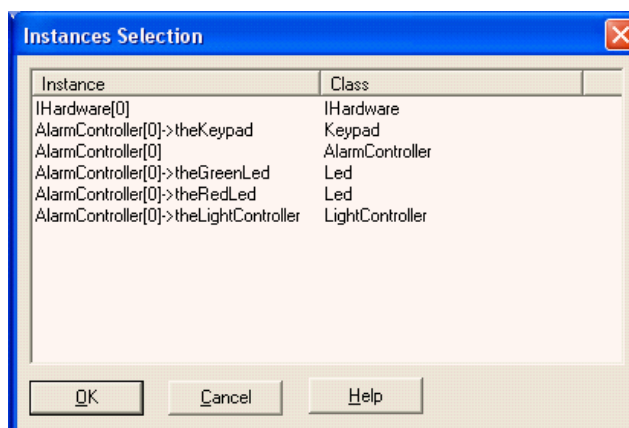
Defining Breakpoints

To define a new breakpoint:

1. In the Breakpoints window, click **New**. The Define Breakpoint window opens.



2. Click **Select** to select the object for which you want to define the breakpoint, or type the name of the instance directly into the **Object** field. The Instances Selection window opens.



3. Select the instance for which you want to define the breakpoint from the list, then click **OK**.

The Instances Selection window is dismissed and the selected object displays in the Object box of the Define Breakpoint window.

Note: In general, entering a class name for the object causes the breakpoint to operate on any instance of the class, whereas entering an instance name causes the breakpoint to operate on a particular instance.

4. Click the down arrow to the right of the **Reason** box to view a list of possible reasons for the breakpoint. Select the appropriate reason.

Some reasons might require additional data. For example, if you want to regain control when an object enters a particular state, you must provide the state name. If a state name is not provided, the break occurs when the object enters any state.

The following table shows the possible reasons for breakpoints and what, if any, optional data you can provide for each breakpoint.

Reason for Break	Object	Data	Description
Instance Created	Class	None	Break when any instance of the class is created.
Instance Deleted	Class or instance	None	Break when an instance of the class is deleted.
Termination	Class or instance	None	Break when an instance reaches a termination connector in its statechart.
State Entered	Class or instance	State name	Break when an instance enters a state.
State Exited	Class or instance	State name	Break when an instance exits a state.
State	Class or instance	State name	Break when an instance: • Enters a state • Exits a state
Relation Connected	Class or instance	Relation name	Break when a new instance is connected to a relation.
Relation Disconnected	Class or instance	Relation name	Break when an instance is removed from a relation.
Relation Cleared	Class or instance	Relation name	Break when a relation is cleared for an instance.
Relation	Class or instance	Relation name	Break when: • A new instance is connected to a relation. • An instance is deleted from a relation. • A relation is cleared for an instance.

Reason for Break	Object	Data	Description
Attribute	Instance	None	Break when any attribute of the instance changes value. A copy of the attribute values is stored and current values are compared to this copy. When a break occurs, the copy is updated with the latest values.
Got Control	Class or instance	None	Break when an instance gets control by: • Starting to execute one of its user-defined operations • Responding to an event • Regaining control after an operation that the instance has called on another object finishes executing
Lost Control	Class or instance	None	Break when an instance loses control by: • Finishing execution of one of its operations • Finishing a reaction to an event • Calling an operation of another object
Operation	Class or instance	Operation name	Break when an instance starts executing a user-defined operation.
Operation Returned	Class or instance	Operation name	Break when an instance returns from executing a user-defined operation.
Event Sent	Class or instance	Event name	Break when an instance sends an event.
Event Received	Class or instance	Event name	Break when an instance receives an event.

Enabling and Disabling Breakpoints

The first column of the breakpoints list shows the status of all breakpoints. By default, new breakpoints are available. The active breakpoints are at the top of the list, whereas disabled breakpoints are at the bottom.

To disable a breakpoint, select an enabled breakpoint and click **Disable**. The breakpoint is disabled and moved to the bottom of the list.

To enable a breakpoint, select a disabled breakpoint and click **Enable**. The breakpoint is enabled and moved below the last enabled breakpoint and above the first disabled breakpoint in the list.

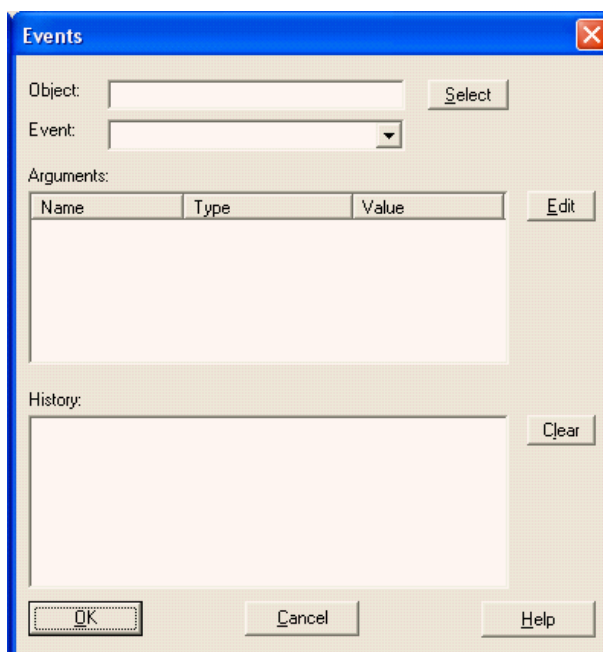
Deleting Breakpoints

To delete a breakpoint, select the breakpoint and click **Delete**. The breakpoint is removed from the list.

Event Generator

You can generate events to inject into the model using either the **Event Generator** tool or the Animation Command bar (see [Generating Events Using the Animation Command Bar](#)).

To generate events using the Event Generator, click the **Event Generator** tool in the **Animation** toolbar. The Events window opens, as shown in the following figure:



This window enables you to select an object as the target of the event and define the event you want to send to that object. If events have previously been generated during the same animation session, those events appear in the Events History list and you can simply select an event from the history list.

Generating Events

1. In the Events window, click **Select**. The Instances Selection window opens (see [Defining Breakpoints](#)).
2. Select an instance from the list, then click **OK**.
3. Click the down-arrow to right of the **Event** box to display a list of events that the object is capable of receiving and select an event from the list.

If the event takes arguments, they are displayed in the arguments list.

4. You must assign actual values to arguments to successfully generate events with arguments. To assign a value to an argument, select the argument from the **Arguments** list, then click **Edit**. The Argument Value window opens.
5. Enter a value for the argument, then click **OK**. The value is displayed next to the argument in the Events window.
6. Click **OK**.

Note that the Events window stays open after sending the event.

Events History List

Successfully generated events are stored in an Events History list that is displayed in the Events window.

Every time you save a project, events are stored in the history list, along with active and inactive breakpoints, to a file named <projectname>.ehl in the project directory. The following example shows an .ehl file:

```
[Events]
A[0]->GEN(evCount())`Default::A
A[0]->GEN(evOn())`A
A[0]->GEN(evDeep())`Default::A
A[0]->GEN(evOn())`Default::A
[Active Breakpoints]
[Inactive Breakpoints]
```

Resending Events

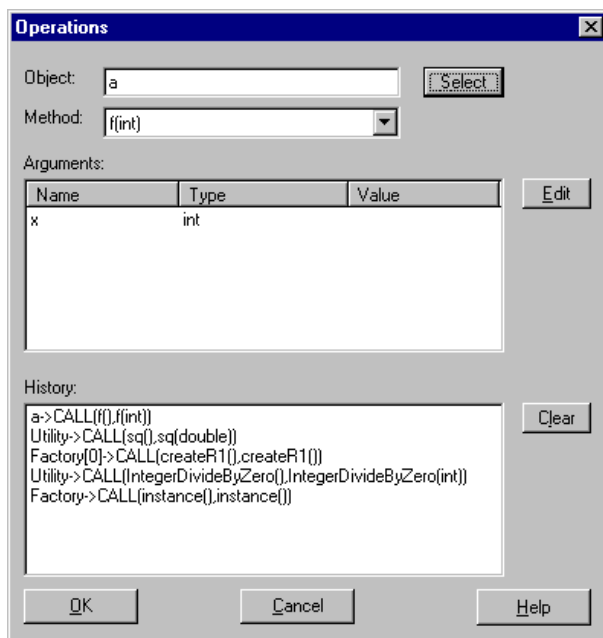
To resend an event from the events history list:

1. In the Events window, select an event from the Events History list.
2. Click **OK**. The event is entered into the Event Queue.

To clear the events history list, click **Clear**.

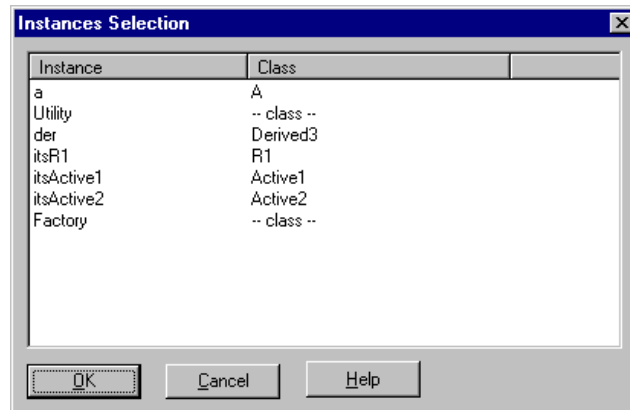
Calling Animation Operations

Using Rational Rhapsody Developer for C++ and C, you can start operation calls during animation and tracing to validate parts of the design model. To call an operation, it must be instrumented; by default, instrumentation is set to None (you cannot call operations). To enable operation calls, either set the property `<lang>_CG::Operation::AnimAllowInvocation` or use the Advanced Instrumentation Settings window (opened by clicking the **Advanced** button on the **Settings** tab for the configuration). To call an operation, click the **Call operations** tool in the **Animation** toolbar.



This window contains the following fields:

- ◆ **Object** specifies the object (or class) that contains the method you want to start. Click **Select** to open the Instances Selection window, as shown in the following figure:



This window displays the classes and instances that have operations that you can call.

Select the appropriate instance, then click **OK**. You return to the Operations window.

- ◆ **Method** specifies the operation to call.

If you selected a class in the Instances Selection window, only static methods instrumented for operation calls are listed in the Operation window. The Operation window displays all the available operations for invocation, including the ones specified in the base classes.

- ◆ **Arguments** displays the arguments used by the specified operation. If necessary, click **Edit** to modify the arguments for the operation.

To start an operation, all the arguments must be animated. If the data type is complex, you must specify the unserialization routine for the type and force its instrumentation.

- ◆ **History** shows the history of all the operations called in the animation session. As with events, the history of operation calls is stored in a log file.

Click **Clear** to delete the history.

Once you have set the appropriate values, click **OK** to launch the method call. The results are displayed in the output window (animation) or console (tracing).

Note that the Operations window stays open after sending the operation.

Scheduling and Threading Issues

The method is launched by the application thread currently in focus. The workflow is as follows:

1. If needed, set the focus to be the thread that should execute the operation.
2. Send the command to the animator. If several commands are sent to the same thread, the application will execute them one after the other in the calling order. In addition, you can switch focus threads and send start requests to different threads to simulate concurrent calls.
3. Continue execution of the application (by one of the “go” commands).
4. The operation is displayed in the callstack of the thread and relevant sequence diagrams, and is carried out.
5. Once the operation returns, its return value is displayed in the output window (animation) or in the console (tracing).

Note

The operation call is *synchronous* where the thread executes the operation, then returns to its last previous position in the interrupted control flow.

Using Partial Animation

By default, operations are not instrumented for calls. To enable operation calls, set the property `<lang>_CG::Operation::AnimAllowInvocation`. See the property definitions in the Properties tab for more information.

Note that if an operation is not animated (either the `Animate` property for the operation is set to `FALSE`, or the class of the operation is not in the configuration's instrumentation scope), instrumentation for the invocation is disabled.

Scheduling and Threading Restrictions

Note the following restrictions and limitations:

- ◆ Only an animated Rational Rhapsody (non-foreign) thread can call an operation.
- ◆ You can launch an operation only if all its argument types are animated and have unserialization routines (as with events).
- ◆ The following aspects are not supported:
 - Constructor and destructor invocations
 - Global functions

- Overloaded operators (such as ++, <<, and so on)
 - Templates
- ♦ You must specify all arguments (default arguments are not supported).
- ♦ Once the operation actually starts, the sent request is shown (not the message) and the operation is shown in the callstack.

Animation Modes

Silent mode enables you to perform design-level debugging and display animation information only at breakpoints. In contrast, Watch mode enables you to continually update animation information in normal step-by-step operation via the **Go** buttons. You can toggle between the two modes at any break, or when the executable is idle, using the **Silent/Watch** tool in the **Animation** toolbar.

Note

Watch mode is the default animation mode. If you are in Watch mode, the text “Watch - Display Continuous Update” is displayed in the tooltip when you move the cursor over the Silent/Watch tool. If you are in Silent mode, the text “Display on Breakpoint Only” is displayed.

Silent Mode

In Silent mode (also known as Update-on-Break mode), the model runs at near production speeds and the animated views are not updated until you hit a breakpoint. Execution speeds can be up to 100 times faster in Silent mode than in Watch mode.

To activate Silent mode:

1. When the model is idle and in Watch mode, click the **Silent** tool. The animated views are immediately updated, but the event queue is not.
2. To update the event queue after a break, click the **Command Prompt** tool.
3. In the Animation Command bar, type refresh and press **Enter**.

Alternatively, you can use **View > Refresh** (or press **F5**). The next time the model reaches a breakpoint, the event queue immediately updates.

Watch Mode

Watch mode is the normal mode of operation in which all animated views and the event queue are continually updated with each **Go**. Watch mode is preferable for unit testing or pinpoint debugging.

To activate Watch mode, click the **Watch** tool when the model is idle and in Silent mode.

Viewing the Model

This section describes how to view and inspect components of a model during animation. The ability to inspect an executable model is a key feature of animation and a major debugging aid during model design.

You can watch the active execution for an application in any of the following views:

- ♦ Output window
- ♦ Call Stack window
- ♦ Event Queue window
- ♦ Animated browser
- ♦ Animated sequence diagrams
- ♦ Animated statecharts
- ♦ Animated activity diagrams
- ♦ Thread view (multithreaded applications only)

During animation, you can access:

- ♦ The model as a whole
- ♦ Objects that make up the model and relations between them
- ♦ Internal information about specific objects

Accordingly, you are provided with three types of views:

- ♦ Application-wide status views, available only in animation:
 - Call stack view
 - Event queue view
 - Threads view
- ♦ Multiobject views provided by animated versions of multiobject design tools:
 - Animated sequence diagrams, which depict messages actually passed between instances during the execution of the application.
 - The animated browser, which enables you to inspect instances currently alive in the application
- ♦ Object-specific view provided by the animated version of single-object design tools:
 - Animated statecharts, which describe the current states and latest transitions of the object
 - Animated activity diagrams

Call Stack

The call stack view describes the current stack of calls for the focus thread. To open the call stack view, select **View > Call Stack**.

- ♦ `startBehavior` initiates the behavior of a reactive object
- ♦ `takeEvent` initiates the response of a reactive object to the reception of events

For C++ and Java, each line in the call stack view depicts a single function using the following member pointer notation:

`<instance>-><operation>`

If the operation does not belong to any particular instance (for example, top-level function calls) or is a constructor, only the operation name is displayed. Operations are added and removed from the stack in LIFO (last-in, first-out) order. The most recent operation is always pushed onto and popped off the top of the stack.

Event Queue

The event queue view describes the current state of the event queue for the focus thread. To open the event queue view, select **View > Event Queue**.

Each line in the display depicts a single event in the format:

`<name of event destination> -> <event name> (<parameters>=
value{,<parameter>=value})`

For example:

```
A[1]->Start(priority=3)
B[3]->NewGame(score = (5,0), time = 3)
```

The top-most event is the next to be dispatched.

Animated Browser

During animation, the browser displays *instances* (objects instantiated) for each class participating in the execution. Typically, instances are deep blue in color. However, an instance that currently has control (is currently executing) is light blue. Selecting an instance in the animated browser displays a window that shows:

- ♦ A list of its attributes for the instance and their current values.
- ♦ A list of its relations. Selecting a relation displays a list of the items in this relation.

From the browser, you can open an animated statechart for an instance (see [Animating Statecharts](#)).

Animated Sequence Diagrams

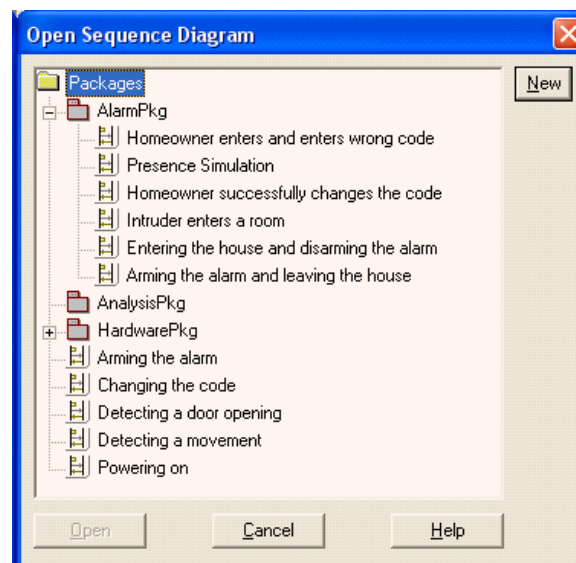
A sequence diagram displays the passing of messages between instances that participate in the chart. Sequence diagrams are a concise and popular way to represent the communication between interacting objects.

Opening Animated Sequence Diagrams

To open an animated sequence diagram (ASD), use either of the following methods.

- ♦ With a nonanimated sequence diagram already open in Rational Rhapsody, start the model running. When the model starts executing, an animated version of the currently open sequence diagram opens along with the original, nonanimated version.
- ♦ Select **Tools > Animated Sequence Diagram**.

The Open Sequence Diagram window opens (as shown in the following figure), listing the nonanimated sequence diagrams that currently exist in the model. Select a nonanimated sequence diagram from the list. The animated version opens in a new window.



Adding and Deleting Instance Lines

To add an instance line to an ASD, create an instance line using the **Instance Line** tool and name the instance, or drag an instance from the browser.

Each line has a label. If the label refers to an instance that currently exists in the executing model, the line is connected to that object.

To delete an instance line, do one of the following actions:

- ♦ Click the **Delete** button.
- ♦ Press **Ctrl+Delete**.

Auto-creating Animated Instances

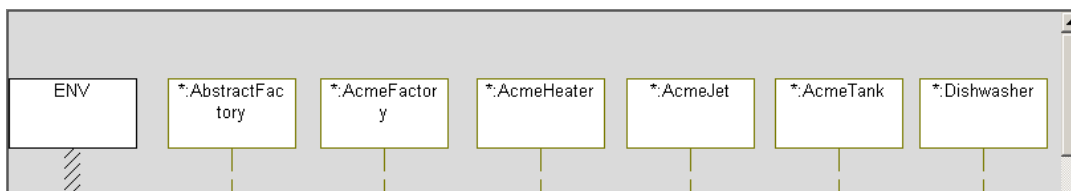
You can make it so that animated instance lines on sequence diagrams are auto-created so that you can see the run-time instance appear in the animated sequence diagram when they are actually created. (Typically during an animation session, you have to drag the created instance from the Rational Rhapsody browser onto the animated sequence diagram to see the operation of that instance getting called.) However, with added notation to an instance line name on a sequence diagram, you can have Rational Rhapsody auto-create the animated instances when you run animation. This capability means that you can mark a specific class to auto-create any sequence diagram instances at run time on the animated sequence diagram.

To auto-create animated instances for a sequence diagram:

1. Make sure you have the active configuration set for Animation. See [Setting the Instrumentation Mode](#).
2. Create a sequence diagram and make sure it is open (or open a current sequence diagram for which you want to auto-create animated instances).

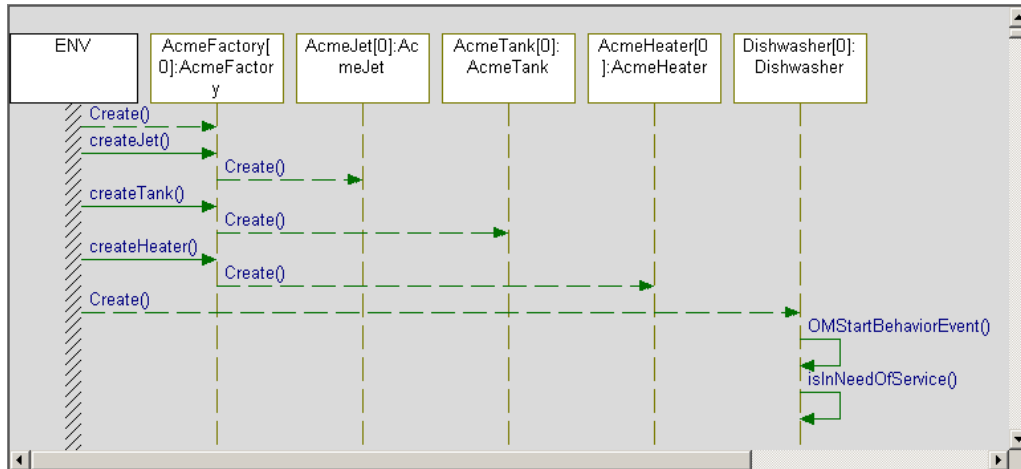
Note: You can set the `SequenceDiagram::General::AutoLaunchAnimation` property to `Always` to make the diagram open automatically when animation starts.

3. Depending on what you did in the previous step:
 - From the Rational Rhapsody browser, drag a class that you want to auto-create instances for on your sequence diagram and add an asterisk (*) to the beginning of the name. For example: `*:Dishwasher`. Or,
 - On the diagram, change the name of an instance by adding an asterisk (*) to the beginning of the name. To do this, click the name to focus the pointer on it. Once the name is highlighted, use your keyboard arrow keys or the mouse to position your mouse pointer to the beginning of the name and add * to it. For example: `*:Dishwasher`, as shown in the following figure:



4. On the **Code** toolbar, click the GMR button  generate, make, and run your model.
5. On the **Animation** toolbar, click the Go button  to start the animation session.

6. Notice that Rational Rhapsody creates an animated sequence diagram that has all auto-generated instances of type `<class_name>`, as shown in the following figure:



Limitations

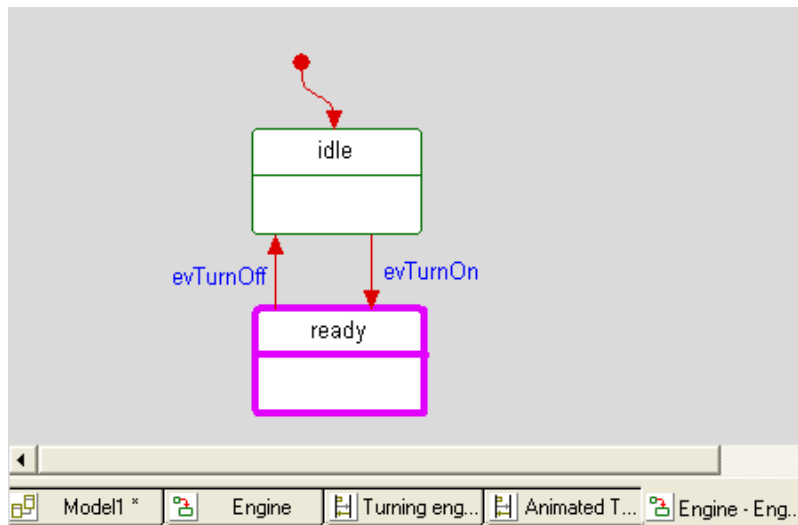
Note the following limitations:

- ◆ There is no auto-creation of derived classes.
- ◆ This feature is unavailable for Rational Rhapsody in Ada.

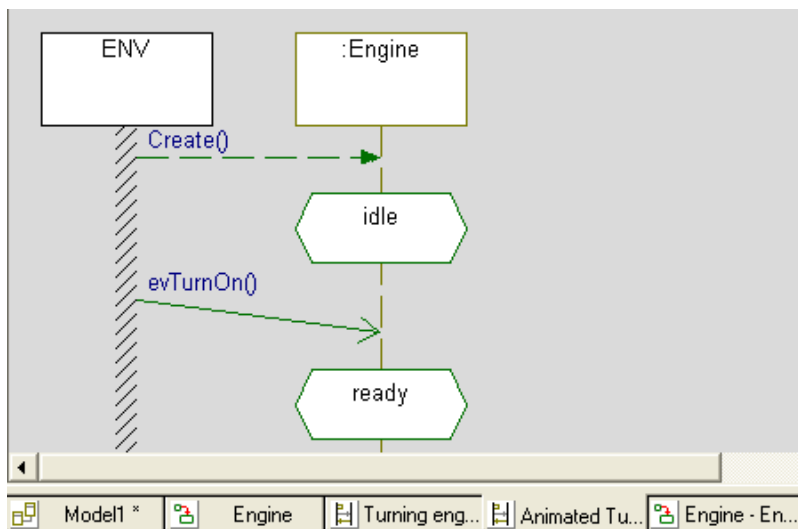
Showing State Transitions in Animated Sequence Diagrams

An animated sequence diagram can cross reference an animated statechart by showing the event of an object entering a state. In Rational Rhapsody, the Condition Mark indicates that an object is in a certain condition or state at a particular point in a sequence. For more information about condition marks, see [Creating a condition mark](#).

The following figure shows an animated statechart:



The following figure shows the animated sequence diagram. Notice the transition states (for example, **idle**).



Display Considerations

The following display considerations apply for showing state transitions in animated sequence diagrams:

- ◆ States are displayed in the order of their notifications.
- ◆ When an inner state changes, all of its and-states are listed again, even if unchanged.
- ◆ When re-entering the same state in a self loop, the entry displays again.
- ◆ And-states are listed with a pipeline character (|) in-between.
- ◆ In the case of sub-states, only the most inner current state is listed.
- ◆ The `SequenceDiagram::General::ShowAnimStateMark` property determines whether or not state transitions are displayed in an animated sequence diagram. Its default is `Checked`.

State Transition Limitations

The following limitations apply for showing state transitions in animated sequence diagrams:

- ◆ Condition marks are not realized with model states.
- ◆ The `DiffMerge` tool and `Sequence Diagram Compare` tool do not support condition marks.

The System Border

The system border, if present, is connected to all instances participating in the execution that do not have a special line of their own. In other words, a message is drawn between two instance lines if both the sending and receiving instances have instance lines in the ASD.

If either the sending or receiving instance does not have a line, messages can be displayed between the instance that has a line and the system border:

- ♦ If the ASD includes a system border, the message is drawn between the instance that has a line and the system border.
- ♦ If the ASD does not include a system border, the message is not displayed in the diagram. In other words, removing the system border prevents the display of messages to or from instances that are not explicitly part of the execution sequence.

Messages

ASDs automatically create the following message arrows while the model is executing:

- ♦ Operation calls
- ♦ Self-operation calls (from an instance to itself)
- ♦ Events sent, but not yet received
- ♦ Events sent and received
- ♦ Timeouts sent (matured), but not yet received
- ♦ Timeouts sent and received
- ♦ Constructors
- ♦ Destructors

The y-axis of a sequence diagram indicates both flow of time and steps. No scale is given on this axis, but synchronization is maintained.

When you remove an instance line, all messages sent to and from the instance are also removed. Future arrows relating to the removed instance will be associated with the system border, if the diagram has one.

When you add an instance line, messages to or from the newly added instance are displayed only from the moment the line is added. Messages sent or received by an instance before it was added to the diagram are associated with the system border.

Note

Do not create more than one line referring to the same instance. Otherwise, message arrows will connect to only one of these instance lines in an arbitrary manner.

When you quit animation, you can either save or delete animated sequence diagrams that have been generated by the execution. Saving them creates a record of the execution.

Limiting Message Display in Animated Sequence Diagrams

When running an animation of a sequence diagram containing a very large number of messages, your system might run low on virtual memory. You can use the `SequenceDiagram::General::MaxNumberOfAnimMessages` and `SequenceDiagram::General::OnReachedMaxAnimMessages` properties to prevent such problems by limiting the number of messages displayed at any one time.

Use the `MaxNumberOfAnimMessages` property to specify the maximum number of messages that should be displayed in the sequence diagram at any one time during animation.

Use the `OnReachedMaxAnimMessages` property to determine how Rational Rhapsody should behave when the maximum number of messages has been reached. The property can take the following values:

- ◆ Stop where Rational Rhapsody stops displaying animated messages in the diagram after the maximum number has been reached.
- ◆ `KeepLast` where after the maximum number of messages specified has been reached, Rational Rhapsody erases the first messages displayed. It will continue erasing displayed messages in this manner so that the number of messages displayed on the diagram at any one time does not exceed the maximum specified.

Suppressing Animated Sequence Diagram Messages

To control the appearance of Create, Destroy, Timeout, and Cancel Timeout messages in an animated sequence diagram, use the following properties in `SequenceDiagram::General`:

- ◆ `ShowAnimCreateArrow`. This property specifies whether to show Create messages in animated sequence diagrams.
- ◆ `ShowAnimDataFlowArrow`. This property specifies whether to show dataflow messages in animated sequence diagrams.
- ◆ `ShowAnimDestroyArrow`. This property specifies whether to show Destroy messages in animated sequence diagrams.
- ◆ `ShowAnimTimeoutArrow`. This property specifies whether to show Timeout messages in animated sequence diagrams.
- ◆ `ShowAnimCancelTimeoutArrow`. This property specifies whether to show Cancel Timeout messages in animated sequence diagrams.

By default these properties are set to `Checked` (so that these messages appear in animated sequence diagrams). To suppress these messages, set the above properties to `Cleared`.

Note that the sequence diagram cloned during animation will use the above properties also.

Dataflows

Rational Rhapsody animation also uses dataflow arrow notation to represent data flow between flowports. For more information about dataflows and flowports, see [Creating a dataflow](#) and [Flow ports](#).

Animating Return Values

To learn how to instrument return types so that you can see them as reply messages on animated sequence diagrams, see [Animation of the return value for an operation](#).

Animating Statecharts

To open an animated statechart, right-click an instance in the browser and select **Open Instance Statechart**.

Alternatively, you can:

1. Select **Tools > Animated Statechart**. The Open Animated Statechart window opens.
2. Select an existing instance from the list, then click **OK**.

Note that animated statecharts operate in *full behavioral steps*. This means that you see all of the final effects of a behavioral step at once (not one-by-one as they occur).

Animation Highlighting

You can change how animation is displayed by right-clicking the project in the browser and selecting **Format**.

The types used for animation highlighting are as follows:

- ♦ `AnimatedTransition` specifies how animated transitions are displayed. By default, an animated transition is displayed using olive, 3-point, solid lines.
- ♦ `AnimatedInState` specifies how an In state is displayed. By default, an In state is displayed using magenta, 3-point, solid lines.
- ♦ `AnimatedPrevState` specifies how the previous state is displayed. By default, the previous state is displayed using olive, 3-point, solid lines.

For detailed instructions on using the Format window, see [Changing the format of a metaclass](#).

Instance Names

The building blocks of a model are its classes and relations. The building blocks of the execution of a model are instances of these classes and relations, which are created during execution. This section describes how the animator names these instances.

Names of Class Instances

During execution, the instances of a class A are referred to as A[0], A[1], A[2], and so on. The name A[0] is given to the first instance of class A, the name A[1] to the second, and so on.

An instance gets a name only after its construction is completed. An item whose chain of constructors has started but not yet completed is referred to as *in construction*.

An instance retains its name only until its destruction. An item whose chain of destructors has started but not yet completed is referred to as *in destruction*.

An instance that no longer exists is referred to as *non-existent*. This can happen if an instance was deleted, but some other instance still points to it via an attribute or relation.

An instance retains its name, unchanged, during its entire lifetime. For example, an instance assigned the name A[5] at its creation continues to be called A[5] even if instances A[0] to A[4] no longer exist.

Names of Component Instances

Instances of a component use the following naming scheme, where the composite is the whole and the component is the part:

`whole[n] -> part[m]`

For example, for a component class A inside a composite class B, the fifth instance of the component A is referred to as follows:

`B[3] -> itsA[4]`

The name B[3] -> itsA[0] is assigned to the first instance of class A as a component of B[3], itsA[1] to the second, and so on.

A similar naming scheme is used for relations. In this case, the composite (whole) is considered the source of the relation, and the component (part) is the target.

Because of the chain of construction, a component is typically created first as a class instance and only then as a component. This is reflected in the name of the instance. First, it is created as A[x] (for example, A[4]), and then as B[y] -> itsA[z] (for example, B[2] -> itsA[3]).

Navigation Expressions

Instances can be referred to by the following navigation expressions:

- ♦ If A is a class, A[#j] denotes the (j+1)th instance of the class currently in existence. For example, A[#4] can denote the fifth instance of the class. This is consistent with the C/C++ convention of calling the first element A[0].
- ♦ If A is a class, A can denote the first instance of the class. This is the same as A[#0].

You can use a class name instead of an instance name only in places where there is no ambiguity as to whether it refers to the class or its first instance. For example, A->GEN(E) generates an event E for an instance A[#0]. However, the animation command “Show A relations” displays relation information about class A and all of its instances.

- ♦ If B is a name or navigation expression that refers to an instance and that instance has a relation itsA, B->itsA denotes the first element in B’s relation with A and B->itsA[#i] denotes the (i+1)th element.

The same navigation expression can refer to different instances during the course of the execution. For example, if instances A[0] to A[5] have been created and then A[3] is deleted, the expression A[#5] refers to A[4] before the deletion and to A[5] after the deletion.

Names of Special Objects

Besides classes and instances, the animator keeps track of a few other special objects such as breakpoints, call stacks, and event queues. You can reference all these from the command prompt using tracer commands. For detailed information about tracer commands, see [Tracing](#).

Animation Scripts

You can create scripts to automate animation sequences using tracer commands. The syntax of these commands is described in detail in [Tracer Commands](#).

Note

To get a list of available scripting commands, type `help` or `?` in the Animation Command bar.

Sample Script

Scripts can use these command types:

Command Type	Command
Breakpoint	break <object> <op> <breakPointType> <data>
Call	[Object name]->CALL([operation call] [signature])
Comments	// the comment goes here
Display	• display • watch
Generate event	<instanceName>->GEN(<eventName>(<parameterName> [, <parameterName>]*)) <instanceName>->GEN(<eventName>()) <instanceName>->GEN(<eventName>)
Go	• go • go event • go idle • go step
Help	• help • ?
I/O	• input [+] <destination> • output <+/-> <destination>
Quit	quit
Resume	• resume threadName • resume #Thread threadName
Set focus	• set focus <threadName> • set focus #Thread <threadName>
Show	show <object> <interest-list>
Suspend	• suspend threadName • suspend #Thread threadName
Time stamp	timestamp <option>
Trace	trace <object> <interest-list>

The following example of a script tests the behavior of a chamber unit in the pacemaker demo:

```
//*****
// file: utChamber.txt
// description: chamber unit test script
//*****// run until we
enter the sensing state
```

```

break ut2Chamber->theChamber stateEntered sensing
go
break ut2Chamber->theChamber -stateEntered sensing

// Trace ... and capture to file utChamber.log
trace #CallStack method
trace #CallStack +timeout
output +test.log

// give several heart beats
ut2Chamber->theChamber->GEN(evHeartBeat)
go idle
ut2Chamber->theChamber->GEN(evHeartBeat)
go idle
ut2Chamber->theChamber->GEN(evHeartBeat)
go idle

// absence of a heartbeat should cause a pace
go idle
break ut2Chamber->theChamber stateEntered sensing
go
break ut2Chamber->theChamber -stateEntered sensing

// regenerate heartbeats
ut2Chamber->theChamber->GEN(evHeartBeat)
go idle
ut2Chamber->theChamber->GEN(evHeartBeat)
go idle
ut2Chamber->theChamber->GEN(evHeartBeat)
go idle

// stop logging to file
output -test.log

```

Running Scripts Automatically

To run a script automatically when an executable starts:

1. Within a component, create a file named `omanipator.cfg`.
2. Give the file the following parameters:
 - ♦ **Path** where you type “.” so the file is generated to a directory one level higher.
 - ♦ **File Type** where you select **Other**.

When animation starts, if Rational Rhapsody finds a file named `omanipator.cfg`, it executes the file. This can be useful for large projects (or those with a GUI), because a script similar to the following could be written and automatically executed:

```

//=====
// Switch off the animation before starting
watch
// Run with animation off to create all objects
go idle
// Switch animation back on
display

// Run continuously

```



```
go  
//=====
```

Black-Box Animation

Rational Rhapsody includes extended animation for sequence diagrams. Animated instance lines (classifier roles) can represent run-time objects and their internal structure (their parts) as opposed to a single, run-time object (as in previous versions). This enables you to validate higher-level sequence diagrams specified prior to the elaboration of the internal structure of the classes.

Note

By default, animated instance lines in Rational Rhapsody are mapped to single objects. To activate this feature, you must modify the property settings of the instance lines, as described in [Animation Properties](#).

You can map any instance line on a sequence diagram to one of the following items:

- ◆ A single object
- ◆ An object and its parts
- ◆ An object and all the objects derived from its reference sequence diagram

In addition, you can specify that during animation, messages sent from the instance line to itself are not displayed.

Animation Properties

Two properties support this functionality:

- ◆ `Animation::ClassifierRole::DisplayMessagesToSelf` determines whether messages-to-self are displayed during animation. The possible values are as follows:
 - None which means do not display any messages-to-self.
 - All which means to display all messages-to-self.
- ◆ `Animation::ClassifierRole::MappingPolicy` specifies how to map instance lines during animation. The possible values are as follows:
 - Smart where Rational Rhapsody decides the mapping policy.

If the instance line has a reference sequence diagram, the mapping will be equivalent to `ObjectAndDerivedFromRefSD`; otherwise, the mapping is equivalent to `ObjectAndItsParts` (which, for an object without any parts, is the same as `SingleObject`).

- `ObjectAndItsParts` where the instance line is mapped to an object and all its parts (recursively), excluding parts that are explicitly shown in the diagram.
- `SingleObject` where the instance line is mapped to a single, run-time object.

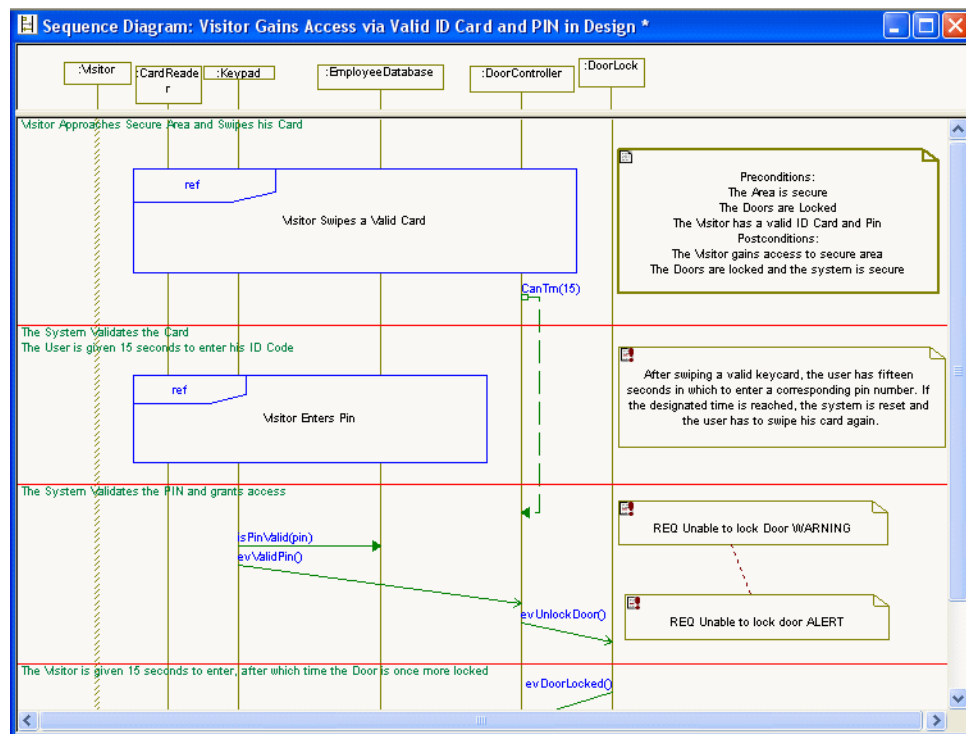
- ObjectAndDerivedFromRefSD where the instance line is mapped to an object by its role name (if it exists) and to all derived objects from the reference SDs (according to their mapping rules).

Note

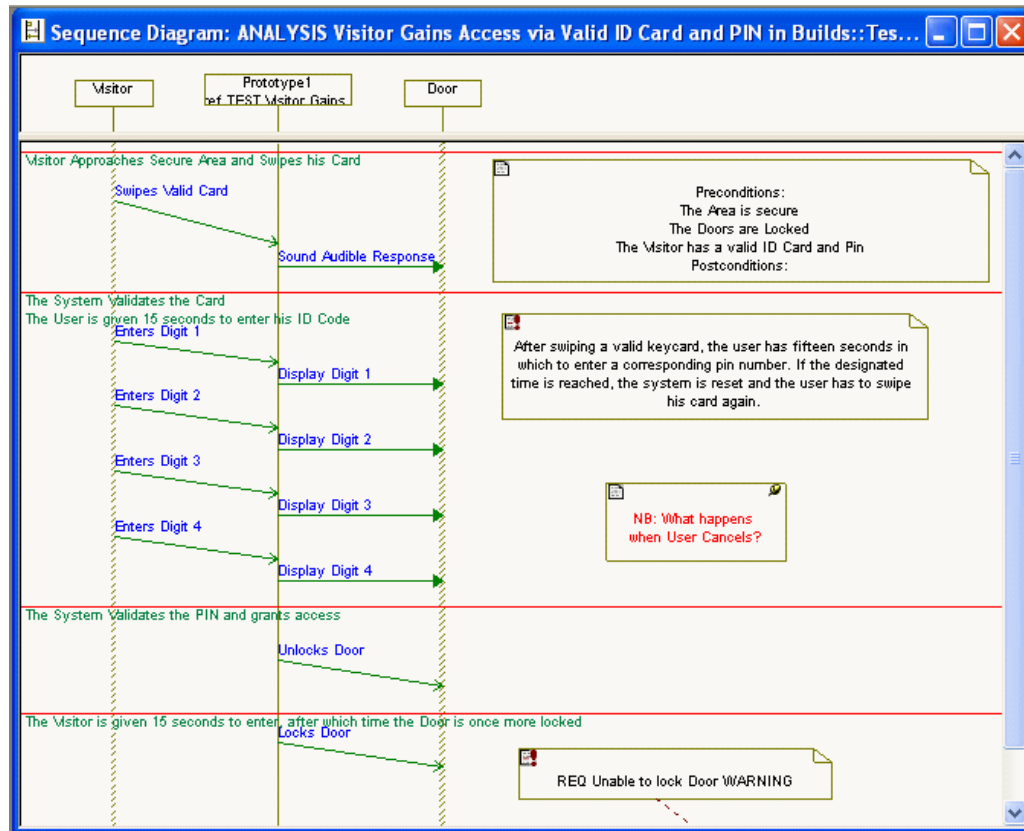
If you change the values of these properties after the sequence diagram has been initialized, the changes will not take effect until you close and reopen the animated SD.

Example

The following figure shows a design sequence diagram that describes a door with keycard access:

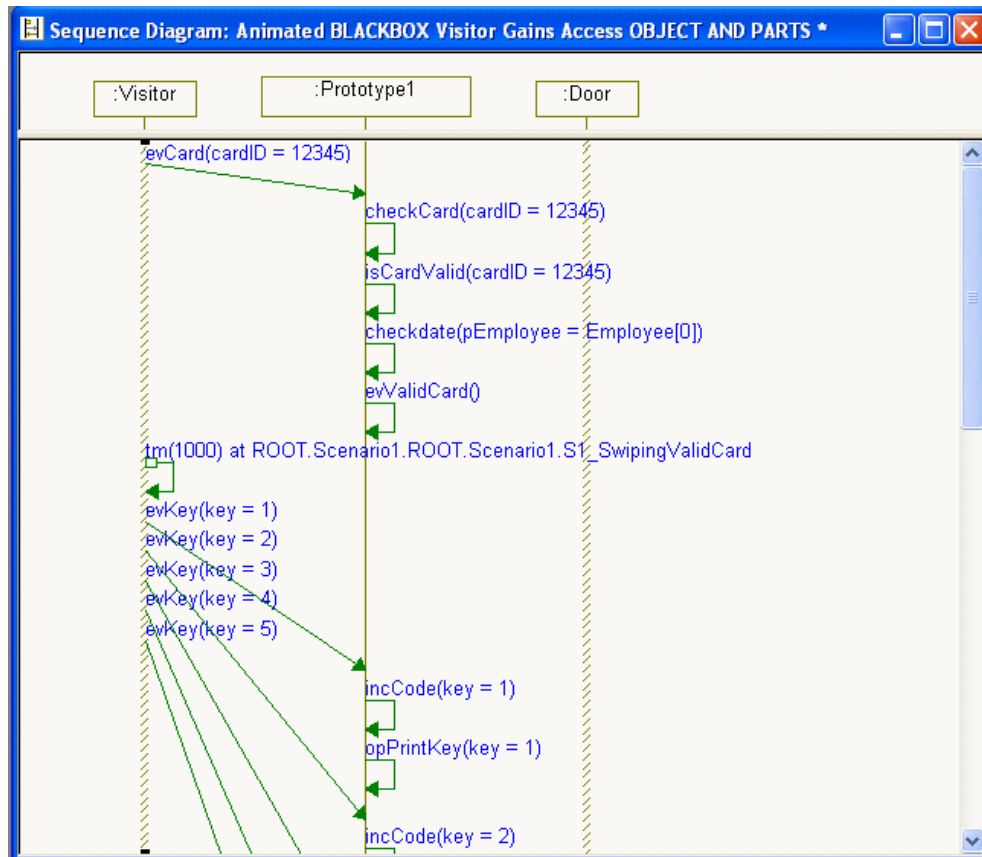


The following figure shows an SD for model analysis:

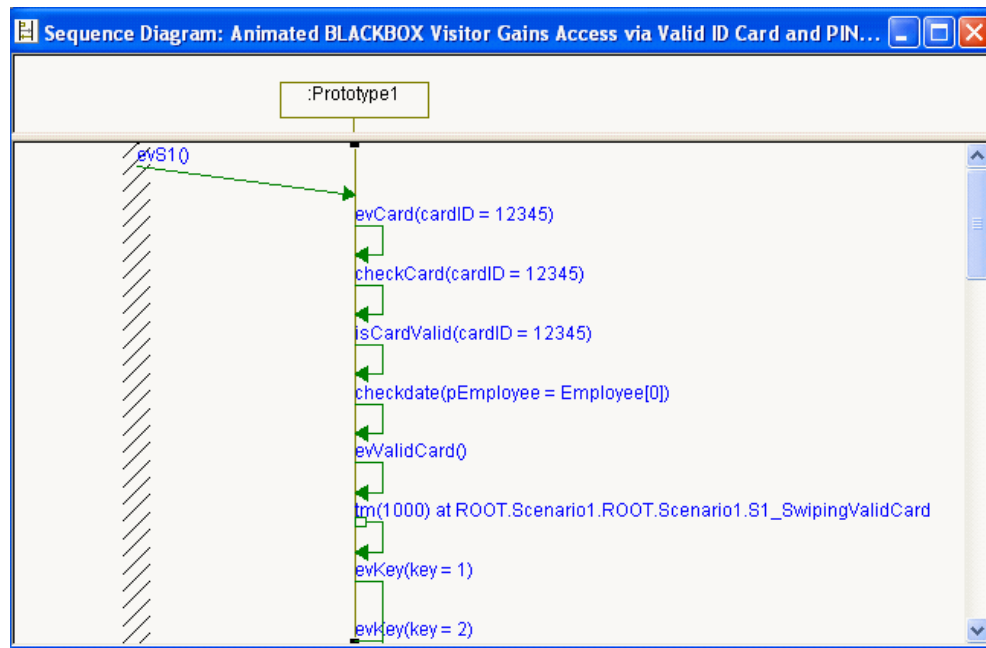


The following figures show some of the ASDs used for black-box testing of the model, and the effects of setting the animation properties to different values.

The following figure shows the resultant ASD when the Prototype1 instance line uses a mapping policy ObjectAndItsParts; the other two instance lines are set to SingleObject. Note that the Prototype1 instance line reflects the messages for the CardReader, Keypad, EmployeeDatabase, and DoorController classes.



The following figure shows another view of the model (the instance line uses `ObjectAndItsParts`). Note that messages for other parts of the model are reflected on this instance line because the `Prototype1` instance line can “see” them (because of the policy setting).



Using the Properties for Black-Box Testing

The following scenarios show the effects of the new animation properties on the ASD:

- ♦ If you are using an animated sequence diagram (ASD), which is a clone of the existing sequence diagram, the mapping relies on the property settings of the original diagram.
- ♦ While animating the model, if you create a new ASD and drag animated objects on it, the mapping is done as if the `MappingPolicy` property is set to `ObjectAndItsParts`. This mapping cannot be changed.
- ♦ While still animating the model, if you create a new, non-animated SD and set the `DisplayMessagesToSelf` and `MappingPolicy` properties, then start animating the new SD; the ASD will map its instance lines according to the property settings.

Instance Line Menu

When you right-click a instance line during animation, the menu contains the following new options:

- ♦ **Open Animated Statechart** or **Open Animated Activity Diagram** displays the appropriate animated diagram for selected object.
- ♦ **Generate Event** opens the Event window so you can generate an event. For more information, see [Event Generator](#).
- ♦ **Add Breakpoint** opens the Breakpoint window, described in [Creating Breakpoints](#).

If the instance line represents more than a single object, these commands are applied to the root object (the object that would have been mapped to the instance line in `SingleObject` mode).

Behavior and Restrictions

Note the following information:

- ♦ If any object is added explicitly to a sequence diagram, messages it receives will be shown as being received by it (rather than shown as going to the “owner”).
- ♦ A instance line that is mapped to an object and its parts will not represent parts that have their own instance lines in the same diagram.
- ♦ A single runtime occurrence such as sending a message will be viewed only once in the case of a message relayed to a part via its owner’s ports. For example, if a message is sent to a part via the port of its owner, the message to a instance line that represents the owner and its parts is shown once. In other words, the fact that the message was relayed via the owner’s port is not displayed because logically this is a single occurrence.

Animation Hints

The following sections provide some hints on some of the fine points of using the animator.

Exception Handling

Exceptions can be thrown within operations and caught within operations. However, if you want to affect some set of objects' state machines, you should catch the exception and GEN events for the appropriate objects and rethrow the exception, if necessary (for example, if you want the exception to be resolved by an operation higher in the call tree).

If Animation and Application are Out of Sync

Rational Rhapsody assumes that you are following a certain programming style, outlined in the following notes. You are not forced to follow this style, but if you choose not to, be aware that the animation might get out of sync with the model. For example, at times, it might be convenient to define a static attribute and use it directly for all class instances. Although it is not the most effective programming approach, it is a quick way to solve a number of problems. You can work this way if you prefer. However, be aware that the value of that attribute might not be updated properly during animation.

The following are standard style guidelines:

- ◆ All internals of an object are private or protected; that is, other objects cannot change the attributes, relations, or states directly for an object. They must use some operation of the object.
- ◆ States and relations can be changed only through a predefined set of mutators.
- ◆ Invoking self-triggered operations is allowed only between and-state components.

Note

On recovery, if you do not follow these guidelines and suspect that a view of a given object is inconsistent with its actual state, try closing the suspect view and reopening it. This should refresh the view and synchronize it with the actual state of the object.

Passing Complex Parameters

If you pass complex parameters (such as structs) and use animation, you must override the `>>` operator. Otherwise, Rational Rhapsody generates compilation errors.

Combining Animation Settings in the Same Model

It is possible to build libraries with animation on for part of an application, with animation off for another part, and then link both parts (the different libraries) into a single executable.

In Rational Rhapsody Developer for C, the architecture was changed from a user object being an animation object to a user object being *associated* with an animation object. As a result, the memory layout of animated and nonanimated objects is the same so, in principle, they can mix. Each class or object type is either completely instrumented or completely noninstrumented.

To create a combined application, you can link:

- ◆ Some instrumented user code
- ◆ Some noninstrumented user code
- ◆ Instrumented framework libraries (`oxfinst.lib`, `aomanim.lib`, and so on)

When some user object calls a user-defined method, the animation recognizes this, as the framework and the call stack are animated. The animation looks in a table for the animation associate of the user object (the `me` parameter in the method call). If it finds one, an animation message is sent to Rational Rhapsody with respect to this action. Otherwise, it ignores this action (the action is taken, but not animated).

Animation Feature Limitations

The animation feature cannot provide animation for classes that are nested in template classes.

Guidelines for Writing Serialization Functions

When you define a complex type, it is not understood by the Rational Rhapsody animator and therefore cannot be animated. To write serialization functions in order to animate such types, you can use the following properties:

- ◆ `<lang>_CG::Type::AnimSerializeOperation`
- ◆ `<lang>_CG::Type::AnimUnserializeOperation`

AnimSerializeOperation

The `AnimSerializeOperation` property enables you to specify the name of an external function used to animate all attributes and arguments that are of that type. Compare with [AnimUnserializeOperation](#).

Rational Rhapsody can animate (display) the values of simple types and one-dimensional arrays without any problem. To display the current values of such attributes during an animation session, open the Features window for the instance.

However, if you want to animate a more complex type, such as a date, the type must be converted to a string (`char *`) for Rational Rhapsody to display it. This is generally done by writing a global function, an *instrumentation function*, that takes one argument of the type you want to display, and returns a `char *`. You must disable animation of the instrumentation function itself (using the `Animate` and `AnimateArguments` properties for the function).

For example, you can have a type `tDate`, defined as follows:

```
typedef struct date {
    int day;
    int month;
    int year; } %s;
```

You can have an object with an attribute count of type `int`, and an attribute date of type `tDate`. The object can have an initializer with the following body:

```
me->date.month = 5;
me->date.day = 12;
me->date.year = 2000;
```

If you want to animate the date attribute, the `AnimSerializeOperation` property for date must be set to the name of a function that will convert the type `tDate` to `char *`. For example, you can set the property to a function named `showDate`. This function name must be entered without any parentheses. It must take an attribute of type `tDate` and return a `char *`. The `Animate` and `AnimateArguments` properties for the `showDate` function must be set to `Cleared`.

The implementation of the `showDate` function might be as follows:

```
showDate(tDate aDate) {  
    char* buff;  
    buff = (char*) malloc(sizeof(char) * 20);  
    sprintf(buff, "%d %d %d",  
            aDate.month, aDate.day, aDate.year);  
    return buff;  
}
```

When you run this model with animation, instances of this object will display a value of 5 12 2000 for the date attribute in the browser.

If the `showDate` function is defined in the same class that the attribute belongs to and the function is not static, the `AnimSerializeOperation` property value should be similar to the following example:

```
myReal->showDate
```

This value shows that the function is called from the `serializeAttributes` function, located in the class `OMAnimated<classname>`.

Note

The `showDate` function must allocate memory for the returned string via the `malloc/alloc/calloc` function in C, or the `new` operator in C++. Otherwise, the system will crash.

The default for this property is an empty string.

AnimUnserializeOperation

The AnimUnserializeOperation property converts a string to the value of an element (the opposite of the [AnimSerializeOperation](#) property). Unserialize functions are used for event generation or operation invocation using the **Animation** toolbar to convert the string (received from the user) to the value of the event or operation before the event generation or operation invocation.

For example, your serialization operation might look similar to the following example:

```
char* myX2String(const Rec &f)
{
    char* cS = new char[OutputStringLength];
    /* conversion from the Rec type to string */
    return (cS);
}
```

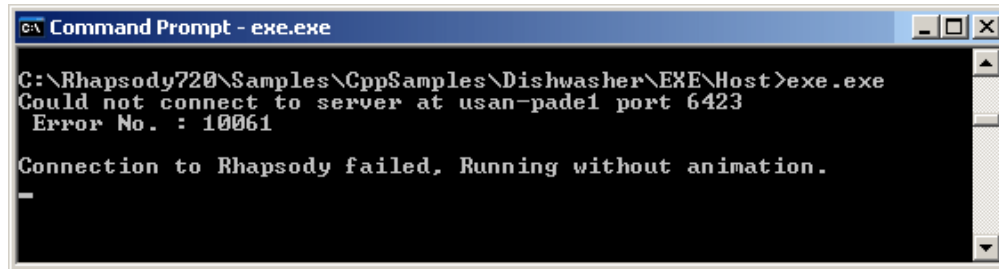
The unserialization operation would be:

```
Rec myString2X (char* C, Rec& T)
{
    T = new Trc;
    /* conversion of the string C to the Rec type */
    delete C;
    return (T);
}
```

The default for this property is an empty string.

Running an Animated Application Without Rational Rhapsody

An animated Rational Rhapsody application will always try to connect to Rational Rhapsody. However, if the application cannot connect to Rational Rhapsody, you can still run it outside of Rational Rhapsody via the command line, as shown in the following figure:



```
C:\> Command Prompt - exe.exe
C:\Rhapsody720\Samples\CppSamples\Dishwasher\EXE\Host>exe.exe
Could not connect to server at usan-pade1 port 6423
Error No. : 10061
Connection to Rhapsody failed, Running without animation.
_
```

In addition, you can use the `-noanim` flag to run the application without animation even though Rational Rhapsody is on.



```
C:\> Command Prompt - exe.exe -noamin
C:\Rhapsody720\Samples\CppSamples\Dishwasher\EXE\Host>exe.exe -noamin
_
```


Tracing

The tracer is a stand-alone text version of the animator. In the tracer, you type commands at a command prompt and receive messages detailing the status of the model as it executes.

Note

Tracing is not supported in code generated for Windows CE™.

You can use the tracer from within Rational Rhapsody, or as a stand-alone application outside of Rational Rhapsody.

Tracer Capabilities

Tracing enables you to monitor and control the application without having the Rational Rhapsody GUI in the loop by providing a text-based, console-like application. However, all tracing commands are also available in animation.

Using tracing, you can:

- ◆ Inspect and trace the status of the executing application:
 - Inspect the application via show commands.
 - Identify items to trace with trace commands.
- ◆ Set and remove breakpoints with break commands.
- ◆ Generate events or call operations with the GEN macro or CALL command (CALL applies to Rational Rhapsody Developer for C++ only).
- ◆ Return control to the Tracer for one or more steps using the go commands.

The tracer:

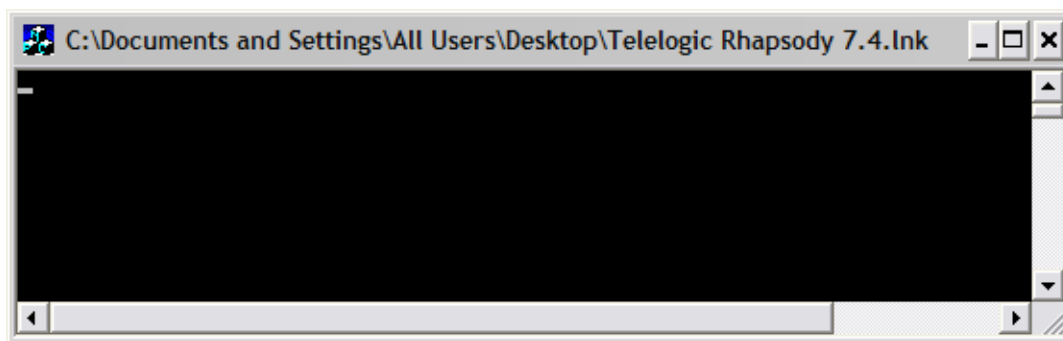
- ◆ Advances execution according to the go commands or until a breakpoint occurs.
- ◆ Displays messages describing what happens to traced objects as the model executes.

Starting a Trace Session

To prepare your application for tracing:

1. Select a configuration, right-click, and select **Features**.
2. In the Settings tab, set the Instrumentation Mode to **Tracing** and click **OK**.
3. In the browser, right-click the configuration and select **Set as Active Configuration**.
4. To generate code for the configuration, select **Code > Generate > <configuration>**. The generation messages are displayed in the Log tab of the Output window.
5. To build the component, select **Code > Build <configuration>**. Build messages are displayed in the Output window.
6. To run the component, select **Code > Run <configuration>**.

A command prompt window opens ready for you to enter a tracer command.



Controlling Tracer Operation

Use tracer commands to control the tracer's operation. The tracer reports on the status of instances as the application executes. Enter tracer commands at the command prompt when using the tracer as a stand-alone application or in the Animation Command Bar during animation. The tracer displays a variety of messages depending on the commands that you enter.

Accessing Tracer Commands

To see a brief description of tracer commands, type `help` at the command-line. See also [Tracer Commands](#).

To place comments in a tracer command, precede the comment text with two forward slashes. The portion of the line from the slashes to the end of the line is considered a comment. For example:

```
trace B[5] relations // Displays a message whenever a
                    // relation of B[5] is modified.
```

Tracer Commands and an Input File

Rather than typing commands at the command prompt, you can give the tracer commands from an input file.

By default, the tracer looks for commands in a file named `OMTracer.cfg` in the configuration directory. If the file does not exist, or the tracer has reached the end of the file, the tracer looks for commands entered at the prompt. You can specify a different input file using the `input` command (see [input](#)).

By default, the animator looks for tracing commands in the `OMAnimator.cfg` file. It is stored in the same directory as the project file.

Note

Because the tracer generates numerous messages, you might want to use the `output` command to send results to a file (see [output](#)).

Sending Commands in the Default Input File

To send commands to the tracer using the default input file:

1. Create a text file that contains each tracer command as you would type it at the command prompt, in the order that you want the commands to execute.
2. Save the file in the configuration directory under the name `OMTracer.cfg`.
3. Generate, make, and run your application using the **Tracing** instrumentation mode.

Sending Commands to the Animator in the Input File

To send tracer commands to the animator using an input file:

1. Create a text file that contains each tracer command as you would type it at the command prompt, in the order that you want the commands to execute.
2. Save the file in the project directory under the name `OMAnimator.cfg`.
3. Generate, make, and run your application with **Animation** instrumentation.

Threads in Tracing

During a tracing session, you can suspend, resume, or set focus on a thread.

Each application starts with a single thread named `mainThread`.

Whenever an instance of an active class is created, a new thread is created with it. During construction of the instance, the name of the thread is `@<in construction>`; from then on, the name is `@<instanceName>`.

When an instance of an active class is deleted, its thread dies with it and cannot be referenced anymore.

When you register an external thread, you give it a name that is used to identify it in tracer or animation. This is an advanced feature needed in special cases.

When an unregistered external thread calls on an instrumented operation, that thread is identified by the handle the operating system gives it. This happens typically if you connect your instrumented code with a GUI engine that does not use events. (This is an advanced feature needed in special cases.)

For more information, see [Notes on Multiple Threads](#).

Tracer Commands

The following sections document the tracer commands. For ease-of-use, the commands are presented in alphabetical order.

The commands are as follows:

- ◆ break
- ◆ CALL
- ◆ display
- ◆ GEN
- ◆ go
- ◆ help
- ◆ input
- ◆ LogCmd
- ◆ output
- ◆ quit
- ◆ resume
- ◆ set focus
- ◆ show
- ◆ suspend
- ◆ timestamp
- ◆ trace
- ◆ watch

break

Description

The break command enables you to add or remove a breakpoint on a given occurrence.

Syntax

```
break <object> <op> <breakPointType> <data>
```

Arguments

object

Specifies the object. This must be #All, a valid class name, or a valid instance name.

Setting a breakpoint on a class implies setting a breakpoint on all its instances and subclasses.

op

Specifies the operation. The possible values are +, -, add, or remove. The default value is add.

breakPointType

Specifies the type of breakpoint. The possible values are as follows:

- ♦ `instanceCreated` means with class only. Breaks when a new instance of this class (or a subclass of it) is created.
- ♦ `instanceDeleted` means breaks when the instance (an instance of the class) is deleted.
- ♦ `termination` means breaks when the instance (an instance of the class) reaches a termination connector. Does not break if the instance is deleted in a way other than by entering a termination connector.
- ♦ `stateEntered <state name>` means if a state name is specified, it breaks when the instance (an instance of the class) enters that state. If the state name is omitted, it breaks when instance enters any state.
- ♦ `stateExited <state name>` means if a state name is specified, it breaks when the instance (an instance of the class) exits the given state. If the state name is omitted, it breaks when instance exits any state.
- ♦ `state <state name>` means if a state name is specified, it breaks when the instance (an instance of the class) enters or exits the given state. If the state name is omitted, it breaks when instance exits or enters any state.
- ♦ `relationConnected <relation name>` means if a relation name is specified, it breaks when a new instance is connected to the given relation for this instance (an instance of this class). If the relation name is omitted, it breaks when a new instance is connected to any relation.
- ♦ `relationDisconnected <relation name>` means if a relation name is specified, it breaks when an instance is removed from the given relation for this instance (an instance of this class). If the relation name is omitted, it breaks when an instance is removed from any relation.
- ♦ `relationCleared <relation name>` means if a relation name is specified, it breaks when the given relation for this instance (an instance of this class) is cleared. If the relation name is omitted, it breaks when any relation is cleared.
- ♦ `relation <relation name>` means if a relation name is specified, it breaks when a new instance is connected to the relation, an instance is deleted from the relation, or the relation is cleared for this instance (an instance of this class). If the relation name is

omitted, it breaks when a new instance is connected to any relation, an instance is deleted from any relation, or any relation is cleared

- ◆ `attribute` means instance only. Breaks when any of the attributes of the given instance changes. When the breakpoint is set, a copy of the attribute values of the instance is stored. When any of the attribute values change with respect to this copy a break occurs. After the break, a copy of the new (modified) values is kept as the reference.
- ◆ `gotControl` means breaks when the instance (an instance of the class) gets control. This happens when the instance starts executing one of its user-defined operations, the instance responds to an event, or an operation the instance called from another object has finished and now it resumes executing.
- ◆ `lostControl` means breaks when the instance (an instance of the class) loses control; that is, either it has finished executing an operation and it now returns, it finished responding to an event, or it calls an operation of another object.
- ◆ `operation <operation name>` means if an operation name is specified, it breaks when the instance (an instance of the class) starts executing the named operation. If the operation name is omitted, it breaks when the instance starts executing any of its user-defined operations.
- ◆ `operationReturned <operation name>` means if an operation name is specified, it breaks when the instance (an instance of the class) returns from executing the named operation. If the operation name is omitted, it breaks when the instance returns from executing any of its user-defined operations.
- ◆ `eventSent <event name>` means if an event name is specified, it breaks when the instance (an instance of the class) sends the named event. If the event name is omitted, it breaks when the instance sends any event.
- ◆ `eventReceived <event name>` means if an event name is specified, it breaks when the instance (an instance of the class) receives the named event. If the event name is omitted, it breaks when the instance receives any event.
- ◆ `all` indicates all break points. This keyword can be used only to remove all breakpoints.

For example, the following command removes all breakpoints on `B[5]`:

```
break B[5] - all
```

The following command removes all breakpoints from the animation:

```
break #all - all  
data
```

Is context-dependent. See [breakPointType](#). Breakpoints that take data are shown with the data parameter in angle brackets.

Setting a breakpoint on some occurrence causes execution to stop when that occurrence happens. For example, the following command causes execution to stop when `B[2]` enters state `ROOT.S1`:

```
break B[2] stateEntered ROOT.S1
```

Saving Breakpoints

To save breakpoints, write them to a file (for example, `myBreakPoints.cfg`). After you have saved them, you can reinsert them next time you run the application by typing the following command:

```
input myBreakPoints.CFG
```

CALL

Description

The CALL command starts an operation call in animation or tracing. TestConductor can use this command to launch operations.

See [Calling Animation Operations](#) for more information on calling operations during animation.

Syntax

```
[Object name]->CALL([operation call]
[, signature (optional)])
```

Arguments

operation call

If the operation is static, this is the class name. Otherwise, it is the name of the object that performs the call. The format is as follows:

```
[method name]([list of argument values])
signature
```

Specifies the signature of the operation. This optional argument is used to distinguish between overloaded functions. For example:

```
a->CALL(f(5, "Hello"),f(int,char*))
```

Examples

```
A[0]->CALL(f(5))
```

Invokes `f(5)` on the object `A[0]`

```
A->CALL(g("Hello, World!"))
```

Invokes `g(char*)` on the object `A`

Notifications

The following table lists sample notifications.

Action	Message Format
The operation will be called when the application resumes.	Message: <CALL command> sent. For example: Message: Utility->CALL(sq(2)) sent.
The operation returned, and there is a return value.	<CALL command> returned <return value> For example: Utility->CALL(sq2)) returned 1.41421.
No matching operation is found.	Unable to perform <CALL command>, no matching operation found. For example: Unable to perform a->CALL(f()), no matching operation found.
More than one matching operation is found.	Unable to perform <CALL command>, more than a single matching operation found. For example: Unable to perform a->CALL(f(5)), more than a single matching operation found. Note that this can happen only if you use the command-line interface and do not specify the signature. If you use the window, this message will never be displayed because the window always fills in the signature.

display

Description

The display command switches the animation mode to “silent.”

Syntax

```
display
```

GEN

Description

The GEN command enables you to generate an event to an object in the executable. The command can be executed with or without parameters.

Syntax

```
<instanceName>->GEN(<eventName>(<parameterName>  
    [, <parameterName>]*))  
<instanceName>->GEN(<eventName>())  
<instanceName>->GEN(<eventName>)
```

Arguments

instanceName

Specifies the canonical name of an instance or a navigation expression.

A canonical name can be:

- ♦ The name of the class, if the class is a singleton class (for example, A)
- ♦ The name of the class followed by a subscript, if the class has multiple instances (for example, B[2])
- ♦ instanceName->CompositeRelation (for example, B[2]->itsC[3])

A navigation expression can be:

- ♦ ClassName[#multiplicity] for a non-singleton class (for example, B[#2])
- ♦ instanceName->Relation[#multiplicity] (for example, B[2]->itsC[#3])

A canonical name always refers to the same instance. A navigation expression can refer to different instances at different times. For example, B[#0] might refer to instance B[4] if instances B[0] to B[3] are deleted.

eventName

Specifies the name of the event to be generated. If the event requires parameters, include them in the GEN command.

If an event has parameters, the GEN command provides the event with the correct number of parameters and the correct types. For example, to generate an event, X where X is defined as `X(int, B*, char*)`, and B is a class defined in Rational Rhapsody, enter:

```
A[1]->GEN(X(3,B[5],"now"))
or
A[1]->GEN(X(1,NULL,"later"))
```

When the parameters of an event are not pointers to classes defined in Rational Rhapsody (for example `int`, `char*`, or `userType` (where `userType` is a user-defined type defined outside Rational Rhapsody), the tracer relies on the C++ operator `>>` (`istream&`) or the template `string2X(T& t)` to interpret the characters you type in correctly. `A[1]->GEN(Y(1))` works because the operator `>>` converts the character 1 to the integer 1, but `A[1]->GEN(Y(one))` does not work because the operator `>>` does not convert the characters “one” to an integer. Similarly, if you use a type you defined outside Rational Rhapsody, you should provide an operator `>>` operation for it if you want to generate events to it via the tracer.

go

Description

In Rational Rhapsody, go commands advance the application one or more steps. In the tracer, use go commands at the command-line. From the animator, you can type go commands at the command-line or use the **Animation** toolbar.

A go command causes the tracer to execute immediately, even in the middle of reading commands from a file. The remaining (unread) lines are used as commands the next time the tracer stops the execution and searches for commands.

If the application reaches a breakpoint, control might return to you before a go command is complete.

There are four go commands:

- ◆ go executes your application until it terminates or reaches a breakpoint
- ◆ go step executes your application for a single step (that is, until the next Rational Rhapsody-level occurrence)
- ◆ go next executes your application until the next Rational Rhapsody-level occurrence

- ♦ `go event` executes your code until the next dispatching of an event, or until the executable is idle
- ♦ `go idle` executes your application until it reaches an idle state where it is waiting for timeouts or events from GUI threads

See also [Notes on Multiple Threads](#).

Syntax

```
go
go step
go next
go event
go idle
```

help

Description

The `help` command displays a brief description of tracer commands.

Syntax

```
help
```

input

Description

The `input` command causes the tracer to read its next command from the specified destination. Once the tracer reaches the end of an input file, it looks to standard input for commands. However, specifying a “+” in the command causes the tracer to resume taking commands from the destination file after it reaches the end-of-file.

Syntax

```
input [+] <destination>
```

Arguments

`destination`

Specifies the destination, either standard input (for example, `cin`) or a file name.

Only standard input or the name of the current file can appear without the plus sign (+).

LogCmd

Description

Traces an animation session and saves the sequence of actions for reuse. Note that this command is not case-sensitive.

Syntax

```
LogCmd [+/-] <filename>
```

Arguments

filename

The name of the file to which to write the commands

Example

To write the commands to the file `myScript.txt`, use the following command:

```
LogCmd + myScript.txt
```

To stop writing commands to the file, use the following command:

```
LogCmd -
```

output

Description

The output command directs tracer output to the specified destination.

Syntax

```
output <+/-> <destination>
```

Arguments

+

A plus sign (+) adds the specific destination to a list of destinations. Output goes to all items specified on this list.

-

A minus sign (–) removes the destination from the list of destinations. Messages will no longer be sent to this destination.

destination

Specifies the destination, either standard output (for example, cout) or a file name.

If the specified file cannot be opened, the tracer displays an error message and does not add the file to the list of destinations.

The default destination list contains standard output only.

quit

Description

Ends the tracer session.

Syntax

```
quit
```

resume

Description

The `resume` command resumes the specified thread, if it has been suspended. It has no effect if the thread is already active.

Syntax

```
resume threadName
resume #Thread threadName
```

Arguments

`threadName`

Specifies the thread to resume

set focus

Description

The `set focus` command sets the specified thread to be the focus thread.

By default, tracing starts with `mainThread` in focus. The focus changes in the following cases:

- ◆ You enter a `set focus` command.
- ◆ A thread encounters a breakpoint and stops the application, in which case it becomes the current thread.
- ◆ The focus thread died (the active instance to which it belonged was deleted). In this case, the tracer set focus on one of the remaining threads. The selection is random, but if there are nonsuspended threads, one of these is selected.

Syntax

```
set focus <threadName>
set focus #Thread <threadName>
```

Arguments

`threadName`

Specifies the new focus thread

show

Description

The show command enables you to view the current status of an object. It enables you to view the status of the object by subject. Subjects include existence, attributes, methods and events.

For example, the following command displays a list of all B[5] attributes and their current values:

```
show B[5] attributes
```

Syntax

```
show <object> <interest-list>
```

Arguments

object

Specifies the object to be traced. It can be one of the following items:

- ◆ The name of a class appearing in code, such as A. The trace command is applied to all instances of class A.
- ◆ The name of an instance that currently exists in the execution, such as A[3]. You cannot refer to instances before their construction or after their destruction.
- ◆ A navigation expression, such as A[3]->itsB[2]. See [Navigation Expressions](#) for more information.
- ◆ The name of a package appearing in the code. The tracer will report on all classes in the package.
- ◆ A keyword understood by the tracer. These keywords are not case sensitive. The possible keywords are as follows:
 - #All means all classes appearing in code.
 - #Breakpoints means the list of all breakpoints.
 - #CallStack means operations currently on stack; that is, operations started but not yet terminated, including behavior operations defined on transitions.
 - #EventQueue means a queue of all pending events; that is, events sent but not yet received.
 - #Thread threadName->#CallStack means the call stack of the thread threadName. (All operations that started on this thread but have not yet terminated, including behavior operations defined on transitions).
 - #Thread threadName->#EventQueue means the queue of all pending events of the thread name threadName. (Events sent but not yet received.)

interest-list

Specifies the list of subjects, separated by a commas. The interest list determines what information about the object is reported to you.

This list is optional; if you do not enter any subjects, the tracer reports on the existence of the object only, as if you had executed the following command:

```
show <object> existence
```

The possible subjects are as follows:

existence	constructors
relations	destructors
attributes	timeouts
states	parameters
controls	subclasses
methods	threads
events	

The subject `existence` reports on the existence of the object.

The subject `subclasses` applies the trace commands to all of a class's subclasses. It is relevant only to class objects.

The following keywords can be used to define which objects to show (they are not case-sensitive):

- ♦ `#All` shows the subjects in the interest list for all classes in code.
- ♦ `#All events` displays all the events recognized by the system. This is useful if you forget the exact name of an event.
- ♦ `#Thread threadName->#CallStack / #CallStack` shows all operations currently on stack on the thread `threadName` on the focus thread.
- ♦ `#Thread threadName-># EventQueue/# EventQueue` shows all events currently in queue on the thread `threadName` on the focus thread.
- ♦ `#Threads` shows the status of all threads. Each live thread is displayed as either reactive or suspended. One of the threads has an asterisk by its name, indicating it is the active thread.
- ♦ `#Breakpoints` shows a list of currently active breakpoints.

Examples

```
show A[0] states
```

Shows the current states of A[0].

```
show #all all
```

Displays all information about all instances.

```
show #Breakpoints
```

Shows all breakpoints.

```
show #Threads
```

Shows all threads.

```
Show MyClass relations
```

Shows all relations of all instances for every instance of MyClass.

Special Cases

Consider the following special cases when using the show command:

- ♦ **Instance objects** means only the relation, attribute, and state subjects are relevant.
- ♦ **Class objects** means showing a class displays a list of all instances belonging to that class. If the interest list includes subclasses, the tracer also displays instances of subclasses. If the interest list contains subjects relevant for instance objects, each displayed object also displays itself with respect to these subjects.

For example, the command `show A states` results in the following code:

```
A[1]
A[2]
A[3]
A[1] currently in states
  ROOT
  ROOT.S1
  ROOT.S1.S2
A[2] currently in states
  ROOT
  ROOT.S7
  ROOT.S8
A[3] currently in states
  ROOT
  ROOT.S1
  ROOT.S1.S2
```

suspend

Description

The `suspend` command suspends the specified thread. It has no effect if the thread is already suspended.

Syntax

```
suspend threadName
suspend #Thread threadName
```

Arguments

`threadName`

Specifies the thread to suspend

timestamp

Description

The `timestamp` command adds a timestamp to the output of a tracer session.

Syntax

```
timestamp <option>
```

Arguments

`no <option>`

A timestamp with no option formats the timestamp as HH:MM:SS.

-

The minus sign (-) option disables the timestamp.

`raw`

The `raw` option enables the timestamp as a cumulative counter, in milliseconds.

trace

Description

The `trace` command enables you to specify which subjects to trace for a given object (class, instance, or keyword). Subjects include existence, attributes, methods, and events. You can choose to trace all objects by one subject, one object by all subjects, or anything in between.

By default, the tracer traces all classes and instances, and does not trace system items (such as call stacks and event queues), as though you had executed the following command:

```
trace #all all
```

In animation, by default, no items are traced, as though you had executed the following command:

```
trace object nothing
```

Syntax

```
trace <object> <interest-list>
```

Arguments

`object`

Specifies the object to be traced. It can be one of the following items:

- ◆ The name of a class appearing in code, such as `A`. The trace command is applied to all instances of class `A`.
- ◆ The name of an instance that currently exists in the execution, such as `A[3]`. You cannot refer to instances before their construction or after their destruction.
- ◆ A navigation expression, such as `A[3] -> itsB[2]`. See [Navigation Expressions](#) for more information.
- ◆ The name of a package appearing in the code. The tracer will report on all classes in the package.
- ◆ A keyword understood by the tracer. These keywords are not case sensitive. The possible keywords are as follows:
 - `#All` means all classes appearing in code.
 - `#CallStack` means operations currently on stack; that is, operations started but not yet terminated, including behavior operations defined on transitions.
 - `#Thread threadName -> #CallStack` means the call stack of the thread `threadName`. (All operations that started on this thread but have not yet terminated, including behavior operations defined on transitions).
 - `#EventQueue` means a queue of all pending events; that is, events sent but not yet received.
 - `#Thread threadName -> #EventQueue` means the queue of all pending events of the thread name `threadName`. (Events sent but not yet received.)
 - `#Breakpoints` means the list of all breakpoints.

`interest-list`

Specifies the list of subjects, separated by a commas. The interest list determines what information about the object is reported to you.

The possible subjects are as follows:

existence	constructors
relations	destructors
attributes	timeouts
states	parameters
controls	subclasses
methods	threads
events	

The keywords `all` and `nothing` indicate all or none of these subjects.

Precede a subject with a plus (+) or minus (-) sign to add or subtract subjects from the current interest list. If there is neither a + nor -, the subjects typed become the current interest list, replacing any subjects previously selected.

The subject `existence` reports on the existence of the object.

The subject `subclasses` applies the trace commands to all of a class's subclasses. It is relevant only to class objects.

Command Semantics

You set or modify the interest list for a given object with the subjects that you list following the name of the object.

Example 1

The following command sets the interest list of `B[5]` to relations:

```
trace B[5] relations
```

The tracer will now display a message every time a relation of object `B[5]` is modified. For example:

```
OMTracer B[5] item A[7] added to relation itsA
```

Note that only messages regarding relations are displayed for `B[5]`.

Example 2

The following `trace` command adds relations to the interest list of `B[5]`:

```
trace B[5] +relations
```

Other messages regarding `B[5]` are displayed or not depending on the value of the interest list before the subject addition command was given.

Example 3

The following `trace` command removes relations from the interest list of `B[5]`:

```
trace B[5] -relations
```

The effect of this command is that no message is displayed about the relations of object `B[5]`. Other messages regarding `B[5]` are displayed or not depending on the value of the interest list before the command was given.

For the full list of messages by subject, see [Tracer Messages by Subject](#).

Example 4

If all subjects in the interest list appear with a `+` or `-` sign, the interest list of the object is modified. For example, the following command adds the subject relations to the interest list of `B[5]` and removes the subject states:

```
trace B[5] +relations, -states
```

In contrast, the following command sets the interest list of `B[5]` to include exactly the subject's relations and states:

```
trace B[5] relations, states
```

Special Cases

Consider the following special cases when using the `trace` command:

- ♦ **Class objects** means class objects propagate their interest lists to all their current and future instances. For example, the following command adds the relation subject to all instances of class `A`:

```
trace A +relations
```

New instances of class `A` created after you execute this command also have relations added to their interest list.

The subject subclasses is relevant only for class objects. It propagates the interest list in the given command to all subclasses (recursively).

- ♦ **#All** means setting the interest list for the keyword `#All` sets the interest lists for all classes in the executable. Modifying the interest list of `#All` modifies the interest lists of all classes in the executable.
- ♦ **#Thread <threadName>-># CallStack / #CallStack** means for call stack objects, only the parameter, method, constructor, and destructor subjects are relevant. Adding these to the interest list will display method (operation), constructor, and destructor calls, with or without parameters, even if the called item is not set to be traced.
- ♦ **#Thread threadName->#EventQueue / #EventQueue** means for event queue objects, only the parameter, timeouts, and method subjects are relevant. Adding these to the interest list will display events sent and received, with or without parameters, even if the sending or receiving item is not set to be traced.

watch

Description

The `watch` command displays all information as changes occur.

Syntax

```
watch
```

Tracer Messages by Subject

The following table lists the possible tracer messages for each subject.

Subject	Messages
Attributes	XXX[1] Attribute Values: J = 1 k = 3.4 b = 0x55f00b myFoo = testA[12] XXX[1] Attribute Values changed - new Values J = 1 k = 3.5 b = 0x55f00b myFoo = testA[12] The following message can appear anywhere a piece of code has changed the values of the attributes: State XXX[1] Entered state testA XXX[1] Exited state Kuku All attribute values are displayed. The sending and receiving of events by XXX[1] is determined by the interest list of state or operations.
Constructors	XXX[1] invoked YYY() main() invoked YYY() YYY() returned
Destructors	XXX[1] invoked ~YYY() main() invoked ~YYY() ~YYY() returned
Existence	class XXX new instance XXX[1] created instance XXX[1] deleted Instances get their names from their class (X[1], X[2], and so on). Names are unique. Once a name is used, it is not used by a new instance even if the original instance no longer exists. Instances with a No for existence are not traced for any other subject, and appear in the tracer messages as untraced. Instance Kuku[4] renamed testA[2].myKuku This message displays only when Kuku[4] is connected via the composite relation myKuku to testA[2].

Subject	Messages
Methods	XXX[1] invoked YYY[2]->doIt(j=3, k = 4.3) XXX[1] invoked YYY[2]->doIt(int, float) main() invoked Kuku[1]->testA() YYY[2]->doIt(j=3, k=4.3) returned YYY[2]->doIt(int, float) returned XXX[2] sent YYY[8] event start(starter = Kuku[8], times = 2) XXX[2] sent YYY[8] event start(Kuku *, int) Kuku[8] sent to itself Event wakeup(time=10.5) Kuku[8] sent to itself event tm(200) at ROOT.testA YYY[8] received from XXX[2] event start(starter = Kuku[8], times = 2) YY[8] sent XXX[2] event start(Kuku *, int) Kuku[8] itself Event wakeup(time=10.5) Kuku[8] received from itself event tm(200) at ROOT.testA
Parameters	These messages indicate whether methods and events are displayed via their: • Parameters For example: (doIt(j=3, k = 4.3), start(starter = Kuku[8], times = 2) • Signature For example: (doIt(int, float), start(Kuku *, int)) When A sends something to B, the parameters in the send message depend on A and the parameters in the receive message depend on B.

Subject	Messages
Relations	A report on the status of all relations displays together with the notification message on the creation of the new instance: <pre> Relation itsFoo - Empty Relation itsKuku - Kuku[1], Kuku[4], Kuku[2] ... XXX[1] instance Kuku[7] added to relation itsKuku XXX[1] relation itsFoo set to testA[2] XXX[1] instance Kuku[7] removed from relation itsKuku XXX[1] relation itsKuku cleared </pre>
Timeouts	<pre> XXX[1] set tm(tttt) at ROOT.sss XXX[1] cancelled tm(tttt) at ROOT.sss </pre>

Ending a Trace Session

End a tracing session by doing one of the following actions:

- ◆ Type `quit` and press the **Enter** key at the command-line.
- ◆ Click the **Stop Make/Execution** button  on the **Code** toolbar.
- ◆ Select **Code > Stop Execution**.

Otherwise, the trace session ends automatically when the application terminates.

Managing Web-enabled devices

This section describes the process of managing Rational Rhapsody-built embedded software via the Internet. It examines how, in the development process, to set components as Web-enabled, and how, in the maintenance process, to monitor, control, and make changes to embedded software. This section also includes how to specify preferences in the automatically generated Web pages, which serve as an interface for updating or changing Rational Rhapsody-built software.

Note

The Webify Toolkit is supported only on Internet Explorer Version 5.0 or higher. See the *IBM Rational Rhapsody Readme* file for the list of currently supported environments.

Use of Web-enabled Devices

Web-enabled devices contain Rational Rhapsody-built embedded software that you can monitor and control remotely, in real-time, from the Internet. After assigning Web manageability to a model's elements and running the code for that model, Rational Rhapsody automatically generates and hosts a Web site that accesses the application's components through a built-in Web server. The Web pages created by running Web-enabled Rational Rhapsody code serve as a graphical interface for the management of embedded applications. By using the interactive functionality of this interface, you can remotely control the performance of devices.

Although Web-enabling requires no knowledge of Web hosting, design, or development, development teams that want to refine the capability or appearance of their Web interface can do so using their favorite authoring tools.

Besides its ability to manage devices remotely, Web-enabling a device offers the following benefits to the development process:

- ♦ Web browser serves as a window into the device, through which you look into the model to see how a device will perform, eliminating the time-consuming development of writing protocol and attaching hooks that report performance information.
- ♦ Graphical interface provides additional testing on-the-fly and debugging through visual verification of the state of a device before shipping to a manufacturer.

- ◆ Easily created interfaces, exposed via the Internet, serve as visual aids in collaborative planning and engineering and provide a vehicle for rapid prototyping of an application during development.
- ◆ Filtered views can focus customer-specific aspects of a model.
- ◆ Capability to refresh continuously only the changed values and statuses does not overtax device resources.

To Web-enable software, you must perform several tasks from both the server side (in Rational Rhapsody) and the client side (from the Web GUI). In Rational Rhapsody, you select which elements of a model to control and manage over a network, and assign Web-enabling properties to those elements, then generate the code for the model.

To manage the model from the Web GUI, navigate to the URL of the model. Pages viewed in a Web browser act as the GUI to remotely manage the Rational Rhapsody-built device. You can control and manage devices through the Internet, remotely invoking real-time events within the device. Teams can use the Web-enabled access as part of the development process (to prototype, test on-the-fly, and collaborate) or to demonstrate the behavior of a model.

Setting Model Elements as Web-Manageable

The first step in Web-enabling a working Rational Rhapsody model is to set elements of the model for Internet exposure. Within a working Rational Rhapsody model, select which elements of the device's application that you want to control or manage remotely through a Web browser, and assign Web-managed properties to those elements.

Note

You cannot webify a file-based C model.

Limitations on Web-Enabling Elements

At the design level, you might find workarounds for building Web-managed models around elements currently not supported by Web-enabling. The following table lists Rational Rhapsody elements and whether they can be Web-enabled.

Element Type	Support and Restrictions
Attributes	Supported types include <code>char</code> , <code>bool</code> , <code>int</code> , <code>long</code> , <code>double</code> , <code>char *</code> , <code>OMBoolean</code> , <code>OMString</code> , <code>RicBoolean</code> , and <code>RicString</code> . Arrays are not supported. Attributes must have either an accessor or mutator, or both.
Classes	Selected objects within classes must be set as Web-enabled. Web-enabling a class does not Web-enable all the child objects of the class.
Native types	Short and all unsigned types (<code>unsigned char</code> , <code>unsigned int</code> , <code>unsigned short</code> , and <code>unsigned long</code>) are supported.
User-defined types	Not supported.
Global variables	Not supported.
Global functions	Not supported.
Packages, components and diagrams	Not supported.

Note the following restrictions:

- ◆ You are limited to 100 Web-enabled instances per model. Depending on the element type, enabling one element might enable multiple instances.
- ◆ Code generation for Web management is unavailable in Rational Rhapsody in Ada.
- ◆ C-style strings (`char*`) and `RicString` types might cause memory leaks. Setting new values for this type of string from the Web interface assigns newly allocated strings to the attribute. Be sure to properly deallocate this memory. Note that `OMString` will not cause memory leaks.
- ◆ C unions, bit fields, and enumerations are not supported.
- ◆ C++ templates and template instantiations are not supported.

Selecting Elements to Expose to the Internet

The first step in Web-enabling a working Rational Rhapsody model is to set its components and elements as Web-manageable. When considering which elements to Web-enable within a model, keep in mind the current restrictions and limitations (see [Limitations on Web-Enabling Elements](#)).

To expose your model to the Web:

1. In the browser, navigate to **Components** > (component name) > **Configurations**. Choose an active configuration belonging to an executable component.
2. In the Features window, click the **Settings** tab.
3. Select the **Web Enabling** check box. This enables Web-enabled code generation for the configuration.

Note: You cannot webify a file-based C model.

4. Optionally, click the **Advanced Settings** button to set the Webify parameters. Rational Rhapsody opens the Advanced Settings window.

This window contains the following controls:

- ♦ **Home Page URL** specifies the URL of the home page. The default value is as follows:
`cgibin?Abs_App=Abstract_Default`
- ♦ **Signature Page URL** specifies the URL of the signature page. The default value is `sign.htm`.
- ♦ **Web Page Refresh Period** specifies the refresh rate for the Web page, in milliseconds. The default value is 1000 milliseconds.
- ♦ **Web Server Port** specifies the port number of the Web server. The default value is 80.

Each of these parameters corresponds to a property under `WebComponents::Configuration`. This enables you to save your updated settings with every model, or change them by editing the property values.

5. Repeat Steps 1–4 for all the library components that contain Web-enabled elements.
6. Navigate to the elements within a package that you have decided to Web-enable.
7. Double-click the element to open its Features window.
8. Set the stereotype to «Web Managed». Do this for each element you want to view or control from the Internet.

If the element already has an assigned stereotype, you need to Web-enable it through a property, as follows:

- a. Right-click the element and select **Properties**.
- b. Select `webComponents` as the subject, then set the value of the `webManaged` property within the appropriate metaclasses to `Checked`.

Currently, the supported metaclasses for the `webComponents` subject are `Attribute`, `Class`, `Configuration`, `Event`, `File`, `Operation`, and `WebFramework`.

9. Generate the code for the model, build the application, and run it.

Connecting to the Web Site from the Internet

Rational Rhapsody comes with a collection of default pages that serve as a client-side GUI to the remote model. When you run a Web-enabled model, the Rational Rhapsody Web server automatically generates a Web site, complete with a file structure and interactive capability. This site contains a default collection of generated on-the-fly pages that refreshes each element when it changes.

Navigating to the Model through a Web Browser

After generating, making, and running a Rational Rhapsody model with Web-enabled objects, open Internet Explorer (version 5.0 or higher).

For Applications Running on Your Local Machine

In the address box, type the following URL:

```
http://<local host name>
```

In this URL, <local host name> is the “localhost,” machine name, or IP address of your machine.

If you changed the Web server port using the Advanced Settings window, type the following URL:

```
http://<host name>:<port number>
```

By default, the Objects Navigation page loads in your Web browser.

For Applications Running on a Remote Machine or Server

In the address box, type the following URL:

```
http://<remote host>
```

In this URL, <remote host> is the machine name or IP address of the machine running the Rational Rhapsody model.

If you changed the Web server port using the Advanced Settings window, type the following URL:

```
http://<remote host name>:<port number>
```

By default, the Objects Navigation page loads in your Web browser.

Troubleshooting Problems

If you have trouble connecting to a model through the Internet, verify the following items:

- ◆ You have IP access to the server you are trying to access.
- ◆ You have Web-instrumented the active component.
- ◆ You have Web-enabled the individual elements.
- ◆ By default, Web-enabled Rational Rhapsody models listen on port 80. Make sure you are currently not running another HTTP protocol application listening on the same port, including another Web-enabled Rational Rhapsody model, or personal Web server. Note that several operating systems (for example, Solaris) do not allow access to port 80. Assign a different port to the Rational Rhapsody model using either the Advanced Settings window or the `webComponents::Configuration::Port` property.
- ◆ A specific setting of Internet Explorer might cause the following behavior:

The right frame of the “Objects Navigation” page is empty, when it should show the selected object.

To resolve this, in Internet Explorer, go to **Tools > Internet Options > Advanced** and clear the **Use Java2 for <applet> (Requires Restart)** check box.

Connecting to Filtered Views of a Model

If you know the HTTP address of a filtered view of a model, you can initially connect to a model through that view. If you want to monitor or control only certain behaviors of a device from one tailored Web GUI page, you go directly to that page using this method. Rational Rhapsody does not yet support access controls, so providing access to filtered views should be done for the sake of convenience only, because it is not secure.

For information on filtering Web GUI views, see [The Define Views Page](#).

The Web GUI Pages

The default Rational Rhapsody Web server comes with a collection of pages, served up on-the-fly. These pages populate dynamically and contain the status of model elements and present different capabilities and navigation schemes.

The GUI includes the following pages:

- ◆ The Objects Navigation Page
- ◆ The Define Views Page
- ◆ The Personalized Navigation Page
- ◆ The Upload a File to Server Page
- ◆ The Statistics Page
- ◆ The List of Files Page

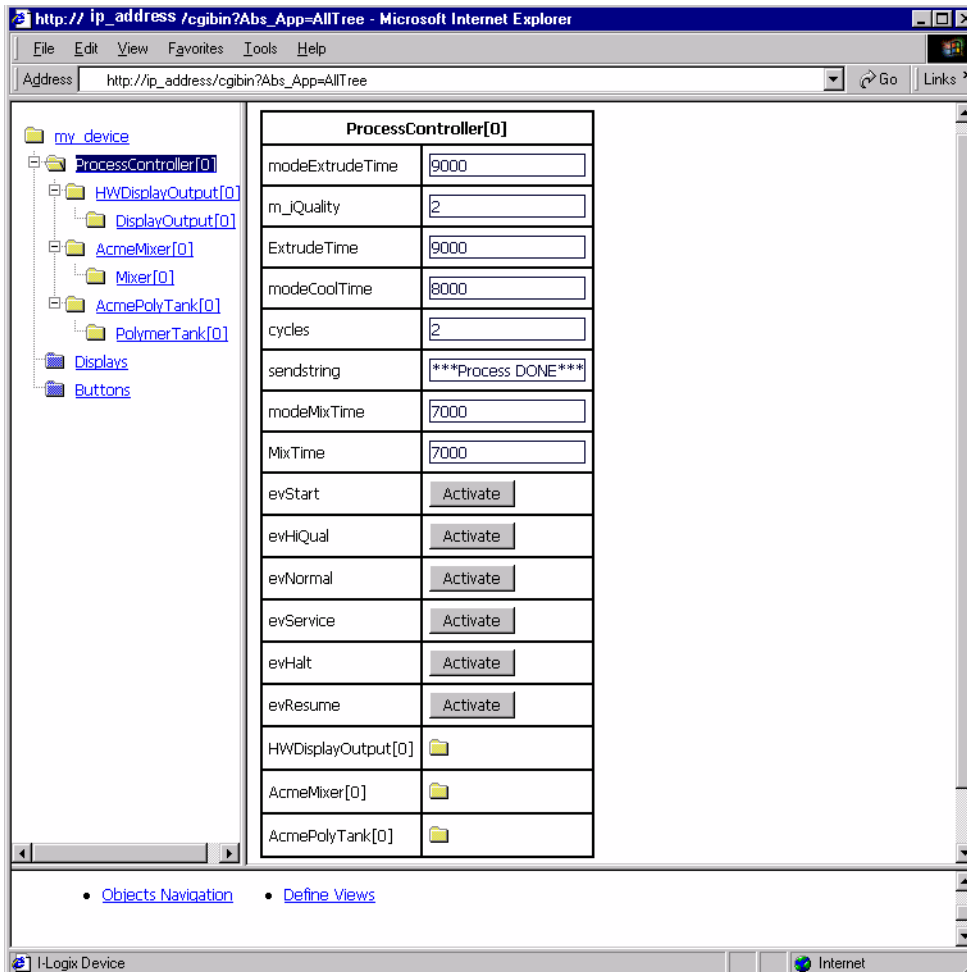
The Objects Navigation Page

The Objects Navigation page provides easy navigation to Web-exposed objects in a model by displaying a hierarchical view of model elements, starting from top-level aggregates. By navigating to, and selecting, an aggregate in the left frame of this page, you can monitor and control your remote device in the aggregate table displayed in the right frame.

This page serves as an explorer-like GUI wherein aggregates appear as folders and correspond to the organization of objects in the model. For information on reading and using aggregate tables, see [Viewing and Controlling of a Model via the Internet](#).

When you select an object in the browser frame, it is displayed in the right frame. The status and input fields of objects in the selected aggregate populate the table dynamically. The left column of the table lists the name of the Web-enabled object; the right column lists the corresponding real-time status, input field, activation button, or folder of child aggregates. For information on controlling a model through an aggregate table, see [Customizing the Web Interface](#).

The following figure shows the Objects Navigation page.



The bottom frame in this page contains links to the other automatically generated pages of the Web GUI to manage embedded devices. These links appear at the bottom of all subsequent pages of the default Web GUI and can be customized with your corporate logo and links to your own Web pages.

The Define Views Page

The Define Views page, shown in the following figure, provides a GUI for filtering views into the Rational Rhapsody model. These views can simplify monitoring and controlling maintenance and help you focus on key elements of your models.

Define Views

Name of view

Mark	Name	Type
☐	ProcessController[0]	aggr
☐	ProcessController[0]::modeExtrudeTime	int
☐	ProcessController[0]::m_iQuality	int
☐	ProcessController[0]::ExtrudeTime	int
☐	ProcessController[0]::modeCoolTime	int
☐	ProcessController[0]::cycles	int
☐	ProcessController[0]::sendstring	string
☐	ProcessController[0]::modeMixTime	int
☐	ProcessController[0]::MixTime	int
☐	ProcessController[0]::evStart	unari
☐	ProcessController[0]::evHiQual	unari
☐	ProcessController[0]::evNormal	unari
☐	ProcessController[0]::evService	unari
☐	HWDisplayOutput[0]::evAlertDPOn	unari
☐	HWDisplayOutput[0]::evAlertDPOff	unari
☐	HWDisplayOutput[0]::evAlertTankDrainOn	unari
☐	Displays	view
☐	Buttons	view

• [Objects Navigation](#) • [Define Views](#)

The Define Views page enables you to create and name a view of selected elements. The page includes a list of check boxes beside all the available elements that can be included in the view,

and their corresponding element types. Click each element or aggregate to drill down to a graphical representation of the element value, or its aggregate collection of name-value pairs.

To design a view:

1. Check the boxes beside each element you want to include in your view. To include all elements, click **Select All**.
2. Type the name of your view in the **Name of view** box.

The **Reset** button clears the name you entered and all of your selections.

3. Click **Submit**.

The new view name displays at the bottom of the element table, next to its check box, with “view” as its type.

Note

All views created from the Design View page store in memory and will not be saved when you close Rational Rhapsody. See the example and readme file provided in <Rational Rhapsody installation path>\MakeTmpl\Web\BackupViews.

The Personalized Navigation Page

The Personalized Navigation page, shown in the following figure, enables you to customize your Web interface and add links to more information about your model. This page, conveniently provided as an HTML page, can be opened and viewed for design testing.

Using this page, you can add hyperlinks to the bottom navigation of your Rational Rhapsody GUI, such as a link to e-mail comments to a customer support mailbox. By default, the file includes two links (to the Objects Navigation and Define Views pages) whose addresses are displayed within anchor tags.

To customize the bottom frame of the Web GUI:

1. If you need a general template to edit, right-click the bottom of the Rational Rhapsody Web GUI, and grab the source code.
2. Design the page to your liking, adding your logo and links and adjusting the table structure in the file to accommodate your changes.
3. Rewrite the default file with your own by uploading it to the Rational Rhapsody Web server, either through the Rational Rhapsody interface or through the Upload a File to the Server page. For more on uploading files to the Web server, see [The Upload a File to Server Page](#).

If you do not want to overwrite the default `sign.htm` file, do one of the following actions:

- ♦ Change the signature page using the Advanced Settings window or the property `webComponents::Configuration::SignaturePageURL`. This is the preferred method.
- ♦ Upload your own .html file to the Rational Rhapsody Web server and call your new file from within the Rational Rhapsody Web configuration file by adding the following function call:

```
SetSignaturePageUrl(<name of your new file>)
```

Viewing and Controlling of a Model via the Internet

Controllers manage Rational Rhapsody-built devices through the Internet, remotely invoking real-time events within the device. After Web-enabling, running a Rational Rhapsody model, and connecting to the device in a Web browser, controllers can view and control a device through the Rational Rhapsody Web GUI, using aggregate tables.

Aggregate tables contain name-value pairs of Rational Rhapsody Web-enabled elements that are visible and controllable through Internet access to the machine hosting the Rational Rhapsody model. Navigate to aggregate tables in the Rational Rhapsody Web GUI by browsing to classes selected on the Objects Navigation page.

The following figure shows an example of the name-value pairs as they appear in an aggregate table. This table shows the name of each element within the aggregates, and whether the values are readable or writable.

ProcessController[0]	
modeExtrudeTime	7000
modeCoolTime	6000
cycles	3
sendstring	***Process... ***Process HALT***
modeMixTime	100
evStart	Activate

You can monitor a device by reading the values in the dynamically populated text boxes and combo-boxes. When a string value extends beyond the width of the text field, position the mouse arrow over the text field to display a tooltip, as shown for the `sendstring` element.

You can control a device in real-time by clicking the **Activate** button, which initializes an event, or by editing writable text fields (the text boxes that do not have a gray background). Note that an input box displays a red border while being edited, indicating that its value will not refresh until you exit the field.

The Web GUI uses different fields and folders to indicate types of values and their read and write permissions. The following table lists the way name-value pairs are displayed.

Name-Value	Read/Write Indication
Numeric values	Readable values dynamically populate the text box. Write-protected values display in a text box with a gray background; otherwise, the value is writable. To send a changed value, type the value in the text box and either press Enter or exit the box by clicking outside of it.
String values	Readable values dynamically populate the text box. Write-protected values display in a text box with a gray background,; otherwise, the value is writable. To send a changed value, type the value in the text box and either press Enter or exit the box by clicking outside of it.
Boolean variable	Writable values display in a combo-box containing True and False values.
Activation buttons	Clicking these buttons initializes the corresponding event in the model.
Tan folders	Denote child aggregates; clicking a folder displays the selected aggregate within.
Blue folders	Denote user-created views nested within aggregates; clicking a folder displays the selected contents within.

Customizing the Web Interface

You can customize the Web interface for a model by creating your own pages to add to their Rational Rhapsody Web GUI, or by referencing the collection of on-the-fly pages that come with Rational Rhapsody. In addition, you can configure the Rational Rhapsody Web server, giving it custom settings appropriate for your management of a Web-enabled device.

Adding Web Files to a Rational Rhapsody Model

You can add your own HTML, image, multimedia or script files to the Rational Rhapsody Web server file system from within the Rational Rhapsody application.

To upload the files to the Web server:

1. In the browser of a working Rational Rhapsody model, navigate to **Components** > **<Component Name>** > **Files**.
2. Right-click the **Files** category.
3. Select **Add New File**. Type the name of the new file.
4. In the browser, right-click the file and select **Features**. Set the following controls:
 - ♦ **Name**. Type in the name of your HTML file.

- ♦ **Path.** Type in, or browse to, the file you want to upload to the Web server.
 - ♦ **File Type.** Set to **Other**.
5. Select the **Properties** tab.
 6. Set the `WebComponents::File::WebManaged File` property to **Checked**.
 7. Click **OK**.
 8. Generate the code. Code generation converts the file to a binary format, and embeds it into the executable file generated by Rational Rhapsody.

You can view Web files you have added to your model by going to the following URL:

`http://<ip_address>/<filename>`

Accessing Web Services Provided with Rational Rhapsody

Rational Rhapsody comes with a collection of generated on-the-fly Web pages for you to add to your Web interface to help in the development process. At run time, these pages provide useful functionality in the development process; however, easy access to them in deployment of devices is not appropriate.

Note

If you need to generate the Web Services libraries for your target, see the *IBM Rational Rhapsody Readme* file for your version of Rational Rhapsody to see the list of currently supported environments for the Webify Toolkit. If the libraries you need do not exist in your `$OMROOT\lib` directory, contact IBM customer support.

The following sections describe how to use, access, and link to each of these pages.

The Upload a File to Server Page

You can upload your own HTML, image, multimedia or script files to the Rational Rhapsody Web server through the Internet from the Upload a File to Server page.

Note

To enable this functionality, you must first add the `RegisterUpload()` call to the `webconfig.c` file.

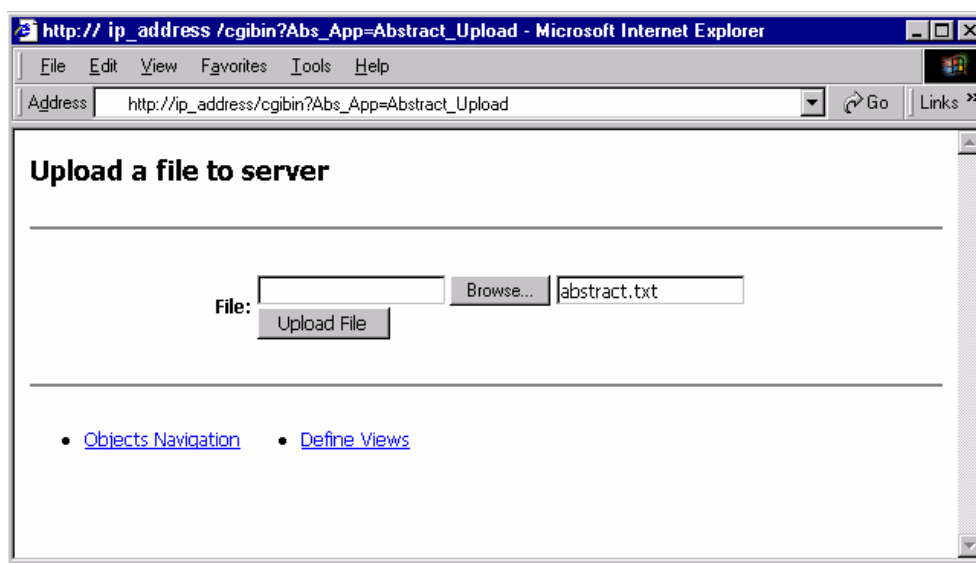
To navigate directly to the page, use the following URL:

`http://<ip_address>/cgibin?Abs_App=Abstract_Upload`

To add this page into your Personal Navigation page, its open anchor tag should read as follows:

```
<a href="cgibin?Abs_App=Abstract_Upload">
```

The following figure shows the Upload a File to Server page.



To upload a new file, or overwrite a file, to the Rational Rhapsody Web server using this page:

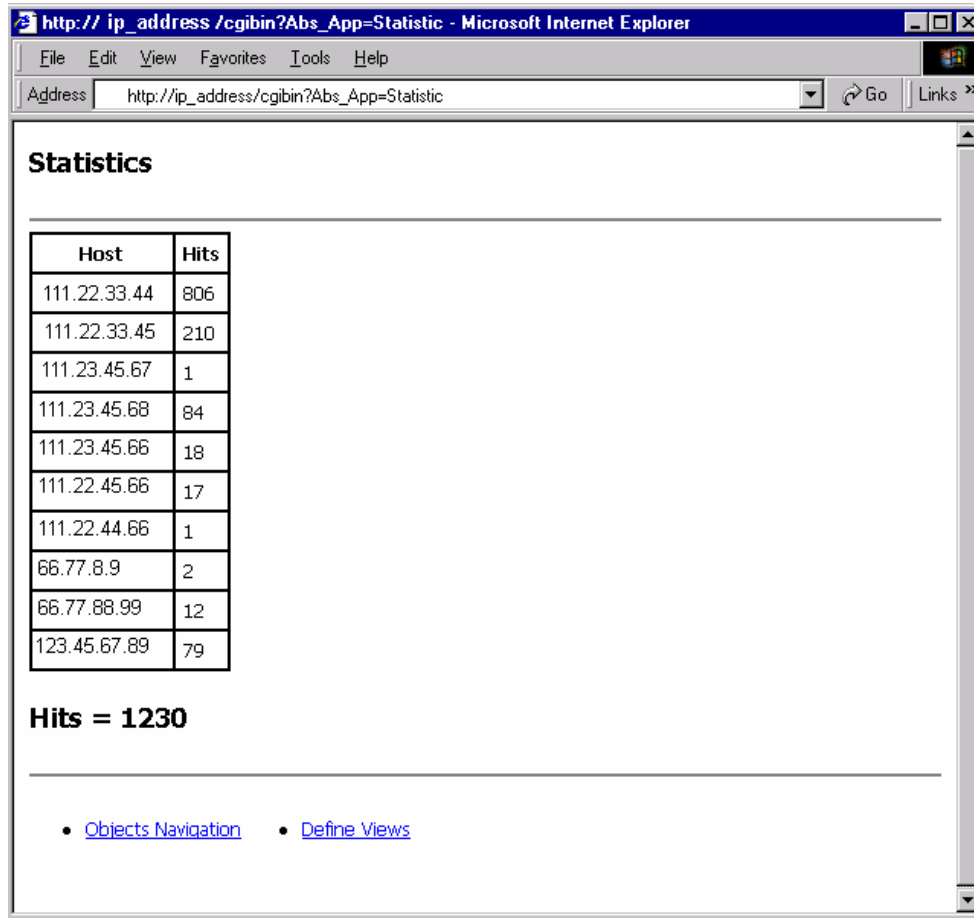
1. Type in the path, or browse, to the file.
2. Click **Upload**.

Note: This page should only be used by developers who understand the impact of their uploads. The collection of on-the-fly services provided with the Rational Rhapsody Web server are intended to assist as a development tool; easy access to them in deployment of devices is not appropriate.

For an example of the server settings for using this page, see <Rational Rhapsody installation path>\Share\MakeTpl\web\WebServerConfig\AddUploadFunctionality.

The Statistics Page

The Statistics page, shown in the following figure, tabulates page and file requests (“hits”) from each machine accessing the model.



You can navigate directly to the page by going to the following URL:

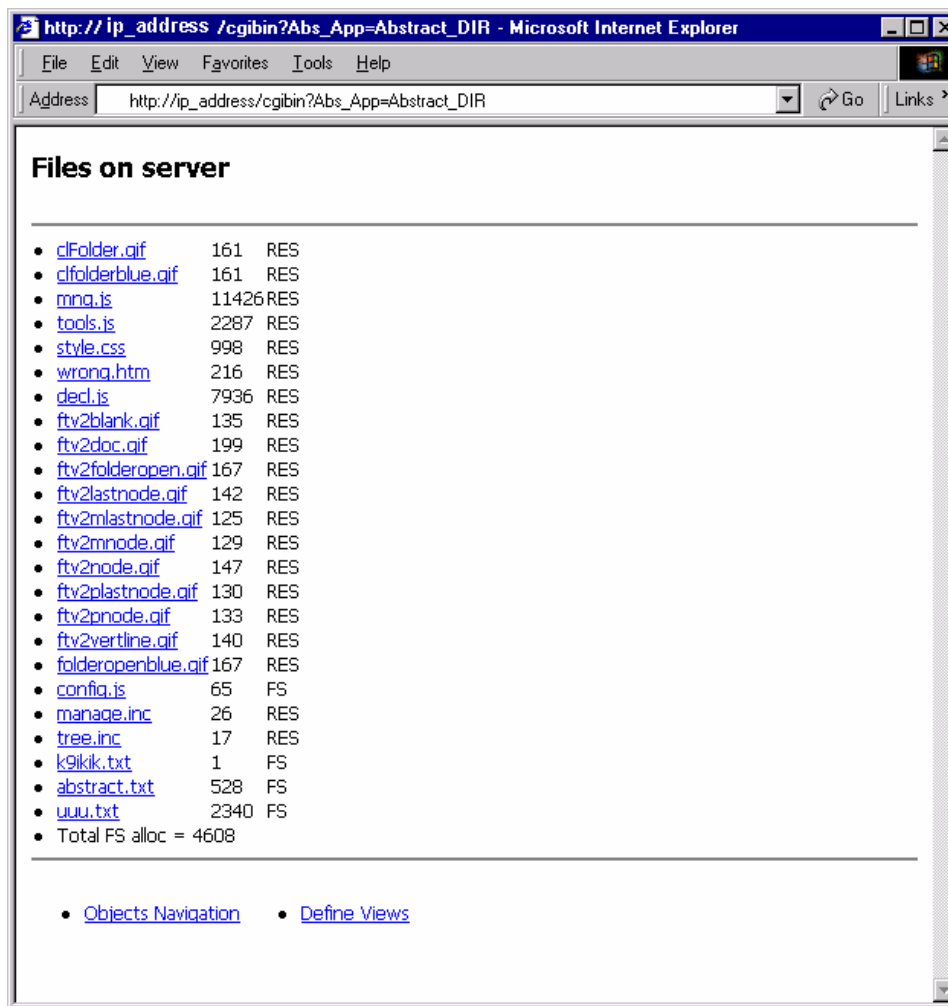
`http://<ip_address>/cgibin?Abs_App=Statistic`

To add this page into your Personal Navigation page, its open anchor tag should read as follows:

`<a href="cgibin?Abs_App=Statistic">`

The List of Files Page

This List of Files page, shown in the following figure, lists all the text and graphic files that come with the default Web GUI that generates automatically when running a Web-enabled Rational Rhapsody model.



You can navigate directly to the page by going to the following URL:

`http://<ip_address>/cgibin?Abs_App=Abstract_DIR`

To add this page into your Personal Navigation page, its open anchor tag should read as follows:

```
<a href="cgibin?Abs_App=Abstract_DIR">
```

The list includes the size, in bytes, and type of each file, and displays the total size of all included Web GUI files.

Files labeled as “RES” denote code segments; files labeled “FS” denote those stored in the file system.

To use the `signEnhanced.htm` navigation page, rename the page `sign.htm`.

Adding Rational Rhapsody Functionality to Your Web Design

You can completely change the layout and design of the Web interface to your Rational Rhapsody model and still have all the functionality provided, by default, when running a Web-enabled Rational Rhapsody model. You can create an interface that refreshes changed element values in real time, and provide the same interactive capabilities to remotely control and monitor a device.

Calling Element Values

After designing a static version of your HTML page, using placeholders for element values, you can overwrite each placeholder text with script that will call the values of each element dynamically from your uploaded page.

1. If you want to design the layout of your page first, design your Web page, leaving static text as a placeholder for a dynamic value.
2. Edit your HTML file, including the `manage.inc` file in the header of the HTML file in script tags before the `</head>` tag:

```
<head>
  <title>Your Title Here</title>
  <script src='manage.inc'></script>
</head>
```

The `manage.inc` file includes a collection of JavaScript files that control client-side behavior of the Rational Rhapsody Web interface.

3. Edit your HTML file, substituting function-calling script for each static value placeholder.

For example:

```
<body>value of evStart</body>
```

This becomes:

```
<body><script>show('nameOfElement')</script></body>
```

In this sample code, `nameOfElement` is the element name assigned to the element by Rational Rhapsody, visible in the default Web interface. Be sure to use the full path name of the element in the `show` function, as in the following example:

```
show('ProcessController[0]::OMBoolean_attribute');
```

4. Save the file and upload it to the Rational Rhapsody Web server (see [Adding Web Files to a Rational Rhapsody Model](#)).

You can link to this file in your Web interface from your new design scheme. If you want to make a page the front page of your Web GUI, see [Setting a Home Page](#). Keep in mind that Rational Rhapsody does not yet support a hierarchical file structure, so HTML and image files are at the Web server's root directory.

Binding Embedded Objects to Your Model

Using this design method, you embed graphical elements in your Web page as JavaScript objects, then bind those objects to the Rational Rhapsody model's real-time values. By binding embedded graphics and mapping them to the values of Web-enabled elements, you can create a page where an image is displayed in the Web interface when a process has stopped (for example, a stop sign), whereas another indicates that the process is running (such as a green light).

One approach to making such pages is to design a page with all your static elements, then add script to your HTML:

1. Edit your HTML file, including the `manage.inc` file in the header of the HTML file in script tags before the `</head>` tag:

```
<head>
  <title>Your Title Here</title>
  <script src='manage.inc'></script>
</head>
```

The `manage.inc` file includes a collection of JavaScript files that control client-side behavior of the Rational Rhapsody Web interface.

2. To embed model elements in the page, create JavaScript objects of the `webObject` type and bind each object with the elements of the device you are managing and controlling through the Internet, using the `bind` function.

The following sample code displays a yellow lamp when a Boolean element's value is true, and a red lamp when the value is false:

```
<html><head><title>Page Title</title>
<script src='manage.inc'></script>
<script>
function updateMyLamp(val)
{
    if (val == 'On')
    {
        document.getElementById('myImage').src = 'redLamp.gif';
    }
    else
    {
        document.getElementById('myImage').src = 'yellowLamp.gif';
    }
}
</script>
</head>
<body>
<script>
var lamp = new webObject;
bind(window.lamp,
```

```
'ProcessController[0]::OMBoolean_attribute');
lamp.update = updateMyLamp;
</script>
<img id=myImage border=0>
<hr noshade>
<i>Rotate the bool values here</i>
<script>show('ProcessController[0]::rotate');
```

The declaration of the object and binding takes place in the header of the document, between the second pair of script tags.

The bind function serves as a bridge between the element values in the model and the Web interface. It takes two arguments, the variable name of the JavaScript object (lamp in the example) and the name of the model element in Rational Rhapsody (ProcessController[0]::rotate in the example).

In the example, the following line refreshes updated model values in the Web GUI:

```
lamp.update = updateMyLamp;
```

This update function accepts one argument, which is the new value of the element.

If a GUI control in the page needs to pass information to the device, call the set method of the corresponding object. The set method accepts one argument, the newly set value.

Calling Model Functions

In addition to the ability to display attribute values on your custom Web pages, you can add to Web pages the ability to call functions in your model.

To allow a Web page to call a function from your model, make the following changes to the page:

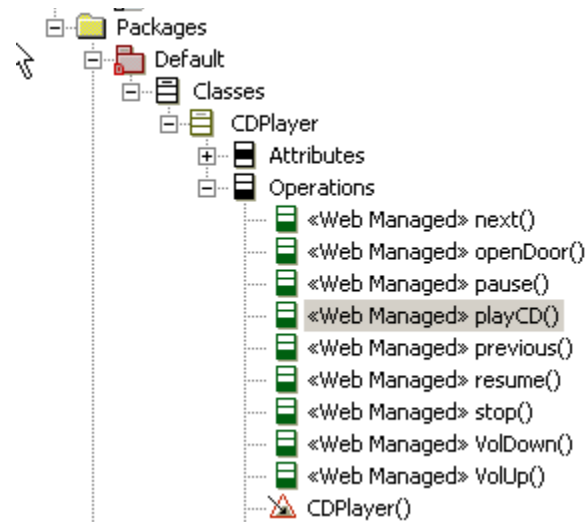
1. Declare a new variable as a webObject.
2. Call the bind function, providing the variable name and the name of the function from your model as parameters.
3. Add a link or other control to call the set() function for the new variable.

The following code snippets reflect these steps:

```
<script>
var play = new WebObject;
bind(window.play, 'CDPlayer[0]::playCD');
</script>

<p><a href="javascript:play.set();"></a></p>
```

In this example, `CDPlayer[0]` represents the relevant object in the model, and `playCD` is the name of the operation that is to be called.



Customizing the Rational Rhapsody Web Server

You can customize the Rational Rhapsody Web server in two ways:

- ♦ Using the Advanced Settings window
- ♦ Editing the `webconfig.c` file

To customize the Rational Rhapsody Web server, you can modify the `webconfig.c` file and add that file to your Web-enabled Rational Rhapsody model. The `webconfig.c` file is located within your Rational Rhapsody installation directory in `Share\MakeTmp\Web`. The file is clearly commented before each function. To change the Web server settings, edit the argument of the function to the appropriate setting.

Because other models will use this file to configure Web server settings, copy the file to another directory within your Web-enabled project, for example, and edit that copy of the `webconfig.c` file. In the process of customizing your Web server, be sure to add it to the configuration of your active component from within the Rational Rhapsody interface.

To add the modified `webconfig.c` file to your Web-enabled model:

1. In the browser of a working Rational Rhapsody model, navigate to **Components** > **<Component Name>** > **Configurations**. Select the active configuration.
2. On the Features window, on the **Settings** tab, in the **Additional Sources** box, type the location of the modified `webconfig.c` file.

The kit includes examples in the `<Rational Rhapsody installation path>\Share\MakeTmp\web\WebServerConfig` directory.

Note

If you customize your Web server using the `webconfig.c` file, your changes will overwrite the properties set in the Advanced Settings window.

Setting a Device Name

To change the name of a device, modify the argument to the `SetDeviceName` method.

For example, to change the setting of the device name to “Excellent Device”, modify the call as follows:

```
SetDeviceName("Excellent Device");
```

See `<Rational Rhapsody installation path>\Share\MakeTmp\web\WebServerConfig\ChangeDeviceName` for an example.

Setting a Home Page

To change the setting of the home page to `index.htm`, the function and argument should read as follows:

```
SetHomePageUrl("index.htm");
```

Setting a Personalized Bottom Navigation

There are two ways to personalize the bottom navigation page:

- ◆ Overwrite the `sign.htm` file with your changed design and upload it to the Rational Rhapsody Web server (see [The Personalized Navigation Page](#)).
- ◆ Use a function in the `webconfig.c` files to use another file for personalized bottom navigation.

For example, to change the page name of the bottom navigation page to `navigation.htm`, the function and argument should read as follows:

```
SetSignaturePageUrl("navigation.htm");
```

Setting a Port Number

By default, the Rational Rhapsody Web server listens on port 80. To change the port number, change the argument to the `SetPropPortNumber` method.

For example, to change the port number to 8000, use the following call:

```
SetPropPortNumber(8000);
```

Setting an Automatic Refresh Rate

To change how frequently the Rational Rhapsody Web server refreshes changed values, use the `SetRefreshTimeout` function. The argument to this function holds the refresh rate, in milliseconds.

For example, to change the refresh interval to every 5 seconds, use the following call:

```
SetRefreshTimeout(5000);
```

See `<Rational Rhapsody installation path>\Share\MakeTmpl\web\WebServerConfig\ChangeRefreshRate` for an example.

Enabling File Upload

To enable the file upload capability, add the `RegisterUpload` function to the `webconfig.c` file. This function takes no arguments.

See `<Rational Rhapsody installation path>\Share\MakeTmpl\web\WebServerConfig\AddUploadFunctionality` for an example.

Reports

Rational Rhapsody offers two ways to generate reports from the models, charts, the generated code, and other items:

- ♦ A simple and quick internal RTF report generator, described in [The internal reporting facility](#).
- ♦ A more powerful reporting tool, Rational Rhapsody ReporterPLUS

ReporterPLUS

ReporterPLUS produces reports that are suitable for formal presentations and can be output in any of these formats:

- ♦ HTML page
- ♦ Microsoft Word
- ♦ Microsoft PowerPoint
- ♦ RTF (.rtf)
- ♦ text (.txt)

You can save the file and view it in any program designed to display the report's format. See [Viewing reports online](#) for more information.

ReporterPLUS creates documents using these techniques:

- ♦ Extracting text and diagrams from a model created in Rational Rhapsody.
- ♦ Adding text and diagrams from the model and images to the document. (Note that text files do not include diagrams.)
- ♦ Adding boilerplate text specified in the ReporterPLUS template to the document.
- ♦ Formatting the document according to the formatting commands in the ReporterPLUS template, as well as the specifications in a *Word* template (.dot file), a PowerPoint template (.pot file), or an HTML style sheet (.css file). Using a .dot, .pot, or .css file is optional. You can also use HTML tags to format HTML documents.

Launching ReporterPLUS

To start ReporterPLUS from inside of Rational Rhapsody, select **Tools > ReporterPLUS**. This menu displays options for printing the model currently displayed in Rational Rhapsody.

The **Report on selected package** option is unavailable from this menu unless a package in the model is highlighted in the Rational Rhapsody browser. If one of the first two items is selected, a report can be generated using a predefined template without displaying the main ReporterPLUS GUI. If the last option is selected, ReporterPLUS starts with no model elements imported.

To start ReporterPLUS from outside of Rational Rhapsody, from the Windows Start menu, select **All Programs > IBM Rational > IBM Rational Rhapsody version number > Rational Rhapsody ReporterPLUS version number > Rational Rhapsody ReporterPLUS version number**.

ReporterPLUS templates

Rational Rhapsody includes numerous pre-fabricated report templates that you might want to use as they are or customize to meet your needs.

Note

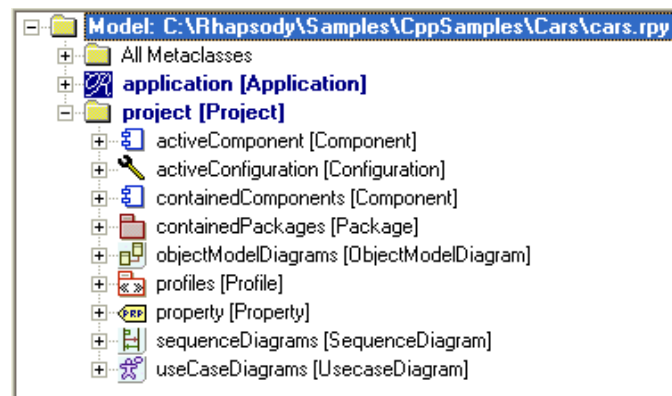
These files are stored in the Rational Rhapsody\reporterplus\Templates directory.

Using the ReporterPLUS interface

Rational Rhapsody models can be loaded into the ReporterPLUS interface and used to create generic or model-specific templates. This interface allows you to create and modify templates graphically using a drag-and-drop method.

To use the ReporterPlus templates with your model:

1. Start Rational Rhapsody if it is not already started and display the project for which you need a report.
2. Select **Tools > ReporterPLUS > Create/Edit template with ReporterPLUS**.
3. The ReporterPLUS interface opens with a **Tip of the Day** window. Close it.
4. Then the ReporterPLUS Wizard window displays. Click **Cancel**.
5. The project model's details are listed in ReporterPLUS' upper left corner as shown in this example.



6. Highlight an item in the model and the detailed description of that item displays to the right, as shown in this example.

Name	Type	Value
☐ description	String	
☐ descriptionHTML	String	
☐ descriptionRTF	String	
☐ displayName	String	
☐ fullPathName	String	
☐ fullPathNameIn	String	
☐ isOfMetaclass	Boolean	
☐ isShowDisplayName	Boolean	
☐ metaClass	String	
☐ name	String	
☐ requirementTraceabilityHandle	Long	
☐ userDefinedMetaClass	String	

Examining pre-fabricated templates

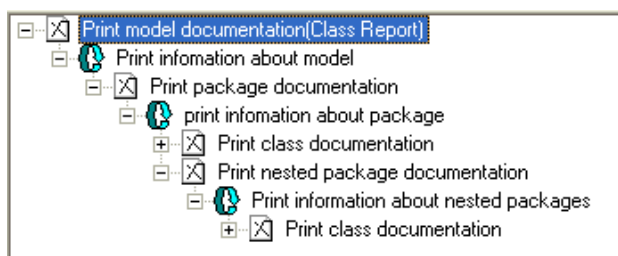
To examine the pre-fabricated Rational Rhapsody templates:

1. From the ReporterPLUS menu, choose **File > Open Template**. This displays all of the existing Rational Rhapsody report templates.

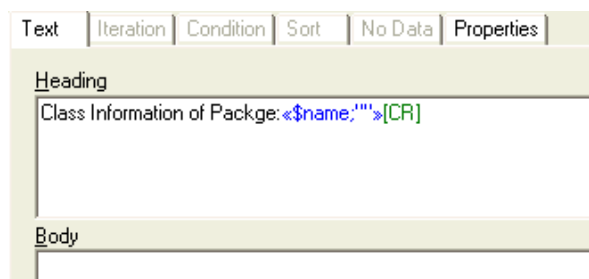
Note

The GetStarted.tpl is a simple template to use when you are becoming familiar with this reporting tool.

2. Select a template from the list. The structure of the selected template displays in the lower left corner. This structure uses a standard Windows tree design.



3. Highlight an item in the tree structure and view the details of that item in the window to the right.



Customizing templates

To customize an existing template for your Rational Rhapsody project:

1. Display your Rational Rhapsody model in ReporterPLUS.
2. From the ReporterPLUS menu, choose **File > Open Template**.
3. Select the template from the list that most closely resembles the type of report you want to generate. The structure of the selected template displays in the lower left corner.
4. An easy method for customizing the generic template for your project is to drag an item from your model down to the template area and drop it in the intended location.
5. You might also want to add standard headings and text to an existing template. To add this “boilerplate” material, highlight a section of the template in the lower left window and click the **Text** tab in the lower right window. Type text in the Heading and Body sections as wanted.
6. For more complex changes, study the Q Language that ReporterPLUS uses to define report expressions. This language is defined in a PDF file accessed from ReporterPLUS Help Topics.

Generating reports using existing templates

To generate reports quickly from the Rational Rhapsody interface using templates you have examined or created previously in ReporterPLUS:

Note

If you are going to generate a report in *Microsoft Word* or *PowerPoint*, be certain that *Word* and *PowerPoint* are closed before you start this procedure to avoid a conflict between *ReporterPLUS* and the Microsoft program.

1. With your project open in Rational Rhapsody, choose **Tools > ReporterPLUS**.
2. From the next menu, select one of these two options:
 - ♦ Report on all model elements
 - ♦ Report on selected package
3. Rational Rhapsody displays the ReporterPLUS Wizard that allows you to select the intended output format and on subsequent windows, the template, and directory location and name for the finished report.

Note

However, if your project model is very large, you should generate the report from ReporterPLUS interface for a more rapid generation.

Viewing reports online

After creating reports using ReporterPLUS templates and facilities, follow these guidelines for viewing the reports online:

- ♦ For reports generated in Linux, view the HTML reports in Mozilla Firefox and the RTF reports in Open Office 2.0 or higher.
- ♦ For reports generated in Windows, view HTML reports in any standard browser available on the PC and for the other report formats, the appropriate programs for viewing these reports launch when the report files are clicked to launch.

Generating a list of specific items

If during development you want to generate a list of items, such as all of the ports using an interface, you can focus the generated report on that section of the model.

To generate a list of specific items in a model:

1. Display the model in Rational Rhapsody.
2. In the browser, select the section of the model containing the specific items that you need in a list.
3. Select **Tools > ReporterPLUS > Report on selected package**.
4. Select the template you want to use for the report and generate and save the report.

Using the system model template

ReporterPLUS includes a template designed for systems engineering called `SysMLreport.tpl`. To use this template for your report:

1. With your model displayed in the Rational Rhapsody interface, select **Tools > ReporterPLUS** from the menu.
2. From the next menu, select **Create/Edit template with ReporterPLUS**.
3. Your model displays in the upper left corner of the ReporterPLUS interface.
4. Choose **File > Open Template**.
5. Select the `SysMLreport.tpl` and use it to produce a report for your model. You might want to change the template to meet your specific needs.

Report layout

The main elements of each section are shown along with their page locations and are hyperlinked. The generated report contains the following sections covering the complete SysML profile:

- ◆ Requirements diagrams
- ◆ Use case diagrams
- ◆ Sequence diagrams
- ◆ Structure diagrams
- ◆ Object model diagrams
- ◆ Internal and External block diagrams
- ◆ Parametric diagrams
- ◆ Data dictionary
- ◆ Model configuration

The report template uses the standard SysML features built into your model when you select the SysML project **Type** when you first created your project. This selects the SysML profile with the predefined diagrams needed for systems designers.

Requirements diagrams

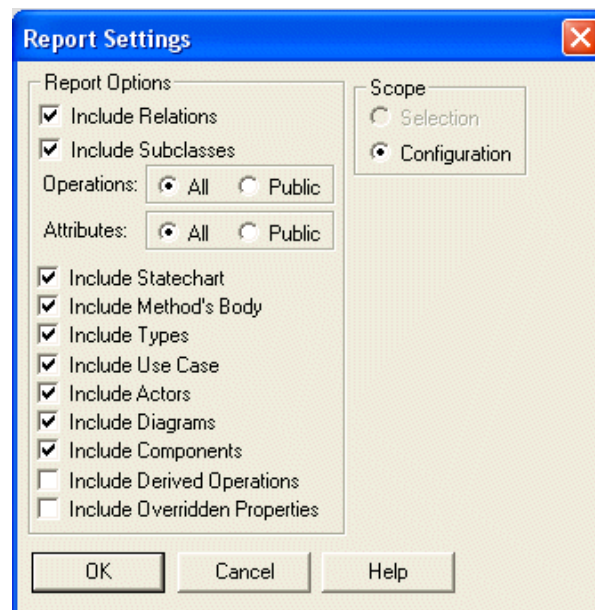
For any requirements diagrams, the `SysMLreport.tpl` supplies hyperlinks to the location of the definition of any use cases, actors, packages, classes or blocks shown in the diagram. Each requirement must have a **Stereotype** setting so that the reporting feature can extract the requirements data.

The internal reporting facility

The internal reporting facility is particularly useful for quick print-outs that the developer needs to use for debugging the model. The reports are not formatted for formal presentations.

Producing an internal report

To create a report using the simple, internal reporter, select **Tools > Report on mode**. The Report Settings window opens, as shown in the following figure.



The window contains the following fields:

- ♦ **Report Options** specifies which elements to include in the report. The possible values are as follows:
 - **Include Relations** include all relationships (associations, aggregations, and compositions). By default, this check box is selected.
 - **Include Subclasses** list the subclasses for each class in the report. By default, this check box is selected.
- ♦ **Scope** specifies the scope of the report. The possible values are as follows:
 - **Selection** includes information only for the selected elements.
 - **Configuration** includes information for all elements in the active component scope. This is the default value.

- ♦ **Operations** specifies which operations to include in the report. The possible values are as follows:
 - **All** includes all operations. This is the default value.
 - **Public** includes only the public operations.
- ♦ **Attributes** specifies which attributes to include in the report. The possible values are as follows:
 - **All** includes all attributes. This is the default value.
 - **Public** include only the public attributes.
- ♦ **Include Statechart** means if the project has a statechart, this option specifies whether to list the states and transitions in the report. By default, this check box is selected.
- ♦ **Include Method's Body** specifies whether to include the code for all method bodies in the report. By default, this check box is selected.
- ♦ **Include Types** specifies whether to list the types in the report. By default, this check box is selected.
- ♦ **Include Use Case** specifies whether to list the use cases in the report. By default, this check box is selected.
- ♦ **Include Actors** specifies whether to list the actors in the report. By default, this check box is selected.
- ♦ **Include Diagrams** specifies whether to include diagram pictures in the report. By default, this check box is cleared.
- ♦ **Include Components** specifies whether to include component information (configurations, folders, files, and their settings) in the report. By default, this check box is cleared.
- ♦ **Include Derived Operations** specifies whether to include generated operations (when the property `CG:CGGeneral::GeneratedCodeInBrowser` is set to Checked). By default, this check box is cleared.
- ♦ **Include Overridden Properties** specifies whether to include properties whose default values have been overridden. By default, this check box is cleared.

To generate the report, select the appropriate values, then click **OK** to generate the report. The report is displayed in the drawing area with the current file name in the title bar.

Setting the RTF character set

For RTF output from the Rational Rhapsody internal reporter, you can define the necessary character set using the `General::Report::RTFCharacterSet` property.

This character set is used in the RTF multi-language and description styles for the **Name Label** and **Description** fields of the report. The RTF file created by the Rational Rhapsody internal reporter must be included as a specific character set for each language. For example, set this property can be set to `\fcharset128` for Japanese.

The default value (an empty string) preserves the current behavior.

Using the internal report output

When you generate a report in Rational Rhapsody using **Tools > Report on model**, the initial result uses the internal RTF viewer. To facilitate the developer's research, this output can be used in the following ways:

- ◆ To print the initially generated report, choose **File > Print**.
- ◆ To locate specific items in the report online, choose **Edit > Find** and type in the search criteria.
- ◆ The initially generated report is only a view of the RTF file that the facility created. This file is located in the project directory (parallel to the `.rpy` file) and is named `RhapsodyRep<num>.rtf`. If you want, open the RTF file using a word processor that handles RTF format, such as Microsoft Word.

Java-specific issues

This section covers Java-specific issues including Javadoc and the Rational Rhapsody JavaDocProfile.

Generation of Javadoc comments

Rational Rhapsody provides a mechanism for including Javadoc comments when code is generated for models developed in Rational Rhapsody for Java.

In general, the Javadoc generation mechanism is based on the following items:

- ♦ Rational Rhapsody properties called `DescriptionTemplate` for elements such as classes and operations. The content of these properties determines the appearance of the generated Javadoc comments.
- ♦ Automatic retrieval of Rational Rhapsody element fields that correspond to Javadoc tags, such as descriptions and operation arguments
- ♦ Special tags and corresponding keywords that can be used for standard Javadoc tags that do not have corresponding Rational Rhapsody elements, for example, `@version`.

Including Javadoc comments in Rational Rhapsody-generated code

Javadoc comment generation is available, by default, for Java developers' Rational Rhapsody projects.

To have Javadoc comments included, just generate code as you normally would.

In the generated code, you should see Javadoc comments based on the descriptions you have provided for model elements. Comments for operations will also include any descriptions you have provided for operation arguments.

If you do not see such comments in your code, open the Features window for the configuration you are using, and verify that the **Generate JavaDoc Comments** check box on the **Settings** tab is selected. If this option is selected, and you still do not see Javadoc comments in your generated code, read the suggestions listed in [Javadoc troubleshooting](#).

In addition to generating these basic Javadoc comments, you can have Rational Rhapsody include the following standard Javadoc tags: author, deprecated, return, see, since, version. To include these Javadoc tags in your generated code:

1. Open the Features window for a model element that you want to document.
2. Add Javadoc content by providing values for the various tags displayed on the **Tags** tab.
3. Repeat this process for each model element you want to document.
4. Generate the code.

Once your code includes Javadoc comments, you can generate a Javadoc report using the standard Javadoc process (see [Javadoc Tool home page](#)).

Changing the appearance of Javadoc comments in generated code

The appearance of Javadoc comments in the generated code is determined by the documentation templates defined using the various `DescriptionTemplate` properties.

To change these templates:

1. Examine the content that `JavaDocProfile` provides for the various `DescriptionTemplate` properties.
2. Modify the values of these properties to match the appearance you would like.

When making changes to these properties, keep in mind the following items:

- ♦ Use new lines to indicate where you would like Rational Rhapsody to begin a new line. You can only see such line breaks by using the `..` button to open the text editor for the property.
- ♦ As you will notice in the default content that `JavaDocProfile` provides for the `DescriptionTemplate` properties, you can use the characters `[ ]` in your template definitions. If you enclose part of the definition in these brackets, Rational Rhapsody will only generate the relevant Javadoc tag, for example, `@version`, if content has been provided for that tag for the element in question.

Enabling/disabling Javadoc comment generation

For new Java projects, Javadoc comments are generated automatically.

To disable Javadoc generation:

1. Open the Features window for the relevant configuration in your model.
2. On the **Settings** tab, clear the **Generate JavaDoc Comments** check box.

Note

When you clear/select the **Generate JavaDoc Comments** check box, it changes the value of the boolean property `CG::Configuration::UseDescriptionTemplates`.

To enable the generation of Javadoc comments for existing projects:

1. Open the project in Rational Rhapsody.
2. Add the JavaDocProfile to your project.

"Built-in" keywords

The content that JavaDocProfile provides for the `DescriptionTemplate` properties contains the following keywords that do not have corresponding Rational Rhapsody tags in the profile.

- ◆ `$Description` represents the description of the corresponding model element.
- ◆ `$Name` represents the name of the corresponding model element.
- ◆ `$Arguments` represents the Javadoc content generated for operation arguments on the basis of the property `JAVA_CG::Argument::DescriptionTemplate`.

Description templates in JavaDocProfile

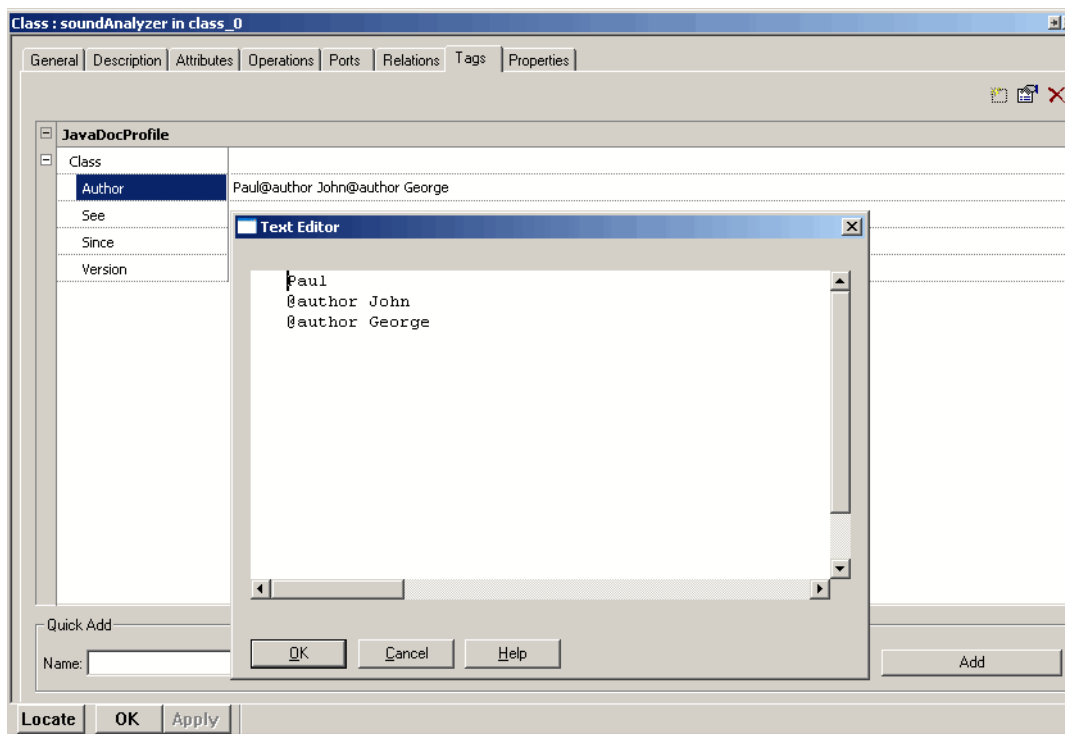
JavaDocProfile provides Javadoc templates for the following properties:

- ◆ `JAVA_CG::File::Header`
- ◆ `JAVA_CG::Package::DescriptionTemplate`
- ◆ `JAVA_CG::Class::DescriptionTemplate`
- ◆ `JAVA_CG::Event::DescriptionTemplate`
- ◆ `JAVA_CG::Attribute::DescriptionTemplate`
- ◆ `JAVA_CG::Relation::DescriptionTemplate`
- ◆ `JAVA_CG::Operation::DescriptionTemplate`
- ◆ `JAVA_CG::Argument::DescriptionTemplate`

Multiple appearance of Javadoc tags

You might want to have certain Javadoc tags appear a number of times for a single element. For example, you might want to have `@author` appear a number of times for a single class that was written by a number of individuals.

In such a case, when assigning a value to the relevant Rational Rhapsody tag, add `@author` before each name except for the first.



Adding new Javadoc tags

The Javadoc generation mechanism allows you to define new Javadoc tags that you would like to use. To define a new Javadoc tag:

1. Create a writable copy of the JavaDocProfile profile by selecting **File > Add to Model**, and then selecting the file <Rational Rhapsody installation path>\Share\Profiles\JavaDoc\JavaDocProfile.sbs (When Rational Rhapsody indicates that the profile already exists in the model, select the **Replace Existing Unit** option.)
2. Create a new Rational Rhapsody tag in your profile. When you create the new tag, select the appropriate item from the **Applicable To** list.
3. Modify the value of the DescriptionTemplate property for the relevant type of element. Use \$<tagname> to have Rational Rhapsody include the tag text in the Javadoc comment.
4. Open the Features window for specific elements of the relevant type, and on the **Tags** tab provide a value for the new Rational Rhapsody tag you have added.

Example:

1. Create under your profile a new tag called codeReviewer. From the Applicable to list, select **Class**.

2. For the `JAVA_CG::Class::DescriptionTemplate` property, add the following to the property value: `[[ * @codeReviewer $codeReviewer]]`
3. For one or more of the classes in your model, open the Features window, and on the **Tags** tab, enter **Steve** for the value of the tag `codeReviewer`.

When you generate code for your model after these changes, the Javadoc comments for these classes will include `@codeReviewer Steve`.

Javadoc handling in reverse engineering and roundtripping

When code with Javadoc comments is reverse engineered, all of the Javadoc comments will be made part of the description of the element.

The Rational Rhapsody roundtripping feature does not support changes to Javadoc comments.

Javadoc troubleshooting

If the **Generate JavaDoc Comments** check box on the configuration **Settings** tab is selected and you still do not see Javadoc comments in your generated code:

1. Verify that the `JavaDocProfile` is loaded in your model.
2. Confirm that the relevant properties are not overridden at some level (`JAVA_CG::File::Header`, the `DescriptionTemplate` properties, and `CG::Configuration::UseDescriptionTemplates`).

Static import

The static import construct was introduced in J2SE 5.0 in order to allow unqualified access to static members of a class. Beginning with version 7.2, Rational Rhapsody is capable of modeling static imports and generating appropriate code. In addition, the reverse engineering feature can handle static imports in Java code, and the roundtripping feature can handle changes to static import statements.

Rational Rhapsody allows you to model both static import of individual class members (`import static java.lang.Math.PI`) and static import of all static members of a class (`import static java.lang.Math.*`).

Modeling of static imports is based on use of the **StaticImport** stereotype in the `PredefinedTypesJava` package. The **StaticImport** stereotype inherits from the **Usage** stereotype.

Adding static imports to a model

To add a static import to your model:

1. Create a dependency in the browser or by drawing a dependency in an object model diagram. The dependency can be from a class to a class or from a class to an individual static attribute or operation.
2. Open the Features window for the dependency you created, and apply the *StaticImport* stereotype to it.

When you next generate code, the code for the dependant class will contain the appropriate static import statement.

Reverse engineering/roundtripping and static import statements

If you reverse engineer code that contains static import statements, Rational Rhapsody will create dependencies that have the **StaticImport** stereotype applied to them.

The roundtripping feature can handle the addition of static import statements to your code, as well as changes to static import statements, including switching regular import statements to static import statements, and vice versa.

If you delete static import statements from your code, the roundtripping behavior will depend upon the value of the property `JAVA_Roundtrip::Update::AcceptChanges`.

Code generation checks

If you create a **StaticImport** dependency between a class and an attribute/method that is not static, Rational Rhapsody will generate the corresponding static import statement but it will issue a warning that you have specified a static import for a non-static class member.

Static blocks

Java allows you to define blocks of code as *static*. Code within static blocks is executed only once, when the class is first loaded.

Rational Rhapsody allows you to add static blocks to classes in your model, and generates appropriate code for such blocks.

Adding static blocks to classes in a model

To add a static block to a class:

1. Right-click the class in the browser and select **Add New > StaticBlock**. (Alternatively, right-click the class in an object model diagram and select **New StaticBlock**.)
2. Open the Features window for the newly created static block, and on the **Implementation** tab enter the code for the body of the block.

Changing a static block to an operation

Rational Rhapsody makes it easy to switch a static block to an operation, and vice versa.

To change a static block to an operation, right-click the static block in the browser and select **Change To > Primitive Operation**.

Note

When you change a static block to an operation, the operation created will be a static operation.

To change a primitive operation to a static block, right-click the operation in the browser and select **Change To > Static Block**.

Reverse engineering/roundtripping and static blocks

If you reverse engineer code that contains static blocks, Rational Rhapsody recognizes these blocks and adds them to the class in the model.

The roundtripping feature can handle the addition of new static blocks to your code, as well as changes to the body of a static block.

When making changes directly to code within a static block, keep in mind that when adding code to the body of a static block, the new code will be roundtripped into the model only if you placed the code between the Rational Rhapsody annotations inside the block.

If you delete static blocks from your code, the roundtripping behavior will depend upon the value of the property `JAVA_Roundtrip::Update::AcceptChanges`.

Generating JAR files

In Rational Rhapsody for Java developers, you have the option of specifying that Rational Rhapsody should generate a JAR file when you build your project.

To specify that a JAR file should be created as part of the build process:

1. Open the Features window for the relevant configuration.
2. On the **Settings** tab, select the **Generate JAR File** option.

The JAR file generation mechanism is controlled by the following properties (under `JAVA_CG::Configuration`):

- ♦ `JarFileGenerate` is a Boolean property that determines whether or not a JAR file will be generated as part of the build process. The value of this property is controlled by the **Generate JAR File** option on the **Settings** tab of the Features window for configurations.
- ♦ `JarFileGeneratorCommand` specifies the jar command that should be carried out if the property `JarFileGenerate` has been set to Checked.

Java 5 annotations

Rational Rhapsody for Java developers supports the concept of the Java 5 annotation through modeling and code generation. Java users can use Java annotations to model and generate code for all key Java 5 concepts. You can create annotations within the Rational Rhapsody environment and then generate the annotations within the generated code.

Note the following about Java annotations:

- ◆ They provide data about the program but do not affect the program itself.
- ◆ They can be used by:
 - compilers
 - documentation tools
 - code analysis tools
 - deployment tools
 - run-time analysis tools
- ◆ They can be applied on any kind of program element (for example, class, field, method, enum, and so on).

To add a JavaAnnotation to a model element, you must do the following general steps:

1. Create a JavaAnnotation type; see [Creating a JavaAnnotation type](#).
2. Add the JavaAnnotation and assign values to the annotation's elements; see [Using a JavaAnnotation type](#).
3. Add the annotation to one or more Java model elements using a dependency with a AnnotationUsage relationships; see [Using a JavaAnnotation within a model](#).

Creating a JavaAnnotation type

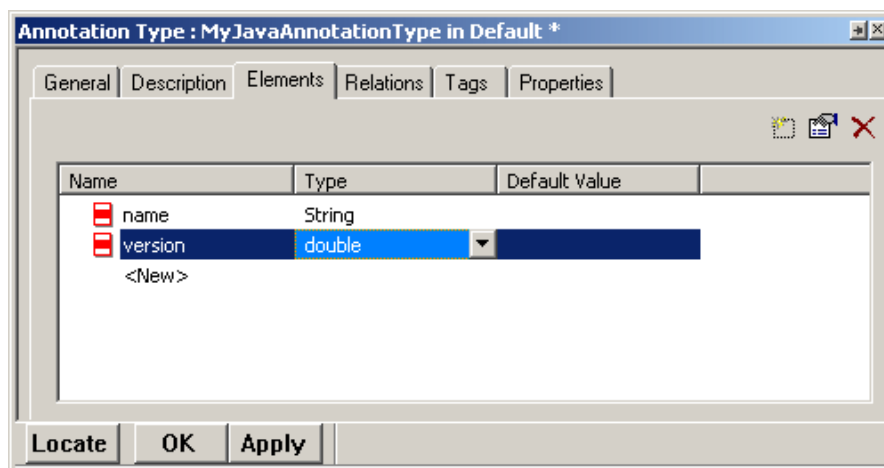
Java 5 annotations are modeled similar to the way classes and objects are modeled. This means you must define this type of annotation before you can use it.

To create a Java annotation type:

1. Open your Rational Rhapsody project in Java, right-click a package or class on the Rational Rhapsody browser, and select **Add New>Annotation Type**.
2. Type a name for your new annotation type.
3. Double-click the annotation type. The Features window opens.
4. On the **Elements** tab, add any legal JDK 5 data type, as shown in the following figure.

- a. Click <<New>> and type a name for the element.
- b. Select a type from the **Type** list.
- c. Enter a default value if necessary.

Note: The rows on this tab relate to annotation elements.



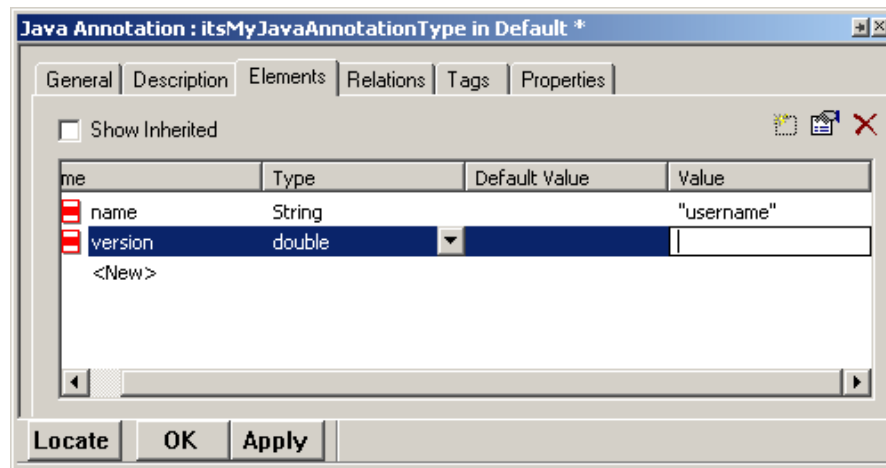
5. Click **OK**.

Using a JavaAnnotation type

To use a JavaAnnotation type, you must create a JavaAnnotation that is of the type you want. To do so:

1. Open your Rational Rhapsody project in Java, right-click a package on the Rational Rhapsody browser and select **Add New > JavaAnnotation**. The Add JavaAnnotation window opens.
2. Select the JavaAnnotation type from the list.
3. Click **OK**.
4. Double-click the JavaAnnotation on your Rational Rhapsody browser. The Features window opens.

5. On the **Elements** tab, enter specific values for the JavaAnnotation.



6. Click **OK**.

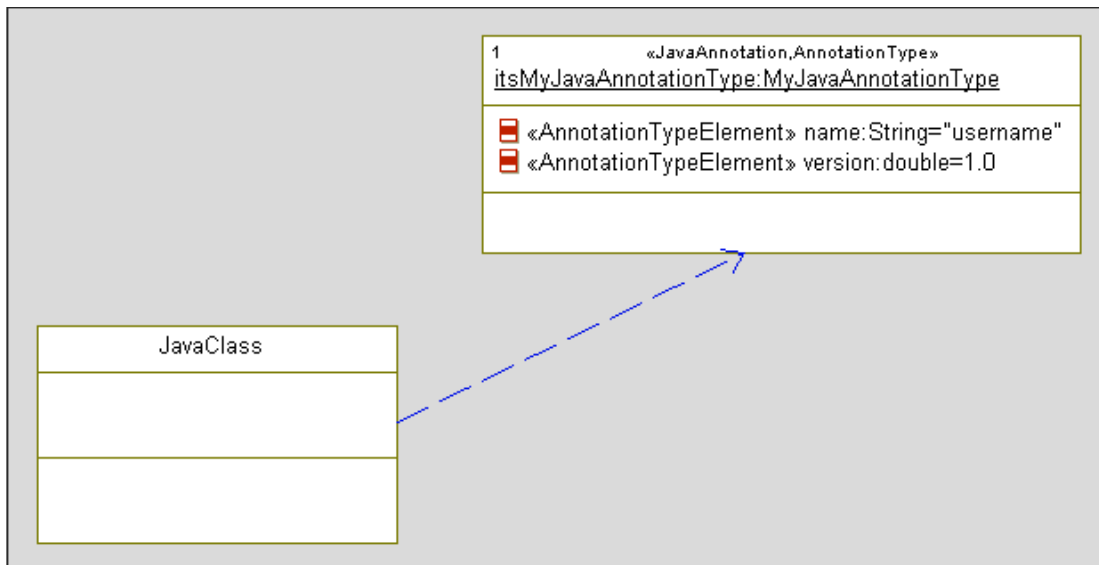
Using a JavaAnnotation within a model

To have classes in your Java design use a JavaAnnotation:

1. Open your Rational Rhapsody project in Java.
2. On a diagram (for example, an object model diagram), drag your JavaAnnotation from the Rational Rhapsody browser onto your diagram.

Notice that from a model perspective, the JavaAnnotation is shown with two stereotypes.

3. Create a new class on your diagram that represents a user-defined Java class in the system that wants to make use of the JavaAnnotation.
4. Draw a dependency from your class to your JavaAnnotation. Your diagram should resemble something like the following figure:



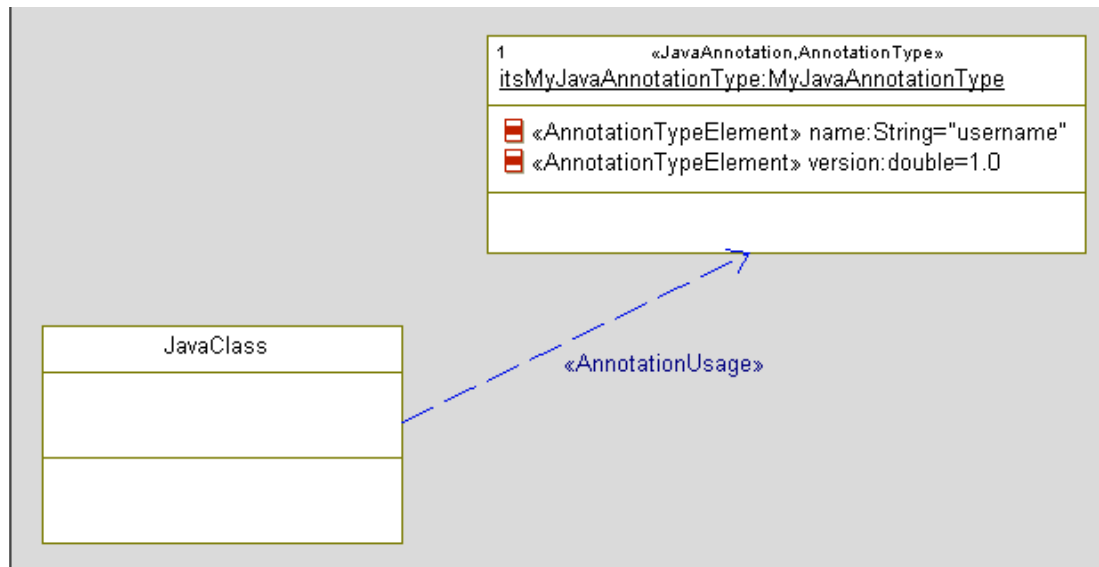
5. Double-click the Dependency link. The Features window opens.
6. On the **General** tab, from the **Stereotype** list, select **AnnotationUsage in PredefinedTypesJava**.

Using this stereotype ensures that the code is generated correctly.

Note: **AnnotationUsage** displays in the **Stereotype** box.

7. Click **OK**.

Your diagram should resemble something like the following figure:



The following figure shows how the annotation is used within the Java class shown above:

```
@MyJavaAnnotationType {
    name = "username",
    version = 1.0
}
///

```
class JavaClass
public class JavaClass {

 // Constructors

 ///

```
auto_generated
    public JavaClass() {
    }

}
/*****
File Path : DefaultComponent/DefaultConfig/Default/JavaClass.java
*****/
```


```


```

Code generation and Java 5 annotations

The code generator interprets the terms `AnnotationType`, `JavaAnnotation` and `AnnotationUsage` to print the corresponding Java code.

In addition, Java annotations of some element can be added as text to the `JAVA_CG::<ElementType>::JavaAnnotation` property. The code generator will print the property content before element declaration.

Reverse engineering and Java 5 annotations

To assure reverse engineering works correctly for Java annotations, note the following information:

- ♦ To control the behavior of reverse engineering, use the `JAVA_ReverseEngineering::ImplementationTrait::ImportJavaAnnotation` property. The following values are available for this property:
 - **None.** All code parts related to Java Annotation are ignored
 - **Model.** Java annotations are imported as model elements (`AnnotationType`, `JavaAnnotation` and `AnnotationUsage`)
 - **Verbatim.** Java annotation Usage is imported as a verbatim text to `JavaAnnotation` property of the corresponded element. `AnnotationTypes` are imported as model elements. This is the default value.
 - **Mixed.** Java annotation are imported as model elements; if this fails, usage will still be imported as a verbatim text to `JavaAnnotation`.
- ♦ Specify a correct `CLASSPATH` for reverse engineering is important for the correct result of Reverse Engineering of Java Annotations.
- ♦ When you use both schemes of Reverse Engineering of Java Annotations (Model and Verbatim) the code generated for the model created by RE should be the same as in the original code (semantically).
- ♦ Roundtrip completely ignores all code parts related to Java annotations. Nothing should be changed.

Limitations for Java 5 annotations

Note the following limitations:

- ◆ There is no roundtripping of Java annotations.
- ◆ Java annotations of events are not supported
- ◆ Predefined Java annotations are not supported.
- ◆ There is no **Default Value** field on the Features window for the annotation element (double-click the annotation element on the Rational Rhapsody browser to open this Features window). Instead, you can add it on the **Elements** tab of the Features window for the Annotation type.
- ◆ Java annotations of constructor, destructor (finalize), and associations are not generated by the code generator. Instead, use the `JAVA_CG:Operation::JavaAnnotation` and `JAVA_CG:Relation::JavaAnnotation` properties.
- ◆ Unnamed types in Java annotations are not supported (without “element = ...”), for example: “@Retention(RetentionPolicy.RUNTIME)”. Instead, use the `JAVA_CG:Operation::JavaAnnotation` property.

Java reference model

Rational Rhapsody includes a reference model for the classes contained in Java SE 6.

When you create a new Rational Rhapsody project in Java, you can add this model to your project as a reference.

The Java reference model can be found in the directory <Rational Rhapsody installation path>\Share\LangJava\JDKRefModel.

Systems engineering with Rational Rhapsody

Rational Rhapsody allows systems engineers to capture and analyze *requirements* quickly and then design and validate system behaviors. A Rational Rhapsody systems engineering project includes the UML and SysML diagrams, packages, and simulation configurations that define the model. Systems engineers can use the [SysML profile features](#) and/or the [Harmony process and toolkit](#) to guide system development through its iterative development process.

Installing and launching systems engineering

Follow the instructions in the Rational Rhapsody installation instructions to set up the development environments you need. The Rational Rhapsody Systems Engineering Add On requires these extra installation steps and start-up steps:

1. When the **Add-on Installation** window displays. Select the **Systems Engineering Add-On** check box.
2. Click **Next** and complete the installation as instructed.
3. When you want to use the systems version of Rational Rhapsody, from the Windows Start menu, select **All Programs > IBM Rational > IBM Rational Rhapsody version number > Rational Rhapsody Designer for Systems Engineers > Rhapsody**.

This installation makes the systems engineering features available to support the UML and SysML standards in these specifications:

- ♦ Check the URL for the [UML specification](#)
- ♦ Check the URL for the [SysML specification](#)

Creating a SysML profile project

To create a new systems engineering project using the [SysML profile features](#):

1. Start Rational Rhapsody.
2. Click the **New** button  on the main toolbar or select **File > New**.
3. In the **Project name** box, type your project name.
4. In the **In folder** box, enter the directory in which the new project will be located, or click the **Browse** button to select the directory.
5. In the **Project Type** box, select the SysML profile so that you can use the SysML modeling language and the systems engineering diagrams.
6. You might also want to select one of the **Project Settings**.
7. Click **OK**. Rational Rhapsody verifies that the specified location exists. If it does not, Rational Rhapsody asks whether you want to create it.
8. Click **Yes**. In this example, Rational Rhapsody creates a new project in the selected subdirectory, opens the project, and displays the browser in the left pane.

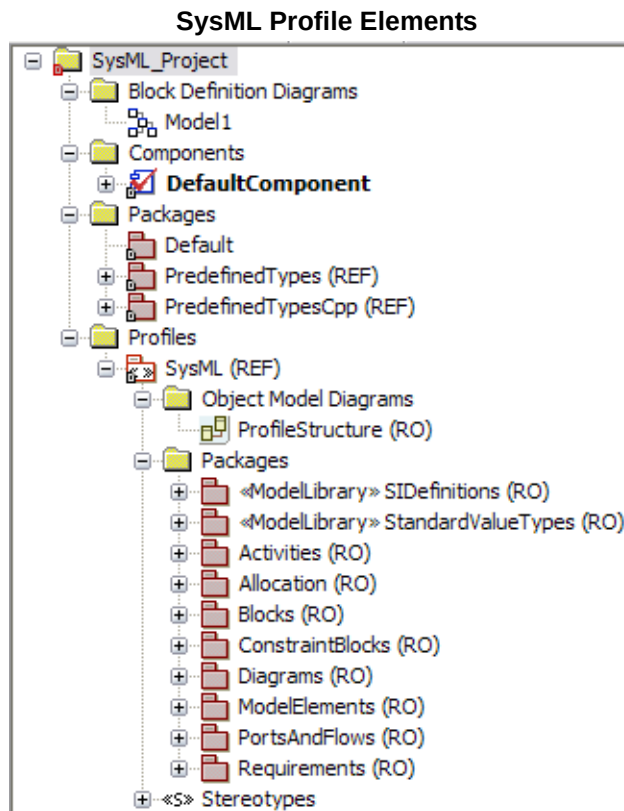
Note

If the browser does not display, select **View > Browser**.

SysML profile features

When you select the SysML profile for your project, Rational Rhapsody provides a starting point with a blank Block Definition Diagram (named Model1), packages, and predefined types, as shown in [SysML Profile Elements](#). This profile is the Rational Rhapsody implementation of the [OMG SysML Specification](#). The Rational Rhapsody SysML profile provides this additional functionality for your model:

- ◆ SysML enhancements to standard UML diagrams including the Use Case, Requirements, Activity, Sequence diagrams and Statecharts
- ◆ SysML Block Definition, Internal Block, and Parametric diagrams
- ◆ XMI 2.1 support



The SysML profile also contains default and predefined packages and a read-only ProfileStructure object model diagram for you to use as reference of the available Rational Rhapsody SysML features in the profile.

Note

The items listed under **Profiles** in the browser are not intended to be used as part of a working model. They are for information purposes only.

When you create a new project, Rational Rhapsody creates a directory containing the project files in the specified location. The name you choose for your new project is used to name project files and directories, and displays at the top level of the project hierarchy in the Rational Rhapsody browser. Rational Rhapsody provides several default elements in the new project, including a default package, component, and configuration.

SysML profile packages

The following Rational Rhapsody packages (shown in [SysML Profile Elements](#)) are available when you select the SysML profile for a new project:

- ♦ **<<ModelLibrary>> SIDefinitions** contains these read-only packages: BaseSIUnits and DerivedSIUnits.
- ♦ **<<ModelLibrary>> StandardValueTypes** contains read-only Complex and Real value types. For more information about valueTypes, see [Adding graphics to block definition diagrams](#).
- ♦ **Activities** stereotypes support the SysML expansion of the activity diagram behaviors and links to the blocks that contain the behaviors.
- ♦ **Allocation** contains read-only stereotypes and table layouts.
- ♦ **Blocks** include stereotypes that represent the system capabilities in the model.
- ♦ **ConstraintBlocks** contains the stereotypes that control the relationships of blocks: ConstraintBlock, ConstraintParameter, ConstraintProperty, and valueBinding.
- ♦ **Diagrams** lists the stereotypes needed to support these SysML diagrams: Block Definition, Internal Block, Parametric, and Requirements.
- ♦ **ModelElements** lists the stereotypes needed to support these diagram elements (see [Adding elements](#)):
 - Conform
 - Problem
 - Rationale
 - Refinement
 - View (see [Views and viewpoints](#))
 - Viewpoint with its tags
- ♦ **PortsandFlows** show how items flow between blocks and parts. Ports are the connection points between blocks or parts and their environments. Ports are often reused and have clearly defined interfaces. These are most often used in [Activity modeling in SysML](#).

- ◆ **Requirements** contains the stereotypes to display model conditions that must be met with the finished product. This profile element also includes read-only requirement table layouts.

Views and viewpoints

A Rational Rhapsody *View* is a representation of a system or subsystem in a Rational Rhapsody package. It allows the designers to focus on specific aspects of the system that are important to them. For example, they want to define the security system within a manufacturing system for a new factory or the fuel efficiency components of a new engine.

A Rational Rhapsody *Viewpoint* defines the rules and conventions to address the requirements for a View. For example, the Viewpoint for a factory security system might include security requirements, the security functional and physical architecture, and the security test cases.

Creating a view

To add a View to the project:

1. Highlight the system or subsystem for which a View is required.
2. Right-click and select **Add New > General Elements > View**.
3. Type the name of the new View into the browser location created.
4. Open the Features window to define the newly added View.

The View can be dragged from the browser onto the diagrams.

Note

You can change an existing package into a View using **Change To > View**.

Adding a viewpoint

To add a Viewpoint to a View:

1. Highlight the View.
2. Right-click and select **Add New > General Elements > Viewpoint**.
3. Type the name of the new Viewpoint into the browser location created.
4. Open the Features window to define the Viewpoint.

You might use the tags in the SysML profile to define your Viewpoint:

- ◆ Concerns
- ◆ Languages
- ◆ Methods
- ◆ Purpose
- ◆ Stakeholders

Adding elements

To add a new element:

1. Highlight the item in the browser to which you are adding the new element.
2. Select **Add New > General Elements** and select the element type from the possibilities for the selected item:
 - ◆ **Comment** is a textual annotation that does not add semantics, but contains useful information
 - ◆ **Constraint** shows restrictions associated with one or more model elements as a logical constraint, a condition on a decision branch, or a mathematical expression
 - ◆ **Problem** describes an unfavorable environment situation that needs to be addressed
 - ◆ **Rationale** states the reason for a specific requirement or design feature
 - ◆ **Viewpoint** specifies the rules and conventions for constructing a View to address a set of stakeholder concerns.
 - ◆ **Conform** details compliance with the Viewpoint rules and conventions.
 - ◆ **Dependency** shows a relationship
 - ◆ **Refinement** describes how a model element or set of elements can be used to further define a requirement.

- ♦ **Realization** specifies the relationship (as a super class) between an interface and a class that implements that interface

3. To create the new element:

- ♦ For the Comment, Constraint, Problem, Rationale, and Viewpoint elements, type the name of the new element into the browser location created.
- ♦ For the Conform, Dependency, and Refinement, select the element on which it depends and click **OK**.
- ♦ For the Realization, select the class to become a super class of the selected browser item and click **OK**.

4. Open the Features window to define the newly added element.

These new elements might be dragged from the browser onto the diagrams.

Note

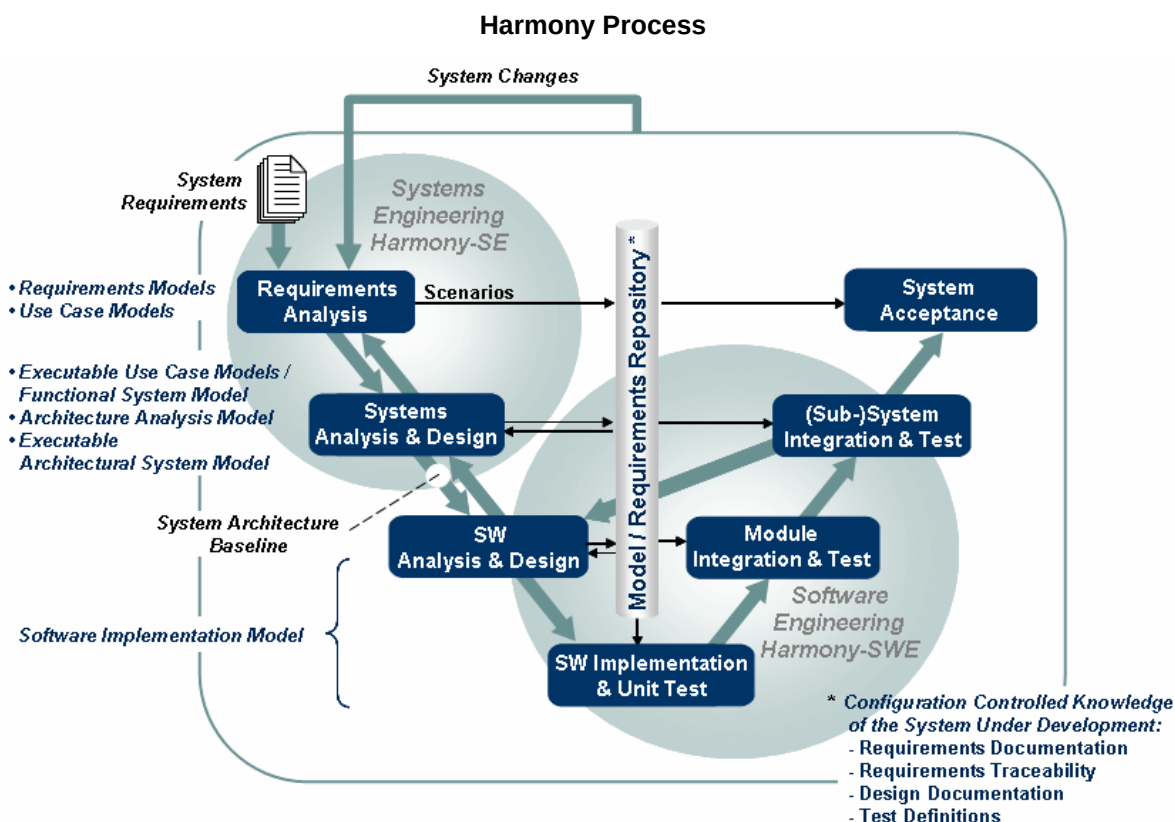
Elements can be removed from a View or deleted from the model entirely.

Harmony process and toolkit

The Harmony process facilitates a seamless transition from systems engineering to software engineering. It uses SysML exclusively for system representation and specification. Harmony, a scenario-driven process, is iterative and promotes reuse of test scenarios throughout system development, as shown in the [Harmony Process](#) diagram.

Harmony process summary

The Harmony process models allow systems engineers to find design errors early in the development when the cost of correcting them is lower. Customer requests can be more efficiently assessed, incorporated, and given timely feedback. However, the greatest benefit of a model-driven process is improved communication, not only between the engineering disciplines, but also among the technical and non-technical parties involved in the system development process. This is possible because models can represent different levels of abstraction and, therefore, avoid the information overload that often occurs when data is passed among the participating groups.



The Harmony process can be used in any systems engineering project. The key objectives of these projects are as follows:

- ◆ Derive required system functionality
- ◆ Identify system states and modes
- ◆ Allocate requirements and functionality to identified subsystems

These key objectives can be met in the UML structure diagrams and the following SysML diagrams:

- ◆ Use case diagrams
- ◆ Sequence diagrams
- ◆ Activity diagrams
- ◆ Statecharts
- ◆ Block definition diagrams
- ◆ Internal block diagrams
- ◆ Parametric diagrams

Creating a Harmony project

Though the Harmony process can be used in any systems engineering project, Rational Rhapsody provides a special profile to make it easier to use this process in a project.

To create a Harmony project for systems engineering:

1. Start Rational Rhapsody.
2. Click the **New** button  in the main toolbar or select **File > New**.
3. In the **Project name** box, type your project name.
4. In the **In folder** box, enter the directory in which the new project will be located, or click the **Browse** button to select the directory.
5. In the **Project Type** box, select the Harmony profile so that you can use the Harmony wizards and other systems engineering features.
6. You might also want to select one of the **Project Settings**.
7. Click **OK**. Rational Rhapsody verifies that the specified location exists. If it does not, Rational Rhapsody asks whether you want to create it and generates a starting point for your Harmony project.
8. Check the **Profiles** folder in the browser to be certain that Harmony is listed with the stereotypes and tags.

To add the Harmony profile to a SysML profile project, use the standard [Adding a Rational Rhapsody profile manually](#) method.

Creating an activity view

To model activities in much the same manner as a use case, create an activity view for each group of sequences. To create an activity view:

1. Create a project with the Harmony profile.
2. Right-click a package or use case in the browser.
3. Select **Add New > Harmony > Activity View**.
4. Open the Features window to define this view.

The activities in each view are then referenced in activity diagrams. Use the [Special Harmony menu commands](#) to automate operations associated with activity views.

Adding measures of effectiveness (moe)

To add a “moe” (measures of effectiveness):

1. Create a project with the Harmony profile.
2. Right-click a package, use case, actor, interface, or class in the browser.
3. Select **Add New > Harmony > moe**.
4. Open the Features window to define the measures of effectiveness.

The “moe” supports the trade analysis feature. For more information, see [Special Harmony menu commands](#) and [Performing a Trade Analysis](#).

Harmony profile features

The Rational Rhapsody systems engineering features includes the following automated tools for *Harmony profile* projects:

- ♦ SE-Toolkit menu commands to perform common tasks quickly
- ♦ Tools that perform repetitive tasks automatically or reduce the number of steps required to perform a systems engineering operation

Special Harmony menu commands

To access the special systems engineering menu commands:

1. Click an item in the browser that has a special option available.
2. Right-click and select **SE-Toolkit** to open the menu with the special systems engineering menu commands.
3. Click the menu option to perform the task. The following table lists the special menu commands by the browser items used to access them and describes the operations that the menu commands perform.

Special Harmony Menu Commands Chart

Accessible from Browser Item	Menu Command	Systems Engineering Operation Description
Activity Diagram	Auto-Rename Actions	Renames all of the actions in an activity diagram to be the actual action body text used
Activity Diagram	Create New Scenario from Activity Diagram	Creates a new sequence diagram from an existing activity diagram with the swimlanes converted to life lines
Activity Diagram	Perform Swimlane Consistency Check	For every Swimlane that is represented by an object/part, it ensures that • Each action has a corresponding operation • Each operation has a corresponding action
Activity View	Duplicate Activity View	Existing activity view is copied using the name of the view appended with “_copy”
Activity View and Activity Diagram	Create Allocation CSV File	• Creates a CSV file based on the Swimlane Allocation • CVS file is added to Rational Rhapsody as a controlled file and can be viewed in Rational Rhapsody

Accessible from Browser Item	Menu Command	Systems Engineering Operation Description
Activity View and Activity Diagram	Create Allocation Table	Creates an Excel Spreadsheet based on the Swimlane Allocation (requires <i>Microsoft Excel</i>)
Activity View and Use Case	Create Simulation	Create an animated version of the highlighted activity view
Activity View and Use Case	Perform Activity View Consistency Check	Displays a window that lists the following information: - Operations in the activity diagram that do not appear in any of the project's sequence diagrams - Errors found in the referenced scenarios The results displayed in this window can be copied into the Clipboard. There is also a Recheck button.
Block Definition Diagram and Object Model Diagram	Perform Trade Analysis	<i>Microsoft Excel</i> must be available for this feature. The selected diagram must contain blocks/classes that represent the "solution" classes, that is, those classes that aggregate the potential solutions with contained measures of effectiveness, <<moe>> stereotype. For more information, see Performing a Trade Analysis .
Class/Block	Copy MOEs from Base	Copies attributes stereotyped <<moe>> or measure of effectiveness from any parent classes/blocks to all classes/blocks that inherit from it. For each attribute, this option copies the "weight" tag value.
Class/Block	Copy MOEs TO Children	Exhibits the same behavior as the Copy MOEs from Base option but in the other direction (that is, from the Base Class/block to any that inherit from it). For each attribute, this option copies the "weight" tag value.
Internal Block Diagram and Object Model Diagram	Generate N2 Matrix	Generates an N2 Matrix from a Block Diagram and its associated Ports/Interfaces/Links
Class/Block	Create Test Architecture	Uses the <i>TestingProfile</i> to generate a <i>Test Context Diagram</i> for the selected block or class and displays the test messages in the Log output window
Object, Class, or Actor	Create Test Bench	Creates a Test Bench style Statechart for the current Actor

Accessible from Browser Item	Menu Command	Systems Engineering Operation Description
Packages	Generate Initial Statechart(s)	
Package and Sequence Diagram	Create Ports And Interfaces	This option might perform any of these operations depending on what was selected in the browser and available in the model at this point: • Creates new interfaces in the InterfacesPkg to hold them • Creates ports on structural blocks involved • Populates Ports with Interfaces • Makes Ports behavioral • Makes any "internal" operations private • Moves Event Declarations to the InterfacesPkg • Errors are flagged and highlighted in red on the sequences
Sequence Diagram	Report Unrealised Messages	Provides the user with a window showing a list of messages on a sequence diagram which are not yet realized
Use Case	Create Use Case Scenario	Creates a sequence diagram based on a selected use case diagram
Use Case	Create System Model from Use Case	Creates a system model diagram based on a selected use case
Use Case	Setup Model Execution	

Performing a Trade Analysis

For block definition diagrams, you can generate a weighted methods *Microsoft Excel* spreadsheet from the selected diagram. To generate a trade analysis:

1. In the browser, highlight a block definition diagram.

Note: The selected diagram must contain blocks or classes that represent the “solution” classes (that is, those that aggregate the potential solutions with contained MOEs (measures of effectiveness)).

2. Right-click and select **Perform Trade Analysis**. The system generates the spreadsheet or displays a message indicating that the selected diagram is not appropriate for a trade analysis.

If the selected diagram is appropriate, the list of potential solutions is converted into a spreadsheet in a standard weighted methods format. The cells are populated with values, weights, and formulas to calculate totals.

If the diagram does not contain blocks that represent the “solution” classes or cannot be used for the analysis, an error message opens with the explanation of the generation failure.

Architectural Design Wizard

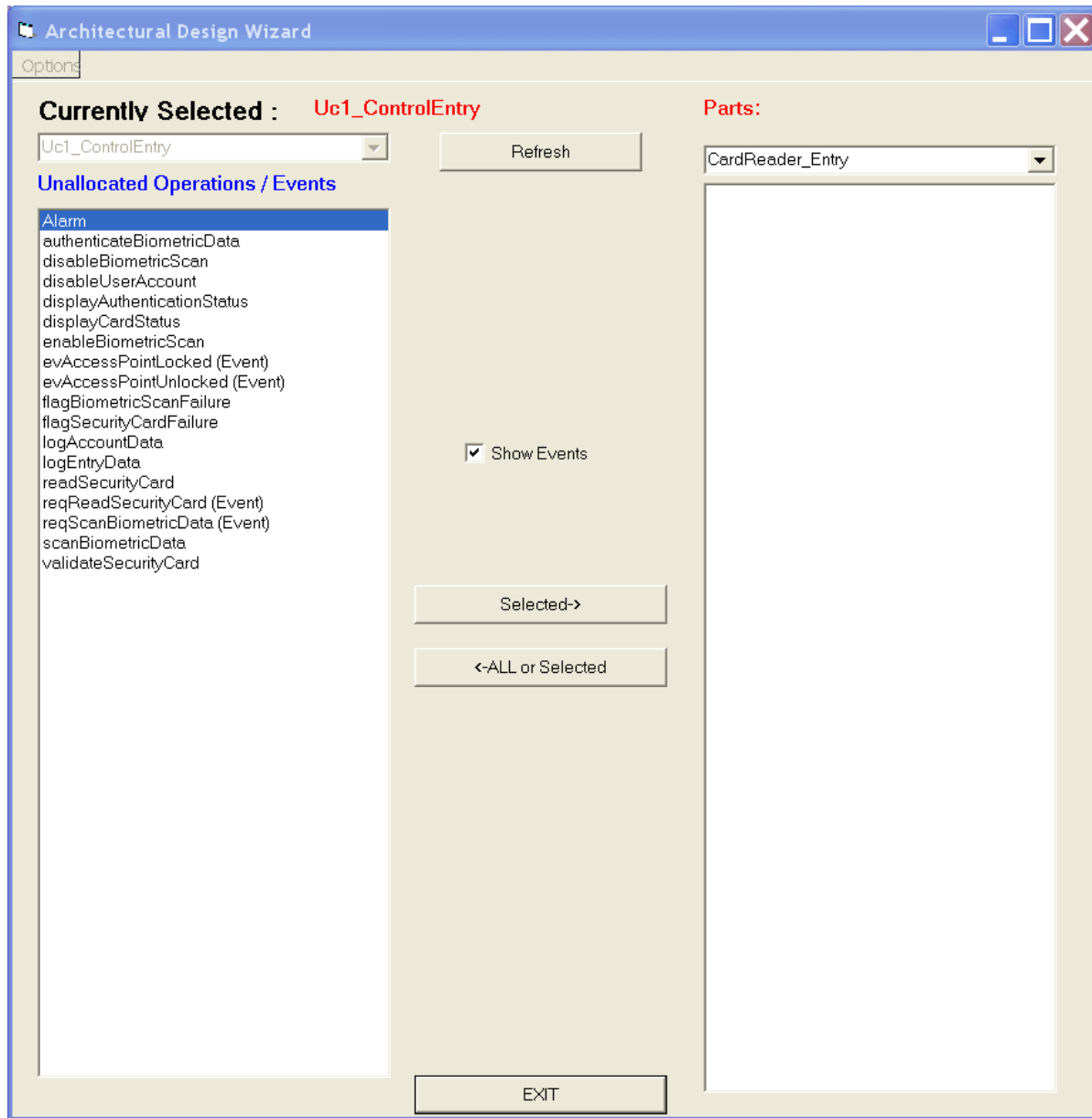
For both the Harmony and SysML profiles, the Architectural Design Wizard is available to copy operations from one architectural layer to another. This allows allocation of operations / events from a parent block to its children by copying and tagging them and provides these features:

- ◆ Allocate the operations (including the documentation and requirement relationships) to their respective systems
- ◆ Track when operations are allocated
- ◆ Allow multiple allocations

To copy unallocated operations/events to specific subsystems:

1. Highlight the block or class in the browser.
2. Right-click and select **SE-Toolkit > Architectural Design Wizard**.
3. If the selected system element does not immediately display, click the **Refresh** button at the top of the window. If you want the events in the selected element to be available for selection, check the **Show Events** box.
4. Use the pull-down menu above the **Parts** column to select one or more subsystems.

5. Highlight the **Unallocated Operation(s) / Event(s)** from the list on the left that you want to allocate to a **Part** listed on the right.



6. If you change your mind and want to return items you put into the **Parts** column back to the **Unallocated** column, highlight individual items in the right column and click the **All or Selected** button. To return all items to the left side quickly, do not select any items and click the **All or Selected** button.

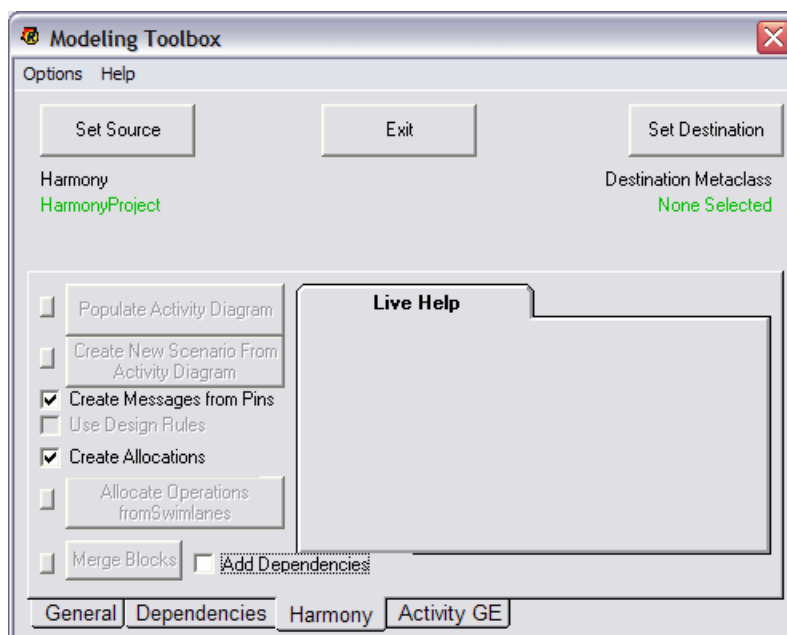
7. Click **Exit** to close the wizard's window.

Modeling Toolbox

The Modeling Toolbox can be used with sequence and activity diagrams to set links between a source and a destination. It allows you to select multiple sources or multiple links and create any type of dependency between them.

To use the automatic Modeling Toolbox:

1. Display the sequence or activity diagram to which you need to add links.
2. Select an element in the diagram that is the starting point for the link.
3. Choose **Tools > SE-Toolkit > Modeling Toolbox**.
4. Click the **Set Source** button and the name of the selected element displays.



5. Click the **Harmony** tab to select one of these options:
 - ♦ **Populate Activity Diagram**
 - ♦ **Create New Scenario from Activity Diagram** (determine whether or not to use the Design Rules)
 - ♦ Create Messages from
 - ♦ Use Design Rules

- ◆ Create Allocations
- ◆ **Allocate Operations from Swimlanes**
- ◆ Merge Blocks

A detailed description of each Harmony command displays in the **Live Help** area when you position the cursor over the small button to the left of the Command button.

6. In the diagram, select the destination elements and then click the **Set Destination** button in the Modeling Toolbox. The toolbox allows these types of links:

- ◆ **Hyperlink(s)**
- ◆ **Anchor(s)**
- ◆ **SD Ref(s)**
- ◆ **Event Reception(s)**
- ◆ **Value Type**

A detailed description of each link type displays when you position the cursor over the small button to the left of the Command button, as shown in the example above. If a button is greyed out, that type of link is not appropriate for the selected Source and Destination. Click the button for the type of link you want to add.

7. Click the **Dependencies** tab to create a stereotype of the selected type automatically. For a **User** defined stereotype, click **User** and type the name. Click **Create Dependency** when you have made your stereotyping selections.
8. Click the **Activity GE** tab to select either a Reference Activity or a Swimlane Reference.
9. Click **Exit** at the top of the Modeling Toolbox to perform the selected operations from the source to the destination.

Note

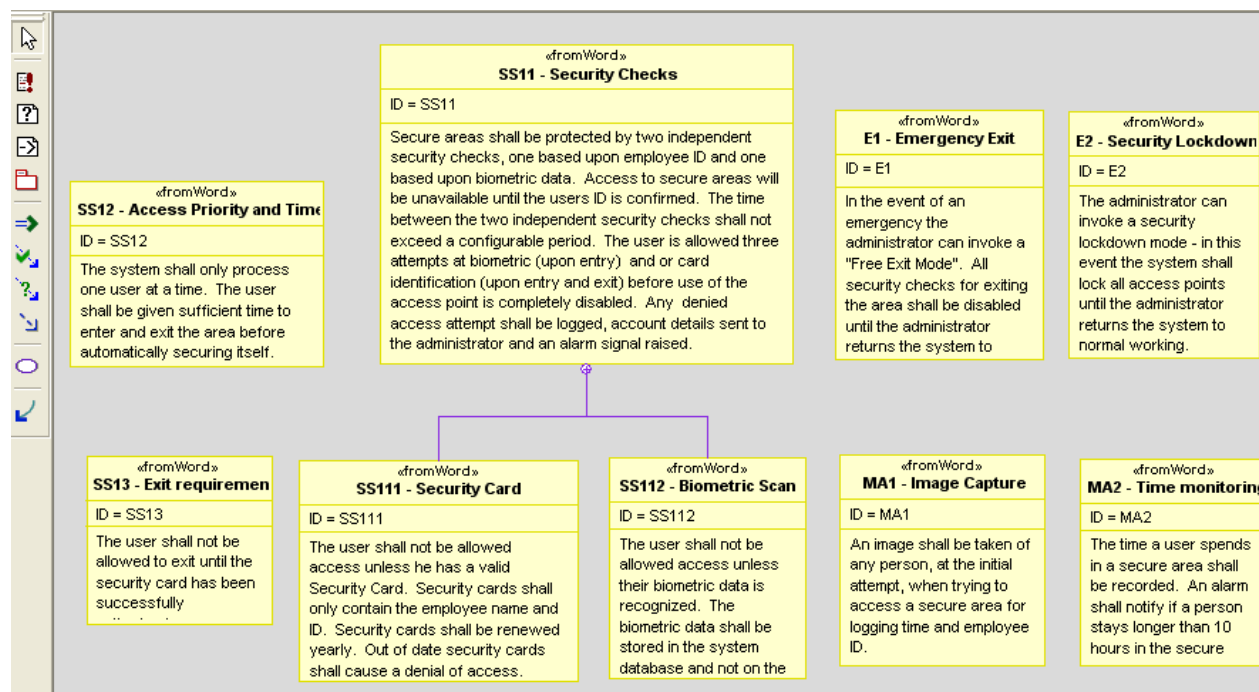
To perform this task manually, see the instructions in the [Systems engineering requirements in Rational Rhapsody](#) section.

Systems engineering requirements in Rational Rhapsody

The Rational Rhapsody Gateway add-on product is often used to define specific requirements to support your analysis. This Add On product allows Rational Rhapsody to hook up seamlessly with third-party requirements and authoring tools for requirements traceability. In addition, it describes importing requirements using the Rational Rhapsody Gateway and using *use case diagrams* (UCDs) to show the main system functions and the entities that are outside the system (actors). The use case diagrams specify the requirements for the system and demonstrate the interactions between the system and external actors. See the [Modeling Toolbox](#) to use the automatic linking feature.

You might also use other manual methods and software to create requirements diagrams in Rational Rhapsody such as importing them from *DOORS* or *Microsoft Word*.

Requirements Diagram with Requirements Imported from Word



Analysis and requirements using the Rational Rhapsody Gateway

When the IBM Rational Rhapsody Gateway analyzes your project information including requirements, documents, and database modules, the software provides the following analysis results:

- ◆ Navigation features between the Rational Rhapsody Gateway and interfaced tools
- ◆ Requirements captured at high level accessible in an authoring tool
- ◆ Filter capabilities for more targeted display and results for reports
- ◆ Requirements traceability graph
- ◆ A list of elements violating default rules and customized rules
- ◆ Additional information within the Rational Rhapsody Gateway environment including attributes, links, and text

Importing Rational Rhapsody Gateway requirements

You can use the Rational Rhapsody Gateway product to define specific requirements to support your analysis.

This Add On product allows Rational Rhapsody to hook up seamlessly with third-party requirements and authoring tools for complete requirements traceability. The Rational Rhapsody Gateway Add On includes the following features:

- ◆ Traceability of requirements workflow on all levels, in real-time
- ◆ Automatic management of complex requirements scenarios for intuitive and understandable views of upstream and downstream impacts
- ◆ Creates impact reports and requirements traceability matrices to meet industry safety standards
- ◆ Connects to common requirements management/authoring tools including DOORS, Requisite Pro®, Word®, Excel®, Powerpoint PDF®, ASCII, FrameMaker, Code and Test files
- ◆ A bidirectional interface with the third-party requirements management and authoring tools
- ◆ Monitoring of all levels of the workflow, for better project management and efficiency

Limitations

This is a limitation in Rational Rhapsody Gateway. If you have multiple Rational Rhapsody projects configured for Rational Rhapsody Gateway, when you right-click and select **Reload** in Rational Rhapsody Gateway, it synchronizes with the active opened Rational Rhapsody project, not the selected project. This creates incorrect data. For example, if you have a Radio and a Handset project configured in Rational Rhapsody Gateway but Radio is set as the active project, you work on the Handset model and select the **Reload** command in Rational Rhapsody Gateway. This would synchronize the Radio project, but not the Handset project because it is not your active project.

Searching requirements

You can search for requirements in the project based on the requirement details such as stereotypes, tags, or text. To locate requirements using the Rational Rhapsody advanced search facility:

1. Select the **Edit > Advanced Search and Replace**.
2. On the **Search elements** tab, you can use the **Clear All** button to remove all of the check marks and then select the specific element or elements you want to research such as Requirement or Tag.
3. You might also want to narrow the search to specific types of information using the **Search in** tab. This allows you to select Name, Description, Requirement ID, Requirement Specification, and Tag Value.
4. On the **Find/Replace** tab, enter the search criteria in **Find what** and click **Find**.

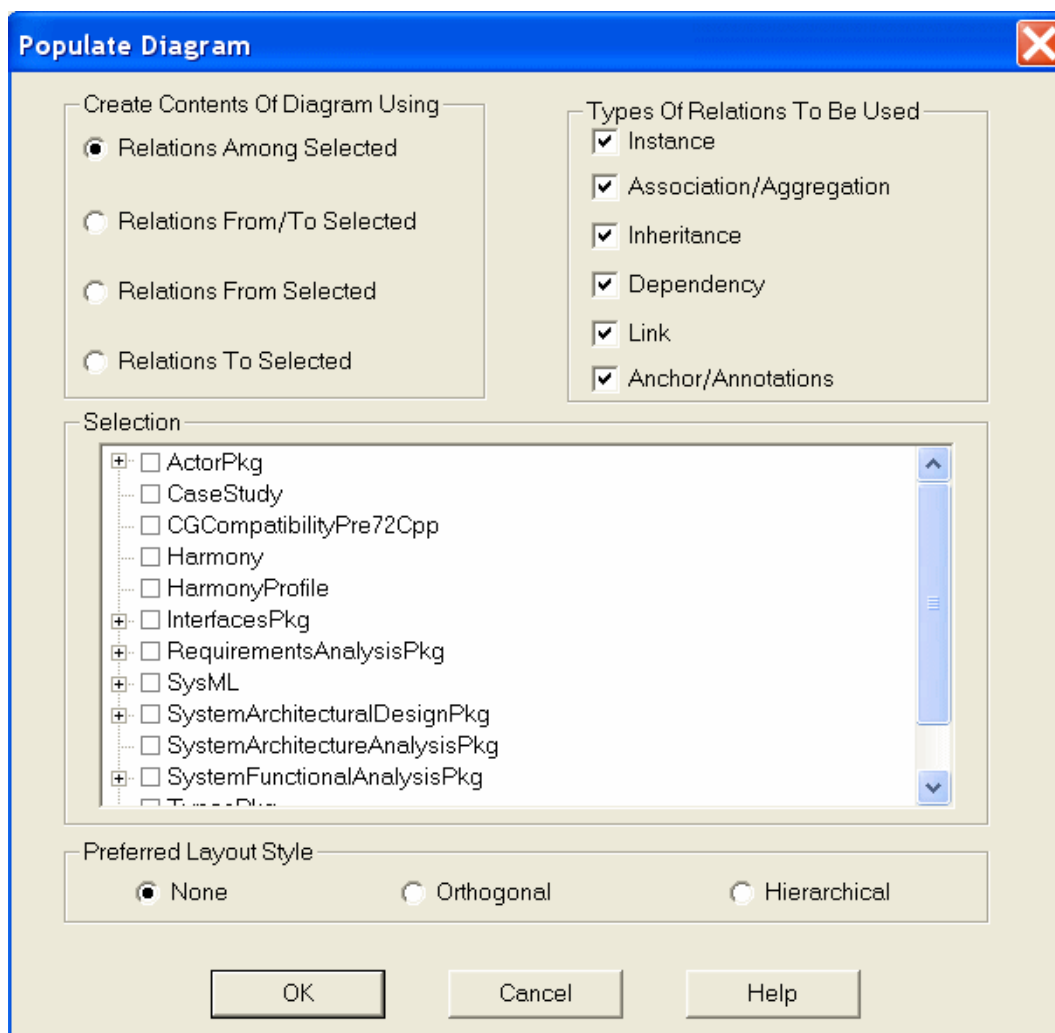
Creating Rational Rhapsody requirements diagrams

In projects created with the SysML or Harmony profiles, you can use requirements diagrams, with requirements imported from other software products or created in Rational Rhapsody, to communicate complex details to team members.

To create a requirements diagram:

1. Highlight the **Requirements** package in the browser.
2. Right-click and select **Add New > Diagrams > Requirements Diagram**.
3. Enter the **Name** of the new diagram. Click **OK** if you want to add items individually to the requirements diagram. However, if you want the system to put existing model items into the new diagram, check the **Populate Diagram** box and select those items.

4. Select the characteristics and content for the new requirements diagram from the Populate Diagram window. Click **OK**.



Requirements diagram drawing tools

Use the drawing tools to show the relationships and functions to create a detailed requirements diagram, as shown in the [Requirements Diagram with Requirements Imported from Word](#).

Requirements Diagram Drawing Tools

Drawing Tool	Name	Purpose
	Requirement	Definition of a system requirement
	Problem	Unfavorable environment situation that the requirement is meant to address or might encounter as a blocking condition
	Rationale	Statement of the reason for a specific requirement
	Package	Used to show the relationship between the package artifact and requirements
	Derivation	Indicates that a requirement was derived from another
	Satisfaction	Indicates an artifact or condition that is necessary to fulfill a specific requirement
	Verification	Indicates how a specific requirement is verified as supplied in the design
	Dependency	Indicates a dependent relationship between two items in the diagram
	Use Case	Used to show the relationship between a use case and a requirement
	Allocation	Indicates that an artifact or requirement is exclusively reserved for use of another

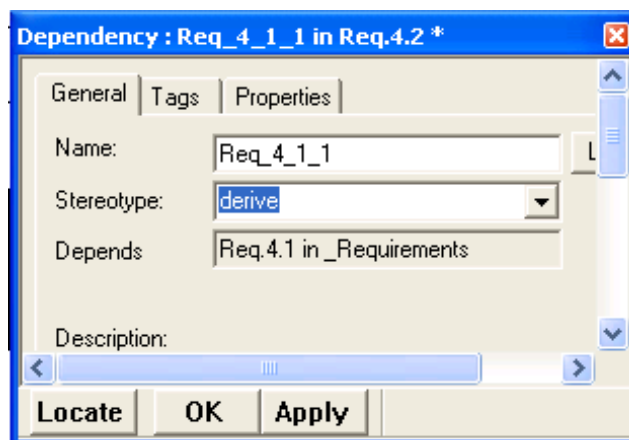
Drawing and defining the dependencies

With elements drawn in the requirements diagram, use a *dependency* to show a direct relationship in which the function of an element requires the presence of and might change another element. You can show the relationship between requirements, and between requirements and model elements using dependencies. You can set the following types of dependency stereotypes:

- ♦ **Derive** indicates a requirement that is a consequence of another requirement.
- ♦ **Trace** shows the dependency from the model element to its requirement with the dependency arrow head on the requirement.

To define the relationships between requirements with dependencies:

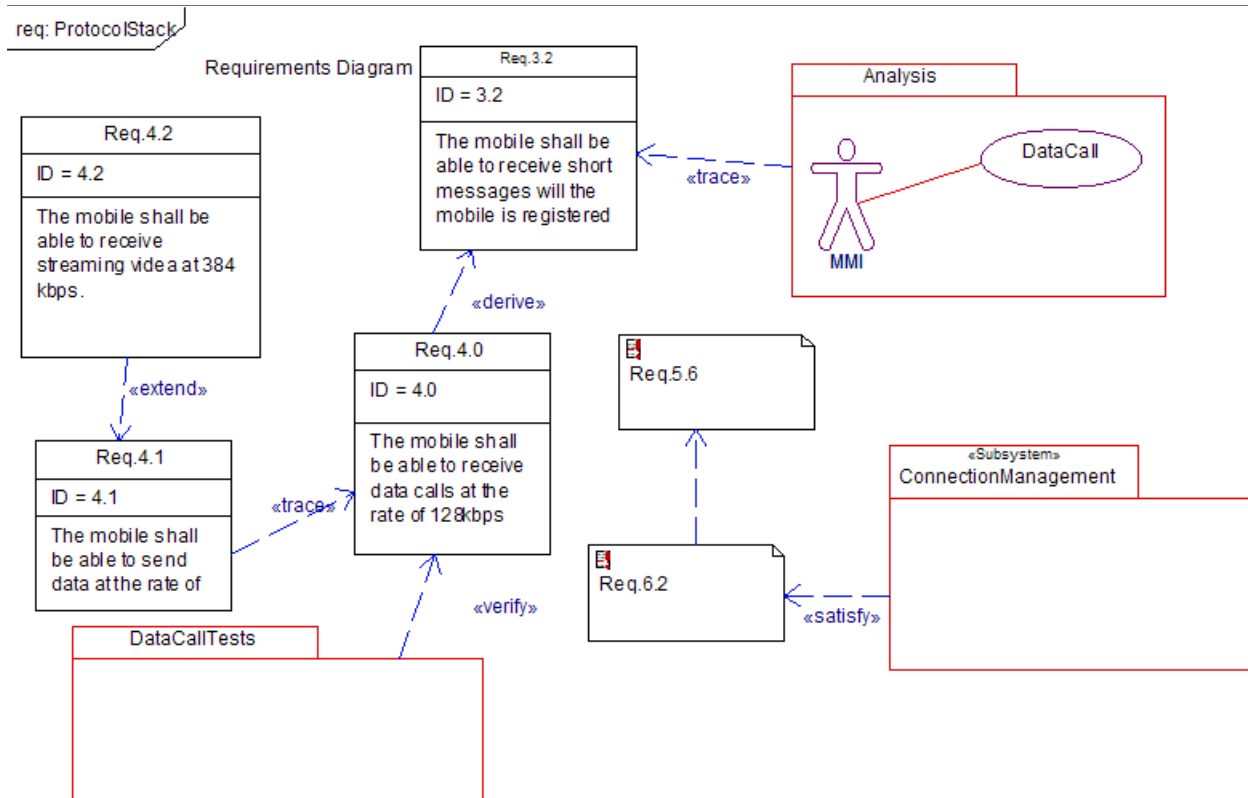
1. Click the **Dependency** button  on the **Diagram Tools**.
2. Draw a dependency line from one requirement to another. Right-click on the dependency line and select **Features**. At this point you might select **derive** as the **Stereotype**, as shown in this example, or another possible stereotype including trace, extend, refine, allocate, conform, decompose, satisfy, verify, valueBinding, Send, Usage, Friend, or <<New>>.



3. Click **OK**.
Rational Rhapsody automatically adds the dependency relationships to the browser.

Creating specialized requirement types

To specify specialized requirements types, use the sub-typing features of stereotypes. The following example of these stereotypes can be examined in the Rational Rhapsody System Samples directory in the SysMLHandset project.



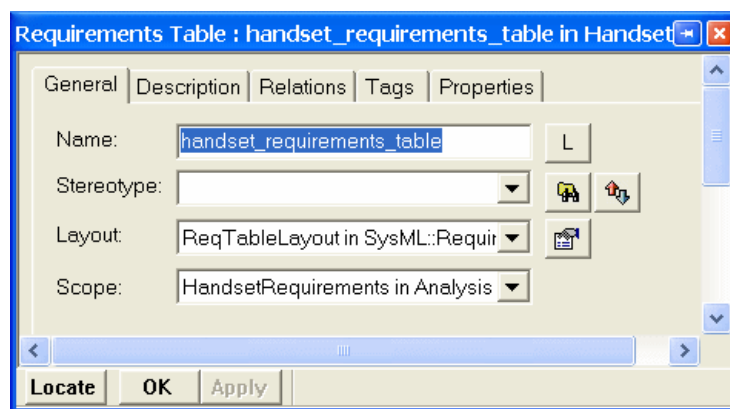
- ◆ <<**extend**>> shows that a requirement expands or provides more detailed view of another requirement. (See Req 4.2 and 4.1 in the example above.)
- ◆ <<**derive**>> shows a relationship between two requirements and supplies additional details. A derive requirement often reflects assumptions about the implementation of the system. (In the diagram, the arrow direction is from the derived to the original requirement.)
- ◆ <<**composite**>> requirements are the sub-requirements within the overall requirements hierarchy. This structure allows a complex requirement to be decomposed into its containing child requirements.
- ◆ <<**satisfy**>> relationship identifies the system or other model element intended to satisfy or fulfill the requirement. (In the diagram, the arrow direction is from the satisfying to the satisfied.)

- ◆ <<**verify**>> shows the relationship between a requirement and its test case. A test case is usually expressed as an activity or interaction diagram.
- ◆ <<**refine**>> relationship shows how a model element or set of elements further explains a requirement.
- ◆ <<**trace**>> requirement relationship provides a general purpose relationship between a requirement and any other model element. The semantics of <<trace>> do not include real constraints and, as a result, should not be used with any of the other requirements relationships listed previously.

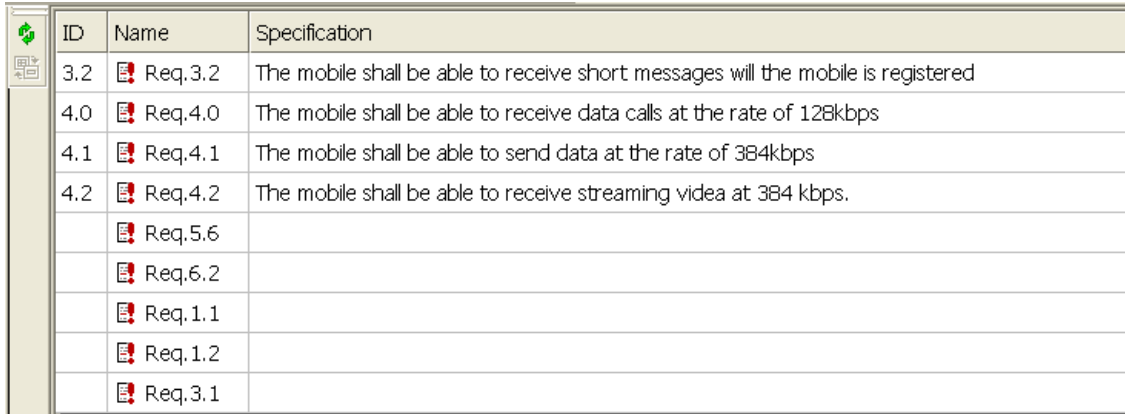
Requirements tabular view

You can use the [Table and matrix views of data](#) feature to create different layouts to view the requirements information in the project. However, you might also use the preformatted SysML requirements table:

1. Right-click the Requirements package in the browser.
2. Choose **Add New > Requirements > RequirementsTable** to access the predefined SysML requirements layout.
3. Right-click the generated name of the new table in the browser and select **Features**.
4. Enter the **Name** of the requirements table to be displayed in the browser list. The preformatted **Layout** is automatically listed, but you need to select the package to be analyzed in the **Scope**, as shown in this example from the Rational Rhapsody SysMLHandset sample project. Click **OK**.



5. In the browser double-click the requirements table name to generate the table in the drawing area, as shown in this example.



ID	Name	Specification
3.2	Req.3.2	The mobile shall be able to receive short messages will the mobile is registered
4.0	Req.4.0	The mobile shall be able to receive data calls at the rate of 128kbps
4.1	Req.4.1	The mobile shall be able to send data at the rate of 384kbps
4.2	Req.4.2	The mobile shall be able to receive streaming videa at 384 kbps.
	Req.5.6	
	Req.6.2	
	Req.1.1	
	Req.1.2	
	Req.3.1	

Creating use case diagrams

Use case diagrams show the system main functions (use cases) and the entities (actors) outside the system.

To start a use case diagram containing the actors and basic use cases:

1. In the browser, right-click the package containing your analysis, and select **Add New > Diagrams > UseCaseDiagram** from the menu. The New Diagram window opens.
2. In the **Name** box replace the generated name with the name you want.
3. Check the **Populate Diagrams** check box if you want to select items in the project to place in the new diagram automatically.
4. Click **OK**.

Rational Rhapsody adds the Use Case Diagrams category in the browser and opens the new diagram. The [Use case diagram drawing tools](#) provides the drawing tools for these diagrams (shown in the following sections).

Note

For systems engineering purposes, the **Extend** and **Generalization** features should not be used in the use cases.

Boundary box and the environment

The boundary box delineates the system under design from the external actors. It shows what is within the system environment. Use cases are inside the system (boundary box); actors are outside the system.

To draw the boundary box in a use case diagram:

1. Click the **Create Boundary box** button  in the **Diagram Tools**.
2. Click in the upper, left corner of the drawing area and drag to the lower right. Rational Rhapsody creates a boundary box.
3. Rename the boundary box to be represent your project.

Actors and systems design in use cases

The Rational Rhapsody actors represent the following in systems design models:

- ◆ Entities that are outside the system
- ◆ External interfaces
- ◆ Parts
- ◆ Flows through standard ports

To draw the actors:

1. Click the **Actor** button  in the **Diagram Tools**.
2. Click the location where you want to position the actor symbol in the use case diagram. Rational Rhapsody creates an actor with a default name of actor_n, where *n* is greater than or equal to 0.
3. Type an appropriate name for the actor to represent the function it serves. Rational Rhapsody adds the actor to the browser.

Note

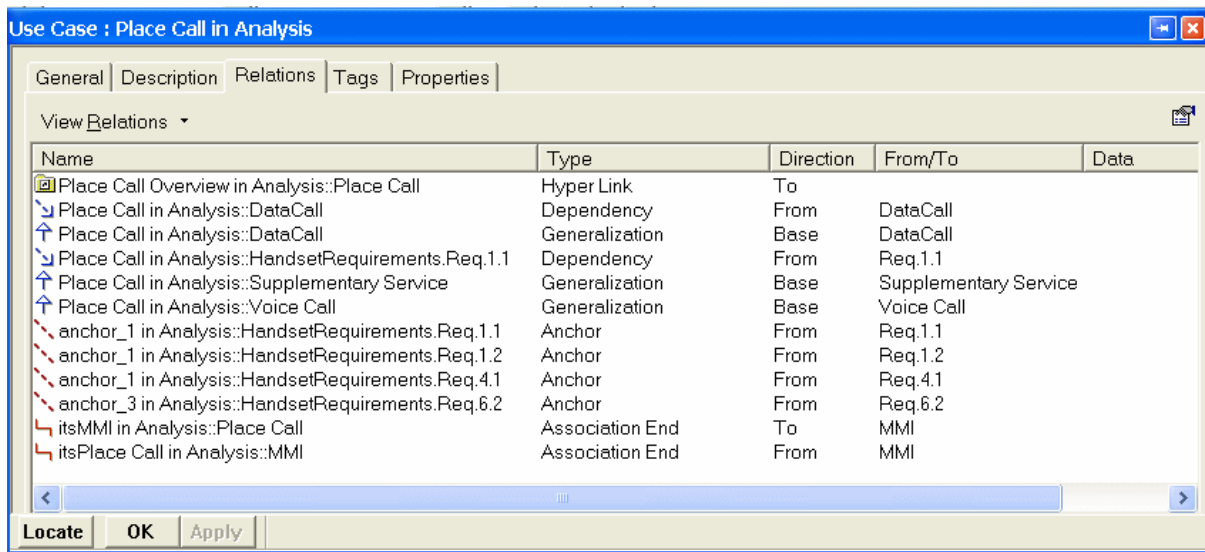
Because code can be generated using the specified names, do not include spaces in the names of actors.

Use case features for systems engineering

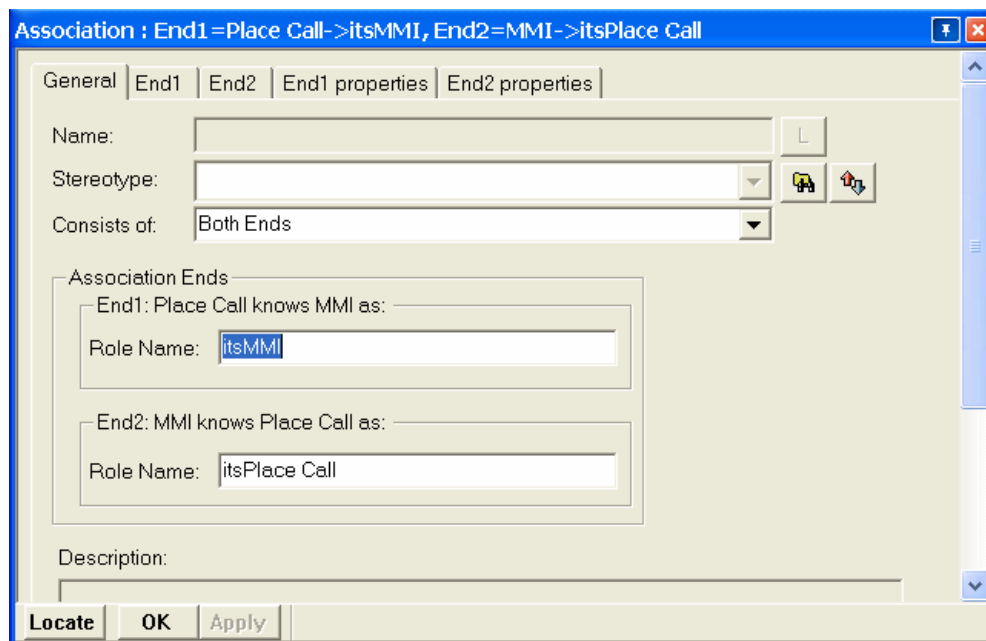
You can define the features of each use case and associate the use case with a different main diagram using the Features window.

To define use case features:

1. In the browser, expand a package and the Use Cases category. Double-click the use case, or right-click and select **Features**. The Features window opens.
2. Select the **Description** tab, and type the text to describe the purpose of the use case using the internal text editor.
3. Select the **Relations** tab to examine the dependencies, flows, generalizations, and associations.



4. Double-click any item in the View Relations list to examine the details and make any required changes.



5. Click **OK**.

Associating actors with use cases

Actors initiate actions or receive information from the system.

To create these necessary connections for interaction, draw association lines:

1. Click the **Create Association** button  in the **Diagram Tools**.
2. Click the edge of the actor, then click the edge of a use case. Rational Rhapsody creates an association line with the name label highlighted. You do not need to name this association, so you can press **Enter**, or you can type label text in the highlighted area.
3. In the browser, expand the **Actors** category to view any relationships you have created between actors and use cases.

Defining requirements in use case diagrams

Systems engineers often employ use case diagrams to define requirements. This technique provides the following advantages:

- ◆ Naming system capabilities to add specificity to design work
- ◆ Showing important user interactions with the system to consider in the design
- ◆ Returning a result visible to one or more actors
- ◆ Organizing requirements by the use cases to recognize possible design flaws early in the design process
- ◆ Assisting project planning by revealing important relationships in the use cases

Tracing requirements in use case diagrams

You can add requirement elements to use case diagrams to show how the requirements trace to the use cases.

To add the requirements to the use case diagram:

1. Select a requirement from the browser and drag it into or beside a use case.
2. To be certain that the requirement is visible in the diagram, right-click the requirement you placed in the diagram and select **Display Options**. The Requirement Display Options window opens.
3. The **Show** group box specifies the information to display for the requirement. Select the **Name** radio button to display the name of the requirement.
4. Click **OK**.

Dependencies between requirements and use cases

You can also use dependencies to link the requirements with the use cases as follows:

1. Click the **Dependency** button  in the **Diagram Tools**.
2. Click on the element in the use case diagram and draw the dependency line to the associated requirement. (The dependency arrow head rests on the requirement.)
3. In the browser, expand the Requirements category to check that the dependency relationship is listed there.

Defining flow in a use case diagram

To specify the exchange of information between system elements, use the **Flow** button  to indicate the flow of data and commands within a system without specifying the details of this communication. As the engineer works on the system specification, these abstractions can be tied to the concrete implementations.

Defining the stereotype of a dependency

You can specify the ways in which requirements relate to other requirements and model elements using stereotypes. A *stereotype* is a modeling element that extends the semantics of the UML metamodel by typing UML entities.

Rational Rhapsody includes predefined stereotypes, and you can also define your own stereotypes. Stereotypes are enclosed in guillemets on diagrams, for example, «*derive*».

To define the stereotype of a dependency:

1. Double-click the dependency between a requirement and a use case, or right-click and select **Features**.
2. Select **trace** from the **Stereotype** pull-down list.
3. Click **OK**.

Activity modeling in SysML

Activity modeling in SysML emphasizes the inputs and outputs, sequence, and conditions for coordinating other behaviors, instead of who owns those behaviors. Therefore, the SysML activities specify the following information:

- ♦ Coordination of executions of lower level behaviors
- ♦ Flow of control
- ♦ Flow of data

Activities are modelled as “Classifiers” or “Types” with the keyword <<activity>>.

Action types in SysML

The following are the five basic action types in SysML:

- ♦ **Atomic action** controls flows into the action, the action is performed, and then control flows out of the action.
- ♦ **Call behavior action** launches other activities.
- ♦ **Call operation** launches an operation call on a target block or part.
- ♦ **Accept event action** handles processing of events during the execution of a behavior.
- ♦ **Send signal action** graphically shows an event sent to interact with another block.

SysML activity diagrams

Activity diagrams show the essential interactions between the system and the environment and the *interconnections of behaviors* for which the subsystems or components are responsible. These diagrams also illustrate the *flow of control* from activity to activity with sequences and conditions. Activity diagrams can model an operation or the details of a computation and be animated to verify the functional flow.

Creating an activity diagram

To create an activity diagram:

1. Start Rational Rhapsody if it is not already running and open the model if it is not already open.
2. In the browser, expand the package containing your subsystems, locate the wanted block, and the **Parts** category.
3. Right-click on the item for which you want to describe its activities and select **Add New > Diagrams > Activity Diagram** or click the **Activity Diagram** icon  at the top of the window.

The blank diagram opens in the drawing area. You might want to add a title to the diagram to help quickly identify the diagram.

Setting activity diagram properties

Action states represent function invocations with a single exit transition when the function completes. In this example, you will draw the action states that represent the functional processes, and then add names to the action states.

The default settings are used when you add an Action and type a name in the action state on the diagram. That name becomes the action text, not the name of the action. Before adding actions, set the *properties* for the diagram:

1. Right-click outside the Swimlanes frame and select **Diagram Properties**.
2. Select the **Properties** tab and click the **All** radio button for the Filter.
3. Open the **Action** category and change the **showName** and **ShowAction** properties to use these values:

```
Activity_diagram::Action::showName = Name
```

```
Activity_diagram :: Action :: ShowAction = Description
```

This second property allows informal text to be displayed on the diagram, while the actual action is described formally using an *executable language*.

4. Click **OK**.

Activity diagram drawing tools for systems engineering

A systems engineering activity diagram has the following drawing tools:

Drawing Tool	Name	Definition
	Accept Event Action	Lets you add this element to a systems engineering activity diagram so that you can connect it to an action to show the resulting action for an event. This element can specify the following actions: • Event to send • Event target • Values for event arguments This button is displayed by default in a new SysML profile project.
	Accept Time Event	Adds an element that denotes the time elapsed since the current state was entered.
	Action	Creates a single, top-level action state. For more information, see Drawing action states .
	Action Block	Draws compound actions that can be decomposed into actions. Action blocks can show more detail than might be possible in a single, top-level action.
	Action Pin	Adds an element to represent the inputs and outputs for the relevant action or action block. An action pin can be used on a Call Operation (derived from the arguments). This button displays in on Diagram Tools when you select the Analysis Only check box when defining the general features of the activity diagram.
	Activity Final	Signifies either local or global termination, depending on where they are placed in the diagram.
	Activity Flow	Creates a transition between action states.
	Activity Parameter	Defines a characteristic of an action block. This button is displayed by default in a SysML profile project.
	Call Behavior	Creates a call to a behavior in another activity diagram or to the entire activity diagram. You can add calls to both activity diagrams and subactivity diagrams.
	Call Operation	Represents a call to an operation of a classifier.
	Decision Node	Combines different flows into a common target.
	Dependency	Indicates a dependent relationship between two items in the diagram.
	Flow Final	Marks the last transition between action states
	Fork Node	Allows the splitting of one in-going flow into two or more outgoing concurrent flows.

Drawing Tool	Name	Definition
	Initial flow	Identifies the starting point for the actions in the activity diagram. For more information, see Drawing a initial flow .
	Join Node	Allows the merging of two or more concurrent flows into a single outgoing flow.
	Merge Node	Combines different flows to a common target and is often a decision point.
	Object node	Identifies where an object is passed from the output of one state's actions to the input of another state's actions.
	ObjectFlow	Identifies the transition of an object from one action state to another.
	Send Action	Represents sending actions to external entities. The Send Action is a language-independent element, which is translated into the relevant implementation language during code generation.
	Subactivity	Adds a new subchart to an existing action. This subchart defines a secondary action to the main action. This helps to simplify the diagram. For more information, see Drawing a subactivity .
	Swimlanes Divider	Divides the swimlane frame using vertical, solid lines to separate each swimlane (actions and subactions) from adjacent swimlanes.
	Swimlanes frame	Organizes activity diagrams into sections of responsibility for actions and subactions.

Drawing action states

To draw action states in the diagram:

1. Click the **Action** button  in the **Diagram Tools** and create action states in the diagram.
2. Name each new action carefully to describe its function.
3. Click the action state, or right-click and select **Features**.
4. In the **Description** box, type the wanted information.
5. Click **OK**.

Drawing a initial flow

One of the Action States must be the *initial flow*. This is the initial state of the Activity. To identify the default flow state:

1. Click the **Initial Flow** button  in the **Diagram Tools**.
2. Click near the default action state and then click its edge. Press **Ctrl+Enter** to stop drawing the connector and not label it.

Drawing a subactivity

A *subactivity* represents the execution of a non-atomic sequence of steps nested within another activity.

1. Click the **Subactivity** button  in the **Diagram Tools**.
2. In the swimlane, click or click-and-drag to draw the subactivity state.
3. Name the subactivity state.
4. To display the subactivity icon in the lower right corner of the state drawing, right-click the subactivity box and select **Display Options** from the menu.
5. Click the **Icon** radio button for the **Show Stereotype** selections and click **OK**.

Note

Limitation: Subactivities do not support swimlanes.

Drawing activity flows

Activity flows represent the response to a message in a given state. They show what the next state will be. In this example, you will draw the following transitions:

- ♦ Transitions between states
- ♦ Fork and join transitions
- ♦ Timeout transition

Note

To change the line shape of an activity flow, right-click the line, select **Line Shape**, and then **Straight**, **Spline**, **Rectilinear**, or **Reroute**.

Drawing activity flows between states

To draw activity flows between states:

1. Click the **Activity Flow** button  in the **Diagram Tools**.
2. Click the subactivity state, and then click the state.
3. Name the activity flow and then press **Ctrl+Enter**.

Rational Rhapsody allows you to assign a descriptive label to an element. A labeled element does not have any meaning in terms of an executable action, but the label helps you to reference and locate elements in diagrams and windows. A label can have any value and does not need to be unique.

Note

When drawing activity flows, it is a good practice to not cross the flow lines. This makes the diagram easier to read.

Drawing swimlanes

Swimlanes organize activity diagrams into sections of responsibility for actions and subactions. Vertical, solid lines separate each swimlane from adjacent swimlanes. To draw swimlanes, create a swimlane frame and then a swimlane divide:

1. Click the **Swimlanes Frame** button  in the **Diagram Tools**.
2. Click to place one corner, then drag diagonally to draw the swimlane frame.
3. Click the **Swimlanes Divider** button  in the **Diagram Tools**.
4. Click the middle of the swimlane frame. Rational Rhapsody creates two swimlanes, named `swimlane_n` and `swimlane_n+1`, where n is an incremental integer starting at 0. Rename the swimlanes as wanted.

If you drag the swimlane left or right, it also resizes the swimlane frame.

Drawing a fork node

A *fork node* represents the splitting of a single flow into two or more outgoing flows. It is shown as a bar with one incoming activity flow and two or more outgoing activity flows.

To draw a fork node bar:

1. Click the **Fork Node** button  in the **Diagram Tools**.
2. Click or click-and-drag between two action states. Rational Rhapsody adds the fork node bar.
3. Click the **Activity Flow** button, and draw a single incoming activity flow from one state to the fork node bar. Type the name and then press **Ctrl+Enter**. This activity flow indicates that a call request has been initiated.

Drawing a join node

A *join node* represents the merging of two or more concurrent flows into a single outgoing flow. It is shown as a bar with two or more incoming activity flows and one outgoing transition.

To draw a join node bar:

1. Click the **Join Node** button  in the **Diagram Tools**.
2. Click or click-and-drag between an action state and a subactivity. Rational Rhapsody adds the join node bar.
3. Click the **Activity Flow** button and draw the incoming activity flows to the join node bar.
4. Draw one outgoing activity flow from the bar to subactivity. Type name, and then press **Ctrl+Enter**.

Creating a sequence diagram from an activity diagram

To generate a new sequence diagram from an existing activity diagram:

1. Select the activity diagram in the browser.
2. Right-click and select **Create New Scenario from Activity Diagram** from the menu.

For more information about using this feature, see [Harmony process and toolkit](#).

Creating a design structure

Internal Block diagrams and *Block Definition diagrams* define the system structure and identify the large-scale organizational pieces of the system. They can show the flow of information between system components, and the interface definition through ports. In large systems, the components are often decomposed into functions or subsystems.

Block diagrams define the components of a system and the flow of information between components. *Structure diagrams* can have the following parts:

- ♦ **Block** contains parts and might also include links inside a block.
- ♦ **Actors** are the external interfaces to the system.
- ♦ **Standard Port** is a distinct interaction point between a class, part, or block and its environment.
- ♦ **Dependency** shows dependency relationships, such as when changes to the definition of one element affect another element
- ♦ **Flow** specifies the exchange of information between system elements at a high level of abstraction.

Block properties

Blocks have three different types of properties:

- ♦ **Structural** properties are parts that refer to other system elements that are required for the system to exist. Parts have a context and, therefore, show the usage of the system elements or blocks.
- ♦ **Reference** properties point to other model elements that are not parts.
- ♦ **Value** properties provide system information such as mass, length, or status, but not the target of any reference. Values can be UML data types (integers) or SysML value types in engineering units with additional characteristics (unit of measure and/or dimension).

Blocks and behaviors

Blocks execute actions that are primitive behaviors as the following examples show:

- ◆ `x=x+1`
- ◆ `y = sin(x)^2 + cos(x)^2`
- ◆ `addTogether(int x, int y)`

Actions might be grouped together in different ways:

- ◆ As a method to start a behavior consisting of a set of actions
- ◆ As a state machine specifying sequences of actions to be executed when the block receives events
- ◆ As an activity diagram specifying sequences of actions from start to completion

Creating a block definition diagram

A *Block Definition diagram* (External block diagram) shows the system structure and identifies the system components (blocks) and describes the flow of data between the components from a black-box perspective. To create a [Block Definition Diagram](#):

1. In the browser, right-click the Architecture package, then select **Add New > Diagrams > Block Definition Diagram**. The New Diagram window opens.
2. Type a name for your architecture diagram.
3. Click **OK**.

Rational Rhapsody automatically creates the Block Definition Diagrams category in the browser and adds the name of the new block definition diagram. In addition, Rational Rhapsody opens the new diagram in the drawing area allowing you to construct the diagram using the [Adding graphics to block definition diagrams](#). As you add blocks and link them to show relationships, you might consider the many uses of blocks in [Block properties](#) and [Blocks and behaviors](#). You can also add [Adding graphics to block definition diagrams](#).

Block definition diagram drawing tools

The **Diagram Tools** for a block definition diagram includes the following tools:

Drawing Tool	Name	Definitions
	Block	Draws an instance of a class that can belong to packages and parts.
	Part	Draws a major component of the model.
	FlowSpecification	Creates a non-atomic flow port typed by an interface.
	Package	Draws a group of parts or blocks to form a single element of the model.
	FlowPort	Shows how the data flows between blocks.
	StandardPort	Draws the connection points among blocks or parts and their environments.
	Association	Creates connections that are necessary for interaction.
	Directed association	Indicates the only object that can send messages to another object.
	Aggregation	Shows an association specifying a whole-part relationship between the aggregate (whole) and a component part.
	Directed Composition	Shows the instance of a class that cannot be contained by other instances.
	Connector	Shows the relationship among blocks or parts.
	Dependency	Indicates a dependent relationship between two items in the diagram.
	Inheritance	Indicates the relationship between the derived class and its parent. The derived class has the same data members and behaviors as the parent class.
	Flow	Indicates the flow of data and commands within a system.
	ConstraintBlock	Defines restrictive properties controlling the relationships of blocks.
	Constraint	Defines a semantic condition or restriction.
	BindingConnector	Specifies the properties at the ends of the connector (link).

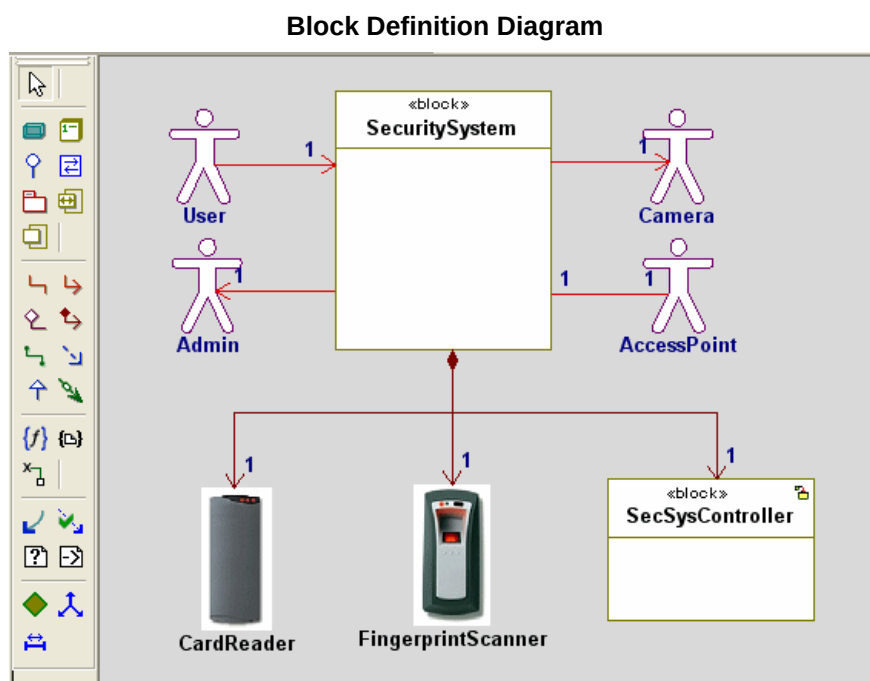
Drawing Tool	Name	Definitions
	Satisfaction	Explains how problem will be overcome. It usually includes the key element or design feature that solves the problem.
	Allocation	Allows a systems engineer to denote the mapping of elements within the structure of a model.
	Problem	Records an unresolved issue, limitation, or failure related to one of the model elements.
	Rationale	Permits you to record the reason for decisions, requirements, and other design issues. A rationale might reference a more detailed document or report.
	Value Type	Creates an extension of the UML dataType. It is used where one would use a dataType (as a type for a value property for a block, for example). A UML dataType usually expresses a quantity in a software implementation type, such as a float or double. However, in SysML, valueType expresses a quantity in a standard unit, such as milliAmpere. The valueType also includes a placeholder for the quantity plus the unit. It should be noted that SysML allows a valueType to be defined without a unit. For example, when the valueType expresses a ratio, the valueType expresses a quantity only.
	Dimension	Used in SysML to separate the concept of "Unit" from "Dimension." A SysML Dimension represents a standard physical concept for a set of Units. For example "Length" is a Dimension in SysML. The Units meter, inch, kilometer, mile, light_year refer to the concept of Length; therefore, they all have Length as the value of their Dimension tag. Dimension helps to organize Units into comprehensive sets based on the physical domain. Like Units, a Dimension should be used only to set the Dimension tag for a Unit or for a valueType and is never used as a type. Dimensions are usually selected from a standard library model (part of the Rational Rhapsody SysML profile).
	Unit	Used in conjunction with a SysML valueType to express a given quantity in a standard way so that other quantities having the same Unit can be compared. The Unit is usually taken from a standard library of Units, such as SI or NIST (part of the Rational Rhapsody SysML profile). Units might be expressed in terms of other units or "derived units." Units are only used to set the Unit tag of a valueType. They should not be used as types for value properties since the Unit concept in SysML does not include the concept of quantity. Instead, a valueType should be used.

Adding graphics to block definition diagrams

To add the graphics to the block definition diagram:

1. Highlight the item in the diagram for which you want to add a image, such as a photograph or drawing.
2. Right-click and select **Display Options** from the menu.
3. In the **General** tab of the window, select the **Enable Image View** check box. This automatically enables the Select An Image option.
4. Browse to find the **Image File Path** and click **OK** to insert the selected image.

The following example shows a completed block definition diagram with two inserted images, the CardReader and FingerprintScanner. Using images to represent system components helps the project team members to identify individual parts quickly.



Creating an internal block diagram

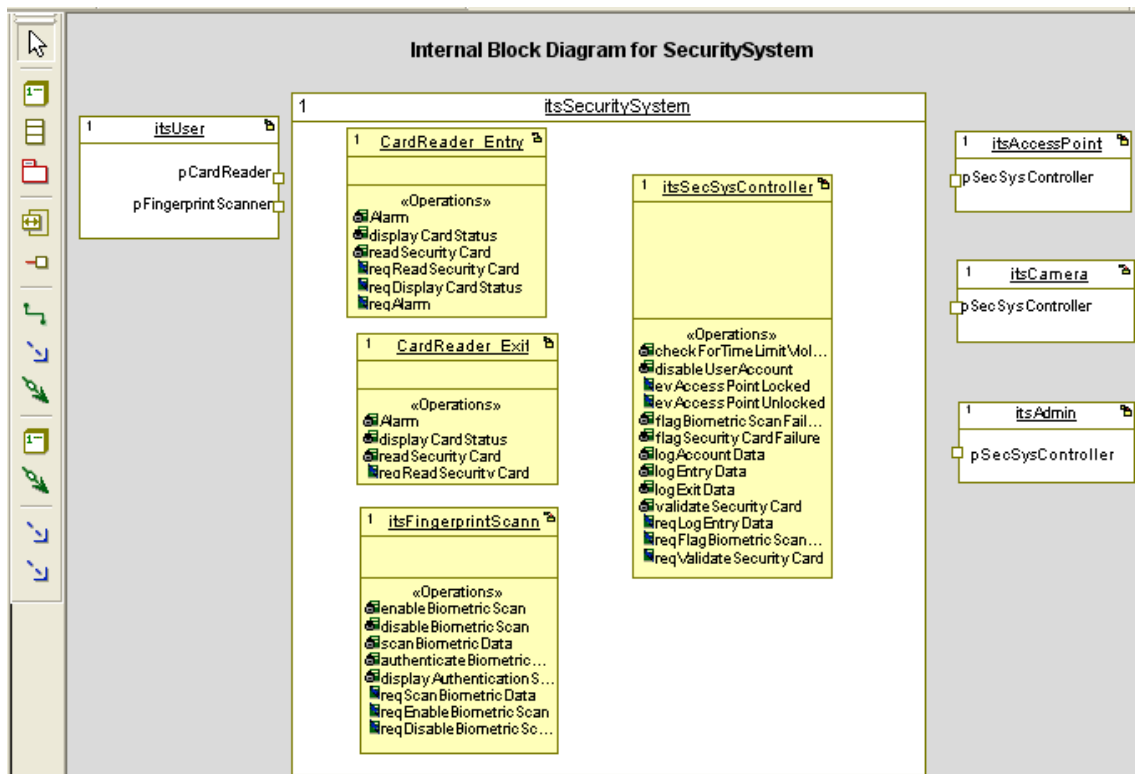
An *internal block diagram* shows the internal structure or decomposition of a block into its parts or subsystems.

To create an internal block diagram:

1. In the browser, expand a package.
2. Right-click block in the package and select **Add New > Diagrams > Internal Block Diagram**.
3. Type the name diagram and click **OK**. Rational Rhapsody creates the diagram in drawing area.

The following example shows a completed internal block diagram for a security system.

Internal Block Diagram Example



If the ports are not visible, right-click the block and then select **Ports > Show All Ports**.

Internal block diagram drawing tools

An internal block diagram has the following drawing tools:

Drawing Tool	Name	Definition
	Part	Draws a major component of the model.
	Block	Draws an instance of a class that can belong to packages and parts.
	Package	Draws a group of parts or blocks to form a single element of the model.
	FlowPort	Shows how the data is bound to each constraint.
	StandardPort	Draws the connection points among blocks or parts and their environments.
	Connector	Shows the relationship among blocks or parts.
	Dependency	Shows the relationships among the packages in the diagram.
	Flow	Shows how the data is bound to each constraint.
	ConstraintProperty	Defines a characteristic of a semantic condition or restriction.
	BindingConnector	Specifies the properties at the ends of the connector.
	Satisfaction	Explains how problem will be overcome. It usually includes the key element or design feature that solves the problem.
	Allocation	Allows a systems engineer to denote the mapping of elements within the structure of a model.

Drawing the parts

The internal block diagram uses parts to represent the activities. To draw the parts:

1. Click the **Part** button  in the **Diagram Tools**.
2. In the upper, left corner of the block click or click-and-drag.
3. Type the name you want, and then press **Enter**.

Drawing standard ports and links

You need to link the parts to show the interactions of parts. To accomplish this, you need to draw standard ports as follows:

1. Click the **Standard Port** button  in the **Diagram Tools**.
2. Click on the edge of the block, name the standard port.
3. Click on the edge of another block, name the standard port.

To draw link the two parts through their standard ports:

1. Click the **Connector** button  in the **Diagram Tools**.
2. Click the standard port of the sending part, and then click the standard port of the receiving part.

Specifying the port contract and attributes

Now you can specify the port contract and attributes for a port *interface* as follows:

1. Double-click the port to display the Features window.
2. In the **General** tab, click the **Behavior** and/or **Reversed** radio buttons to set the wanted **Attributes**. Click **Apply** to save the changes and keep the window open.
3. Select the **Contract** tab.
4. Select the **Provided** folder icon and click the **Add** button. The Add new interface window opens.
5. Select **In** or another option from the pull-down list, then click **Apply** to save the changes and leave the window open.
6. Select the **Required** folder icon and click the **Add** button. Select **Out** or another option from the drop-down list.
7. Click **OK**.
Rational Rhapsody automatically adds the provided and required interfaces.
8. Click **OK**.

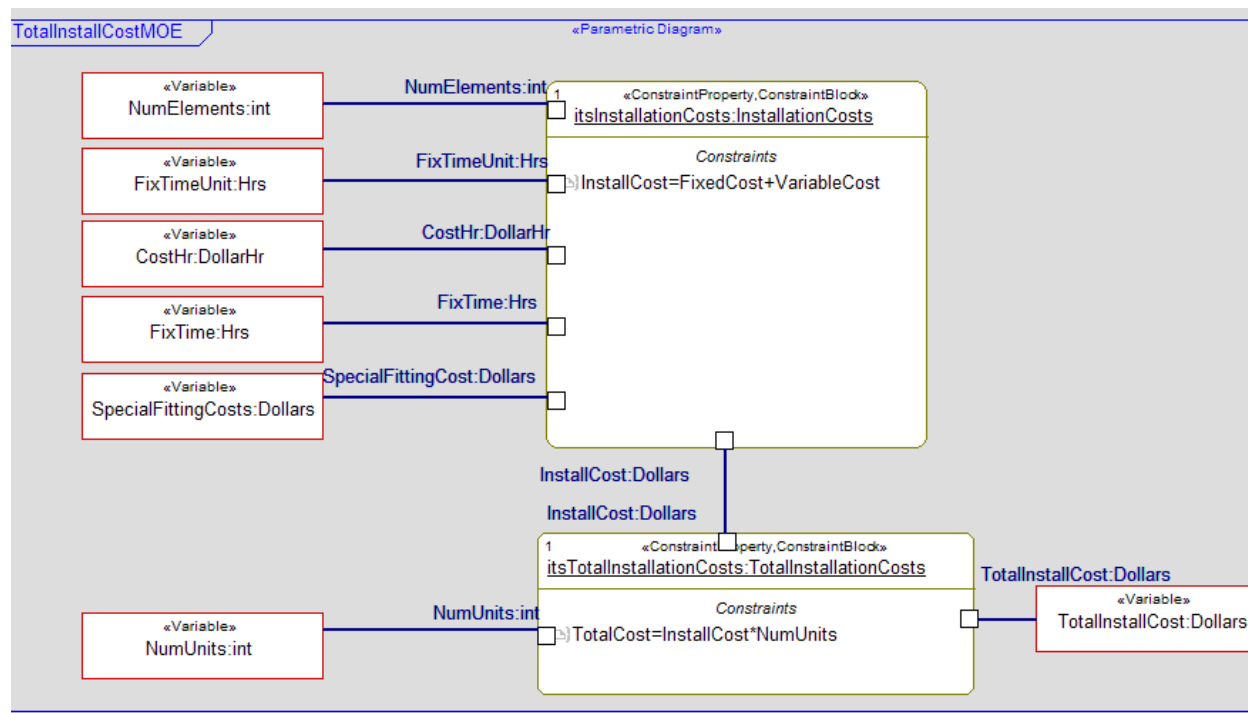
Parametric diagrams

Parametric diagrams show mathematical relationships (such as performance constraints) among the pieces of the system being designed. These diagrams are only available if you are using the SysML profile for your project. Parametric diagrams have the following general uses:

- ◆ Indicate the relationships for the objective analysis of functions
- ◆ Measure effectiveness
- ◆ Clarify the relationship between one variable and another

Parametric Diagrams cannot exist in isolation. They are created using model elements called *constraint blocks* that define generic or basic mathematical formulas. See the example of a completed parametric diagram in [Security System Total Installation Costs for X number of units](#).

Security System Total Installation Costs for X number of units



To illustrate another possible use for a parametric diagram, a set of constraint blocks could define the volume of a tube, a disc, and the formula for an objects mass ($\text{Volume} * \text{Density}$). The parametric diagram would show how these constraint blocks combine, in a particular usage as a set of constraint properties, to give the mass of, perhaps, a tin can or a hollow cylindrical container based upon a set of input parameters. Constraint blocks are created within a block definition diagram, as described in [Creating a block definition diagram](#).

Parametric diagram drawing tools

A parametric diagram has the following drawing tools:

Drawing Tool	Name	Definition
	ConstraintBlock	Defines restrictive properties controlling the relationships of blocks.
	ConstraintProperty	Defines the usages of a Parametric Constraint Block so that the blocks can be reused with changes to the usage of the property, but without any changes to the underlying equations.
	Package	Draws a group of parts or blocks to form a single element of the model.
	Block	Draws an instance of a class that can belong to packages and parts.
	Part	Draws a major component of the model.
	ConstraintParameter	Defines a characteristic of a semantic condition or restriction.
	Binding Connector	Binds the data to the constraint.
	Satisfaction	Explains how problem will be overcome. It usually includes the key element or design feature that solves the problem.
	Allocation	Allows a systems engineer to denote the mapping of elements within the structure of a model.
	Dependency	Indicates a dependent relationship between two items in the diagram.

Creating the constraint block

Parametric diagrams are based upon *constraint properties*. Constraint properties can only be created from *constraint blocks*. To create a constraint block in a block definition diagram:

1. Right-click the package in the browser where you want the diagram to be created and then select **Add New > Diagrams > Block Definition Diagram**.
2. Click the **Constraint Block** button  above the window and place the Constraint Block on the block definition diagram.
3. Rename the new block using the Features window.
4. Since constraints can only be added to an element in the browser, right-click the constraint block and select **Locate**. This navigates to that block in the browser.
5. Right-click the block and select **Add New > General Elements > Constraint**. This specifies the relationship between the constraint parameter and the block.
6. Open the **Constraint Features window** and rename the constraint. Click **Apply**.
7. In the **Specification** of the constraint, add the appropriate mathematical relationship, that is, $\text{Volume} = B * D * H$. Click **Apply** and the constraint features appear in the constraint block.
8. Add attributes to the constraint block if there are any constants that the constraint formula might use, for example g which is 9.81 M/s^2 . Click **OK** to close the window and save the Features you entered.
9. Add constraint parameters for the variables in the constraint formula. This is also accomplished from the browser. Right-click the constraint block and select **Add New > Constraint Blocks > ConstraintParameter**. Rename the parameter in the Features window.
10. The constraint parameter might be typed with an SI unit by opening its Features window and then selecting **Type**. From the pull-down menu, scroll to the top and select **<<Select>>**, navigate through the package tree to the SysML profile, and locate the ModelLibrary unit definitions. Select the correct unit definition.
11. The constraint parameter with its type then displays on the constraint block. New constraint parameters can be added to the constraint block directly from the constraint parameters section of the browser hierarchy. Repeat the constraint parameter definition steps (9–11) for each variable element in the constraint.

Note

Typically, constraint parameters are not shown on constraint blocks. To hide the parameters, right-click the constraint block and select **Ports > Hide All Ports**.

Creating the parametric diagram

Constraint properties show how a constraint block is used, and the parametric diagram illustrates this usage. To define a parametric diagram, first create a constraint property and then “type” it with the constraint block.

To create a parametric diagram for a constraint block:

1. From the appropriate package in the browser, right-click the package and select **Add New > Diagrams > Parametric**.
2. From the **Diagram Tools**, select **ConstraintProperty** .
3. Drag and drop the Constraint Property onto the diagram.
4. Open the Features window and set its Type to the correct Constraint Property.
5. Rename the Property to its usage.
6. Repeat for other relevant blocks for the calculation. Each block has a set of Constraint Parameters which show relationships between the blocks when they are joined together with Binding Connectors.
7. Next you need to add new pieces of data needed in the parametric diagram to bind the parametrics.
8. Double-click a part and select **Features** and then the **Attributes** tab.
9. Click the <<**New**>> item and add mathematical attributes with the correct name and appropriate type.
10. Click **OK**.
11. Drag the data attributes from the browser onto the parametric diagram and connect it to the appropriate constraint parameter with a value binding.

Binding constraint properties together

To show how the data is bound to each constraint, you need to add constraint parameters and binding connections to the parametric diagram:

1. Click the **ConstraintParameter** button  on the **Diagram Tools** and draw this connection point on a constraint.
2. Click the **BindingConnector** button  and draw the connection between the data source and the constraint to bind a value to its constraint.

Adding equations

To add the required equations to the constraints:

1. Right-click on a constraint to display the Features window.
2. Type the equation in the **Description** area and click **OK** to save the equation and close the window.
3. To display the equation in the constraint, right-click it and select **Display Options** from the menu. Click the **Compartment** button.
4. Select **Description** from the Available list and click the **Display** button to move it to the Displayed column. Click **OK** to save this change. You might want to perform this action on each of the boxes containing an equation.

Implementation using the action language

In order to show actions in a model, the designer needs an implementation language. Rational Rhapsody includes an Action Language, a subset of C++ that uses a C++ compiler to allow you to simulate the model. This language provides the following actions:

- ♦ Message passing
- ♦ Data checking
- ♦ Actions on activity flows
- ♦ General model execution

To learn the Action Language, examine the [Basic syntax rules](#) first. Then review the [Action language reference](#) for more details.

Basic syntax rules

This streamlined version of C++ has these basic syntax rules:

- ♦ It is case-sensitive, so “evGo” is different from “evgo.”
- ♦ Names must follow these rules:
 - No spaces (“Start motor” is not correct.)
 - No special characters other than underscore (“_”) (“StartMotor@3” is not correct.)
 - Must start with a letter, can’t start with an underscore (“2ToBegin” is not correct.)
- ♦ All statements must end in a semicolon
- ♦ Do not to use [Reserved words](#) such as *id*, *for*, *next*.

Frequently used statements

To add some simple operations to your model, you can use the following statements:

- ♦ These increment/decrement operators provide standard functions:
 - `X++`; (Increment X)
 - `X--`; (Decrement X)
 - `X=X+5`; (Add 5 to X)
- ♦ To print out on the screen, use one of these:
 - `cout << "hello" << endl;`
 - `cout << attribute_name << endl;`
 - `cout << "hello : " << attribute_name << endl;`

Reserved words

The Action Language reserved words are listed below. All reserved words for built-in functions are lower case, for example, *if*.

asm	continue	float	int	params	sizeof	typedef
auto	default	for	IS_IN	private	static	union
break	delete	friend	IS_PORT	protected	struct	unsigned
case	do	GEN	long	public	switch	virtual
catch	double	goto	new	register	template	void
char	else	id	operator	return	this	volatile
class	enum	if	OPORT	short	throw	while
const	extern	inline	OUT_PO RT	signed	try	

Assignment and arithmetic operations

The following are the the assignment and arithmetic operations available in the action language:

X=1;	Sets X equal to 1
X=Y;	Sets X equal to Y
X=X+5;	Adds 5 to X
X=X-3;	Subtracts 3 from X
X=X*4;	Multiplies X by 4
X=X/2;	Divides X by 2
X=X%5;	Set X to remainder of X divided by 5
X++;	Adds 1 to X
X--;	Subtracts 1 from X

Defining an action using the action language

To define action states, Rational Rhapsody provides an action language that is a subset of C++. To define an action:

1. Double-click an action state, or right-click and select **Features**.
2. Type action language into the **Action** box, as shown in this example of an action language instruction:

```
OUT_PORT(mm_cc)->GEN(RegistrationReq);
```

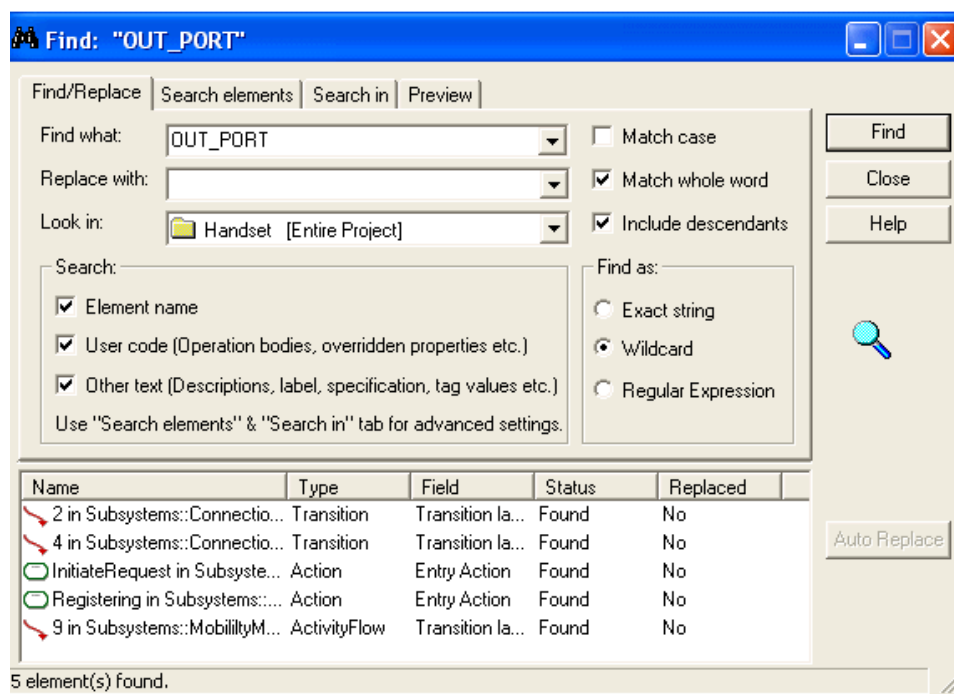
3. Click **OK**.

Checking action language entries

After entering action language into several models, it is useful to check those entries using the Rational Rhapsody search facility.

To check your action language entries:

1. Select **Edit > Search in Model**.
2. Type a portion of the instruction that you want to use for the search in the **Find What** box.
3. Click **Find**. Rational Rhapsody lists all of the locations where it found that search item.
4. Click on the entries in the list of elements found, and the system displays the diagram containing that entry and the window with the full action language content. Make any corrections that are needed.



Action language reference

This section lists each of the action language commands with its definition and syntax.

Printing

Use the following action language commands to control print operations within an application.

Using printf

```
printf(format, arg1,...,arg_n)
```

Prints arguments utilizing format specified.

Format is %type, where type is: c character s string d decimals f float

Examples:

Syntax	Output
<code>printf ("Characters: %c %c \n", 'a', 65);</code>	Characters: a A
<code>printf ("Decimals: %d %ld\n", 1977, 650000);</code>	Decimals: 1977 650000
<code>printf ("floats: %f \n", 3.1416, 4.67);</code>	Floats: 3.1416 4.67
<code>printf ("%s \n", "A string");</code>	A string

Using cout

```
cout << "Str_1" << Var_1 <<...<< endl;
```

(Prints items listed between cout and endl)

Example:

```
cout << "the value of X is" << X << endl;
```

In this example, if X equals 5, then the output will be: *the value of X is 5*

Comparison operators

`X==5;`

(Is X equal to 5)

`X!=Y;`

(Is X not equal to Y)

`X<3;`

(Is X less than 3)

`X<=12;`

(Is X<=12)

`X>Z;`

(Is X greater than Z)

`X>=34;`

(Is X greater than or equal to 34)

`X>Y && X<7`

(Is X greater than Y and X less than 7)

`X>Y || X<7`

(Is X greater than Y or X less than 7)

Conditional statements

`if (comparison expression) statement;else statement;`

Single Statement Example:

```
if (X<=10)    X++; else    X=0;
```

(If X is less than or equal to 10 then add 1 to the value of X, otherwise set X to 0)

Multi Statement Example:

```
If (X<=10) {
```

```
    X++;
```

```
    printf ("%s \n", "X is less than 10");
```

```
} else {
```

```
    X=0;
```

```
    cout << "Finished" << endl;
```

```
}
```

(If X is less than or equal to 10 then add 1 to the value of X, and print the statement "X is less than 10". Otherwise set X=0 and print the statement "Finished".)

Incremental looping

for (Variable=Start Value; Comparison Statement; increment/decrement Variable)

{ Statement; Statement;...}

(Execute the statements as long as the variable is true in the comparison statement, then increment or decrement the variable. Variable starts at the defined Start Value.)

Example:

```
For (X=0; X<=10; X++)    cout << X << endl;
```

(If X is less than or equal to 10, then print the value of X at that time, then increment X.)

Conditional looping

While (Conditional Statement)

{ Statement; Statement;...}

(Execute the statements as long as the conditional statement is true)

Example:

```
X=0;
while(X<=10) {
    cout << X << endl;
    X++;
}
```

(X starts with the value of 0. While X is less than or equal to 10, print the value of X, and then add 1 to the value of X.)

Launching block operations

Operation_Name(parm_1, ...,parm_n);

(Launch the block operation Operation_Name with/without parameters.)

Examples:

go();

(Launch operation go without parameters)

min(x,y)

(Launch operation min with parameters X and Y)

Generating events

GEN(evName);

(Generate event evName and send to yourself.)

Examples:

GEN(evStart);

(Send event evStart to your own statechart)

GEN(evMove(10,X));

(Send event evMove with parameters 10 and X to have statechart respond.)

Generating port events

OUT_PORT(pName)->GEN(evName);

(Generate event evName and send it to the port pName)

Examples:

OUT_PORT(p2)->GEN(evStart);

(Send event evStart to port p2)

OUT_PORT(p2)->GEN(evMove(10,X));

(Send event evMove with parameters 10 and X to port p2)

Referencing event parameters

`params->event_parameter;`

(references value of event_parameter)

Examples:

`if (params->velocity <= 5)...`

(Test value of parameter velocity for an event to see if less than or equal to 5.)

Testing port for an event

`IS_PORT(port_name);`

(Returns TRUE if event that caused current activity flow arrived through port port_name)

Examples:

`if (IS_PORT(port_2))...`

(Test to see if event that caused current activity flow arrived through port_2, result is TRUE if yes, FALSE if no.)

Test to see if currently in a state

`IS_IN(state_name);`

(Returns TRUE if in state state_name)

Examples:

`if (IS_IN(Accelerating))...`

(Test to see if currently in state Accelerating. Result is TRUE if yes, FALSE if no.)

System validation

Rational Rhapsody enables you to visualize the model through simulation. *Simulation* is the execution of behaviors and associated definitions in the model. Rational Rhapsody simulates the behavior of your model by executing its behaviors captured in statecharts, activity diagrams and textual behavior specifications. Structural definitions like blocks, ports, parts and links are used to create a simulation hierarchy of subsystems.

Once you simulate the model, you can open simulated diagrams, which let you observe the model as it is running and perform design-level debugging. You can perform the following tasks:

- ◆ Step through the model
- ◆ Set and clear breakpoints
- ◆ Inject events
- ◆ Create an output trace

It is good practice to test the model incrementally using model execution. You can simulate pieces of the model as it is developed. This allows you to determine whether the model meets the requirements and find defects early in the design process. Then you can test the entire model. In this way, you iteratively build the model, and then with each iteration perform an entire model validation.

Creating a component

A *component* is a level of organization that names and defines a simulatable component. Each component contains configuration and file specification categories, which are used to build and simulate model.

Each project contains a default component, named `DefaultComponent`. You can use the default component or create a new component. In this example, you can rename the default component `Simulation`, and then use the `Simulate` component to simulate the model.

To use the default component:

1. In the browser, expand the `Components` category.
2. Select `DefaultComponent` and rename it `Simulation`.

Setting the component features

Once you have created the component, you must set its features.

To set the component features:

1. In the browser, double-click `Simulation` or right-click and select **Features**. The Component window opens.
2. The **Executable** radio button to set the Type.
3. If you used the common design package names, select `Analysis`, `Architecture`, and `Subsystems` as the Selected Elements. These are the packages for which you create a simulatable component. Do not select the `Requirements` package because you do not simulate it.
4. Click **OK**.

Creating a configuration

A component can contain many configurations. A *configuration* includes the description of the classes to include in code generation, and settings for building and Simulating the model.

Each component contains a default configuration, named `DefaultConfig`. In this example, rename the default configuration to `Debug`, and then use the `Debug` configuration to simulate the model.

To use the default configuration:

1. In the browser, expand the `Simulate` component and the `Configurations` category.
2. Select `DefaultConfig` and rename it `Debug`.

Preparing to Web-enable the model

The first step in Web-enabling a working Rational Rhapsody model is to set its configuration and elements as Web-manageable and then to simulate, build, and run the model.

Note

You cannot webify a file-based C model.

Creating a Web-enabled configuration

In this example, create a new configuration and then set its features as follows:

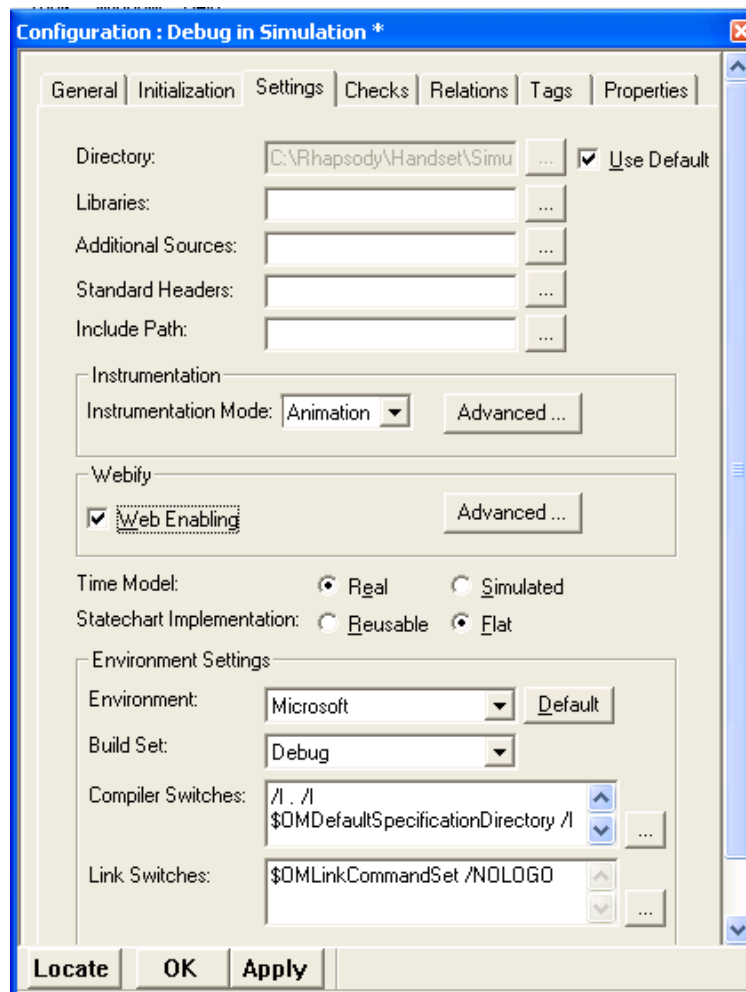
1. Right-click the **Configurations** category and select **Add New Configuration**.
2. Type **Panel**.
3. Double-click **Panel** or right-click and select **Features**. The **Features** window opens.
4. Select the **Initialization** tab and set the following values:
 - ♦ For the **Initial instances** box, select **Explicit** to include the classes which have relations to the selected elements.
 - ♦ Select **Generate Code for Actors**.
5. Click **Apply** to save these selections and keep the window open.
6. Select the **Settings** tab, and set the following values:
 - ♦ Select **Animation** from the **Instrumentation Mode** pull-down list. Rational Rhapsody adds instrumentation code to the simulated application, which enables you to simulate the model.
 - ♦ Select **Web Enabling** for **Webify**.
 - ♦ If wanted, click the **Advanced** button to change the default values for the Webify parameters. Rational Rhapsody opens the Advanced Webify iconkit Settings window.

This window contains the following boxes, which you can modify:

- **Home Page URL** is for the URL of the home page
- **Signature Page URL** is for the URL of the signature page
- **Web Page Refresh Period** is for the refresh rate in milliseconds
- **Web Server Port** is for the port number of the Web server
- ♦ Select **Real** (for real time) as the **Time model**.

- ♦ Select **Flat** as the **Statechart Implementation**. Rational Rhapsody implements states as simple, enumerated-type variables.

Rational Rhapsody fills in the **Environment Settings** section, based on the compiler settings you configured during installation. At this point the window should resemble this example.



7. Click **OK**.

Selecting elements to Web-enable

To Web-enable the model, set the elements that you want to control or manage remotely over the Internet using either the Rational Rhapsody **Web Managed** stereotype or the `WebManaged` property. To select elements to Web-enable:

1. To locate the items you want to change, choose **Edit > Search in Model**. Type the name into the **Find What** box and click **Find**. The search shows all instances of that text and the browser path for each.
2. Double-click the item located under the Subsystems browser category.
3. In the Features window, select **Web Managed** from the **Stereotype** pull-down list.
4. Click **OK**.
5. Make the same change to the remaining three events to make them Web Managed.

Note

If the element already has an assigned stereotype, set the element as Web-managed using a property. In the **Properties** tab, select `WebComponents` as the subject, then set the value of the `WebManaged` property within the appropriate metaclasses to `Checked`.

Connecting to the Web-enabled model

Rational Rhapsody includes a collection of default pages that serve as a client-side user interface for the remote model. When you run a Web-enabled model, the Rational Rhapsody Web server automatically simulates a Web site including the file structure and interactive capability. This site contains a default collection of simulated on-the-fly pages that refreshes each element when it changes.

Note

You can also customize the Web interface by creating your own pages or by referencing the collection of pages that come with Rational Rhapsody.

Navigating to the model through a Web browser

You can access a Web-enabled model running on your local machine or on a remote machine. In this example, you will connect to the model on your local machine.

To connect to the Web-enabled model on your local machine:

1. Open Internet Explorer.
2. In the address box, type the following URL:

`http://localhost`

Other users on the same network can connect to your local model using the IP address or machine name in place of `localhost`.

If you changed the Web server port using the Advanced Webify iconkit Settings window, type the following code:

`http://<localhost>:<port number>`

In this URL, `<localhost>` is `localhost` (or the machine name or IP address of the local machine running the MyProject model), `<port number>` is the port specified in the Advanced Webify icon kit Settings window.

By default, the Parts Navigation page of the Rational Rhapsody Web user interface opens.

Note

If you cannot view the right-hand frame in Internet Explorer, go to **Icons > Internet Options > Advanced** and clear the **Use Java xx for <applet>** check box.

Viewing and controlling a model

The Parts Navigation page provides easy navigation to the Web Managed elements in the model by displaying a hierarchical view of model elements, starting from the top level aggregate. By navigating to and selecting an aggregate in the left frame of this page, you can monitor and control your model in the *aggregate table* displayed in the right frame.

Aggregate tables contain name-value pairs of Rational Rhapsody Web-enabled elements that are visible and controllable through Internet access to the machine hosting the Rational Rhapsody model. They can contain text boxes, combo-boxes, and **Activate** buttons. You can monitor the model by reading the values in the dynamically populated text boxes and combo-boxes. You can control the model by pressing the **Activate** button, which initializes an event, or by editing writable text boxes.

Sending events to your model

You can simulate events in the Rational Rhapsody Web user interface and monitor the resulting behavior in the simulated diagrams.

1. If the simulated sequence diagram is not already open, simulate it and click the **Go** button.
2. If the simulated statechart is not already open, simulate it.
3. If the simulated activity diagram is not already open, simulate it.
4. Resize the Rational Rhapsody Web user interface browser window so that you can view the simulated diagrams while sending events to the model.
5. In the navigation frame on the left side of the browser, expand **ConnectionManagement_C[0]**, and click **ConnectionManagement_C::CallControl_C[0]**
6. In the Rational Rhapsody Web user interface, click **Activate** next to the starting point in the sequence diagram.
7. Open the simulated sequence diagram. Rational Rhapsody displays how the instances pass messages.
8. The simulated activity diagram shows activity flows from the active state to the inactive state.

You can continue generating events and viewing the resulting behavior in the simulated diagrams.

Importing DoDAF diagrams from Rational System Architect

Rational System Architect customers can import Rational System Architect DoDAF (non-ABM) to use in a Rational Rhapsody project. The SA Importer is a Rational Rhapsody Add On requiring a special license. The “SA Importer” requires type mapping between Rational Rhapsody and Rational System Architect elements using an external mapping file. This allows the users to modify the map to extend the import scope as wanted. The out-of-the-box scope of the import information into Rational Rhapsody as SysML elements.

Mapping the import scope

The external map file ties Rational System Architect elements to Rational Rhapsody elements and allows the users to analyze and adjust the imported data.

To map the import scope:

1. Locate the “SA Importer” map file (`RhpSAMap.xml`) in the Rational Rhapsody folder in the `AddOn\ProductIntegrator` directory.
2. Edit this file, using any XML editor, to map the two Rational System Architect element types to Rational Rhapsody elements.
 - ♦ **Relation** type maps only relation elements such as flows, links or dependencies.
 - ♦ **Element** type handles all other elements.
3. If two element types from the Rational System Architect are not sufficient, create new map entries for the wanted types or create a default element mapping entry under the default map.

Rational System Architect type mappings to Rational Rhapsody

This chart shows the supported Rational System Architect type mappings to the Rational Rhapsody metaclasses:

Rational System Architect Type	Rational Rhapsody Metaclass	Comments
SV-1 diagram	Package	Direct mapping through a map file
System Node	Block (SysML element)	Direct mapping through a map file
System Entity	Part (SysML element)	Direct mapping through a map file
System Function	Operation	Final conversion is made by a post processing script
System Interface	Flow	Direct mapping through a map file
System Data exchange	Flow item	Final conversion is made by a post processing script
Data Element	Attribute of a Part (a Part that stands for System entity)	Final conversion is made by a post processing script

Note: All elements that are not created through a direct mapping are mapped to Comments and the related attributes are kept as Tags.

Adding a default map entry

To create a default element mapping entry:

1. Open the “SA Importer” map file (RhpSAMap.xml) in an XML editor.
2. Add the following element:

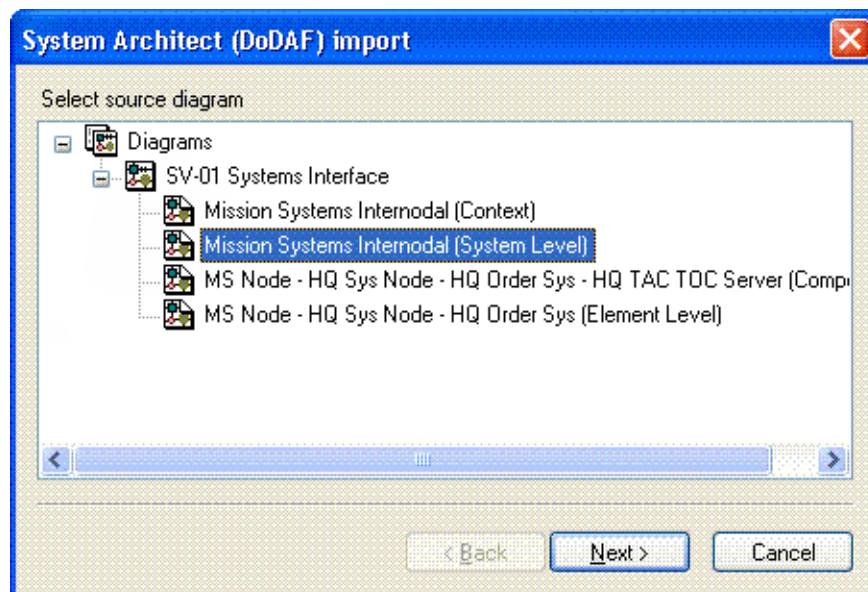
```
<element Rhapsody_MetaClass="Comment" SA_Type="*">
  <attribute Rhapsody_Field="Description"
SA_Property="Description"></attribute>
  <attribute Rhapsody_Tag="SAElementType"
SA_Property="TypeName"></attribute>
</element>
```

This mapping converts all Rational System Architect elements, that are not mapped explicitly into comments in Rational Rhapsody, with additional information from Rational System Architect about their original type kept as Tags.

Importing the Rational System Architect elements

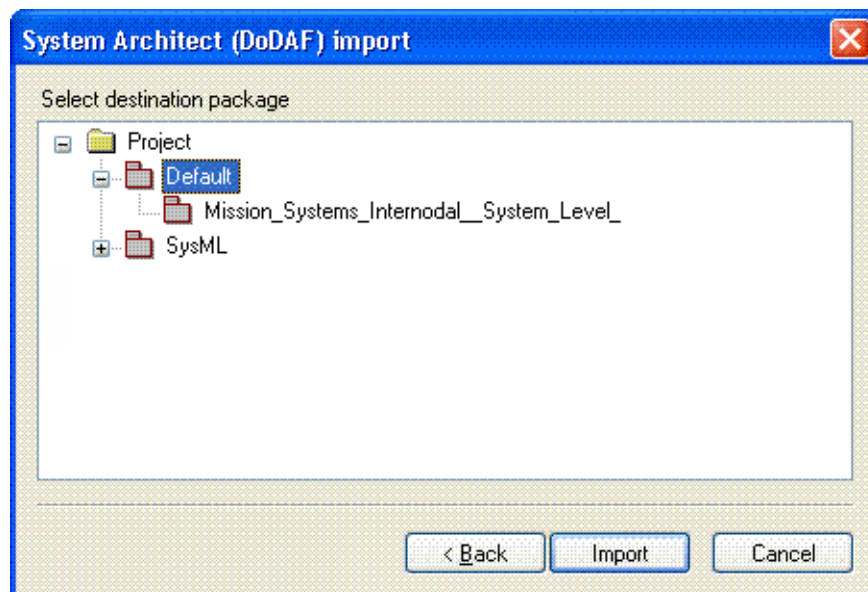
To run the SA Importer:

1. Launch Rational System Architect and load a DoDAF (non-ABM) encyclopedia.
2. Launch Rational Rhapsody.
3. Choose **Tools > Import from System Architect**. Rational Rhapsody launches the System Architect selection window.



4. Highlight the Rational System Architect diagram of interest and click **Next**.

5. In the next window, select the Rational Rhapsody package where the diagram should be placed in your Rational Rhapsody project. Click **Import**.



Only valid Rational Rhapsody elements are imported. The SA Importer puts the selected diagram data into Rational Rhapsody under the selected package and maintains the element hierarchy as it is in the Rational System Architect encyclopedia. The import operation also adds SysML profile characteristics to the project, unless it was already a SysML profile project.

Converting imported data into a Rational Rhapsody diagram

To convert the imported data into a Rational Rhapsody diagram:

1. Right-click the package that contains the imported data from Rational System Architect and select **Add New** and the SysML diagram type you want from the menu.
2. In the New Diagram window, enter the name of the diagram, select the **Populate Diagram** check box, and click **OK**.
3. On the Populate Diagram window, select the elements you want to add to the new diagram.
4. Click **OK**.

Post processing mechanism for Rational System Architect users

You can use the post processing mechanism, `SAIntegratorListenerPlug-in`, to perform analysis on the imported data. This Java plug-in is stored in the `<Rational Rhapsody installation path>\AddOn\ProductIntegrator\PostProcessing\SAIntegratorListenerPlugin`. For more information, see the `readme.txt` file in that folder.

Generating a Imported Elements report

To create an out-of-the-box SysML data flow report on the imported data:

1. Highlight the package holding the imported System Architect elements.
2. Choose **Tools > ReporterPLUS > Report on selected package**.
3. In ReporterPLUS, select to generate a Word or HTML report and use the `SysMLDataFlowInPackage.tpl` report template.
4. Follow the instructions in the ReporterPLUS wizard to generate the report.

Integration with Teamcenter systems engineering

Rational Rhapsody allows you to use its modeling abilities in conjunction with Teamcenter Systems Engineering from UGS.

The integration between the two tools allows you to work on the elements common to both Teamcenter and Rational Rhapsody models, such as requirements, use cases, and actors, from within either of the tools. Specifically, you can:

- ◆ Create a new Teamcenter design by importing an existing Rational Rhapsody model
- ◆ Generate Rational Rhapsody models from existing Teamcenter designs.
- ◆ Work on a project in Teamcenter, and then have the common elements updated automatically the next time you open the corresponding model in Rational Rhapsody.
- ◆ Work on common elements in the framework of a Rational Rhapsody model and then save the changes to the Teamcenter repository.

UML or SysML

Out of the box, you can use UML or SysML with the Teamcenter Interface and Rational Rhapsody. For Systems Engineers, this means you can interactively exchange information between Rational Rhapsody models using SysML and the Teamcenter Systems Engineering/ Requirements Management environment. For example, you can create and modify SysML elements (such as block and activity) defined in the Rational Rhapsody SysML profile.

In addition to new term SysML elements, you can modify the provided `ElementsMap.xml` file to map a Teamcenter element with a more domain-specific Rational Rhapsody element with one stereotype. This stereotype can be one of the predefined types in Rational Rhapsody or a user-defined stereotype.

To specify UML or SysML, you have to set the default in the `ElementsMap.xml` file, which handles the mapping between Teamcenter and Rational Rhapsody elements.

1. Open the `ElementsMap.xml` file (found in the Rational Rhapsody installation path, for example, `<Rational Rhapsody installation path>\AddOn\TcSE`) in a text editor.

2. Set either the **RhapsodyUMLTcSEUML** portion of the `ElementsMap.xml` file to be the default or set the **RhapsodySysMLTcSESysML** portion of the file to be the default. Do not set both as the default.

- For UML, the following figure shows RhapsodyUMLTcSEUML set as the default (Default="Yes"):

```
<RhapsodyTcSEMaps>
  <Map
    Name="RhapsodyUMLTcSEUML"
    Default="Yes"
    Rhapsody_domain="UML"
    TcSE_domain="UML"
    Description="">

    <element
      Rhapsody_MetaClass="Action"
      .
    </Map>
```

- For SysML, the following figure shows RhapsodyUMLTcSEUML set as the default (Default="Yes"):

```
<Map
  Name="RhapsodySysMLTcSESysML"
  Default="Yes"
  Rhapsody_domain="SysML"
  TcSE_domain="SysML"
  Description="">

  <element
    Rhapsody_MetaClass="Class"
    .
  </Map>
</RhapsodyTcSEMaps>
```

Note: If SysML is specified as the default in the `ElementsMap.xml` file and the Teamcenter design contains Rational Rhapsody SysML elements (meaning the map file has elements that contain `Rhapsody_Profile="SysML"`), then when you select **Open Model** or **New Model** in Teamcenter for Rational Rhapsody, Rational Rhapsody SysML will be used. However, if your Teamcenter design does not contain any Rational Rhapsody SysML elements, then Rational Rhapsody UML will be used even if SysML is set as the default in the map file.

Prerequisites for working with Rational Rhapsody

The prerequisites for working with Rational Rhapsody are as follows:

- ♦ For each Teamcenter project where you would like to use Rational Rhapsody integration, you must first import the provided Rational Rhapsody XML schema.
 - a. Right-click a Teamcenter project and select **Import > Import Schema**.
 - b. Navigate to the Rational Rhapsody installation folder path (for example, <Rational Rhapsody installation path>\AddOn\TcSE\Server).
 - c. Select the appropriate schema:
 - For UML, select **Rhp_Integration_Schema.xml**
 - For SysML, select **RhpSysML_Integration_Schema.xml**
- ♦ Only Teamcenter users with **Architect** permission can use Rational Rhapsody integration from within Teamcenter.

Importing a Rational Rhapsody model into Teamcenter

To create a new Teamcenter design by importing a Rational Rhapsody model:

1. In Teamcenter, right-click a folder and select **Rhapsody > Import Model**.
2. Select the appropriate Rational Rhapsody file.

A new Teamcenter design will be created, based on the relevant elements in the Rational Rhapsody model. The Rational Rhapsody model will also be attached to the Teamcenter design.

Note

If a Rational Rhapsody model element does not have a corresponding type in Teamcenter, but has children elements that do, these children elements will be ignored and will not be added to the Teamcenter design.

Creating a Rational Rhapsody model from existing Teamcenter Project

To create a new Rational Rhapsody model from an existing Teamcenter project:

1. In your Teamcenter project, add the Rational Rhapsody elements that you would like to use. (These elements are available after you have imported the provided Rational Rhapsody XML schema; see [Prerequisites for working with Rational Rhapsody](#).)
2. Right-click the relevant Teamcenter project folder and select **Rhapsody > New Model**.
3. Browse to where you want to save the Rational Rhapsody model.

A new Rational Rhapsody model is created, and all applicable elements in the folder are added to the Rational Rhapsody model. The name of the new Rational Rhapsody project will be the same as the name of the selected folder.

Note

If an element does not have a corresponding type in Rational Rhapsody, but has children elements that do, these children elements will be ignored and will not be added to the model.

Modifying shared elements from within Teamcenter

To modify elements shared with Rational Rhapsody from within Teamcenter:

1. In Teamcenter, right-click the relevant folder and select **Rhapsody > Open Model**.

Note: When the Rational Rhapsody model is opened, it is updated with any changes that other users might have made to the Teamcenter database, or that you might have made in Teamcenter before selecting **Open Model**.

2. Browse to where you want to save the Rational Rhapsody model.
3. Make your changes to the model.
4. Save your changes.
All changes made to shared elements will be applied to the corresponding Rational Rhapsody project as well.

View corresponding Rational Rhapsody element

To view the corresponding Rational Rhapsody element for a given Teamcenter element:

1. Right-click the element in Teamcenter and select **Rhapsody > Open Model**.
2. Browse to where you want to save the Rational Rhapsody model.

Rational Rhapsody is launched and the relevant Rational Rhapsody model element is displayed.

Note

This feature works only for the following model elements: object model diagrams, use case diagrams, collaboration diagrams, component diagrams, and sequence diagrams.

Modifying shared elements from within Rational Rhapsody

To modify shared elements from within Rational Rhapsody:

1. Make changes to the Rational Rhapsody model.
2. From Rational Rhapsody menu bar, select **File > Synchronize with TcSE**.

This saves any changes that were made to the Rational Rhapsody model and synchronizes the Teamcenter project with the Rational Rhapsody model.

Because changes made through Rational Rhapsody might conflict with changes made by other users to the Teamcenter database, any changes made from within Rational Rhapsody must be merged with changes that might have been made by other users. This is done by using the Rational Rhapsody Base DiffMerge feature. Each time the user opens the corresponding Rational Rhapsody model, a “base” version is also copied to the client machine. When the user saves their changes, Rational Rhapsody compares these changes and any changes contained in the current version from the server against the “base” version. In a rare case, where conflicts are found between the versions, these conflicts can be resolved using the Rational Rhapsody DiffMerge tool.

Limitations

Note the following limitations:

- ♦ Activity Flow, Flow, and Link relationships are not supported, as well as SysML new terms applied to these types.
- ♦ Attribute and Operation types are not supported, as well as SysML new terms applied to these types.
- ♦ The mapping between a Teamcenter element and a Rational Rhapsody element can only contain one stereotype.

The MicroC profile

The *MicroC profile* provides capabilities that are designed for C applications that will run on operating systems with very limited resources or systems with no operating system. These capabilities include:

- ♦ extended execution model
- ♦ a highly-efficient execution framework
- ♦ modeling of network ports
- ♦ optimizations for static systems
- ♦ segmented memory support
- ♦ monitoring of application running on target

The extended execution model

The extended execution model is implemented via:

- ♦ An OXF, called mxf.
- ♦ Profile-specific code generation behavior
- ♦ Changes to the Rational Rhapsody standard Features window that allows you to provide additional information for classes and objects.

MicroC code generation

The profile-specific code generation mechanism is used when the value of the property `General::Model::ExecutionModel` is set to `Extended`. When you create a project based on the MicroC profile, this is the default value for the property.

UI changes

When using the MicroC profile in a project, the Features window for classes and objects contains additional fields that allow you to provide the additional information required for classes and objects when using the extended execution model.

The mxf

mxf is a derivative of the standard Rational Rhapsody OXF. Unused functionality, such as cleanups and malloc'ed data, has been flagged out using compilation flags. The framework is also MISRA98-compliant.

All data and containers are statically defined.

Modeling network ports

The MicroC profile contains two “new terms” whose purpose is to allow you to bind data elements to signals on a bus:

- ♦ `inNetworkPort` - connects to an input signal from a bus
- ♦ `outNetworkPort` - connects to an output signal on a bus

Both of these types of network ports can be created in the browser and then be dragged to an object model diagram. (If you have applied the *ArchitectureDiagram* stereotype to the diagram, you can also create network ports by selecting the appropriate icon in the Diagram Tools.)

The following constraints apply to the creation of network ports:

- ♦ Network ports must be connected to a flow port on a Part, using a link.
- ♦ The network port must be of the same type as the flow port to which it is connected.

Adding a network port

To add a network port:

1. If you have applied the *ArchitectureDiagram* stereotype to the diagram, select the appropriate icon in the **Diagram Tools** and drag it to the diagram.

If you are using an object model diagram without the *ArchitectureDiagram* stereotype:

- a. In the browser, right-click the class that is the parent of the instance to which you will be connecting the port and select **Add New > General Elements > inNetworkPort** (or **outNetworkPort**).
 - b. Drag the network port to the relevant diagram.
2. Connect the network port to the flow port on the Part.
 3. Open the Features window for the network port, and provide values for the various fields.

Features window for network ports

The Features window for network ports contains the following fields:

- ♦ **Name** - the name of the network port
- ♦ **Stereotype** - currently not supported
- ♦ **Type** - the type of the signal on the bus, for example, int.

Note: In terms of type, the two types of network ports support unidirectional atomic types. This means that you cannot use non-atomic types, bi-directional ports, or multiplicity other than 1.

For inNetworkPorts, the Features window also contains the following network access fields:

- ♦ **Get API** - the API call to use to get the signal from the bus
- ♦ **Polling Mode** - allows you to choose synchronous polling or periodic polling (in synchronous, polling is handled by the execution manager that owns the network port in the hierarchy)
- ♦ **Polling Period** (if Periodic mode is selected) - interval at which the port will poll the input - in ticks
- ♦ **Polling Delay** (if Periodic mode is selected) - delay between system startup and the first polling action - in ticks

For outNetworkPorts, the Features window also contains the following network access fields:

- ♦ **Set API** - the API call to use to set the signal value on the bus

Optimizations for static systems

The MicroC profile also includes the following optimizations that are geared to static systems:

- ♦ Direct Flow ports
- ♦ Direct Relations
- ♦ ROMable Application
- ♦ Initial Value for Instance Attribute

Direct flow ports

Direct flow ports are flow ports which result in generated code that has a higher level of optimization than the code ordinarily generated by Rational Rhapsody for flow ports. This is done by using a direct connection between the source and target rather than using an interface-based

approach. The code is further optimized by connecting flow ports directly to a composition's inner parts where relevant.

The generation of this optimized code for flow ports is controlled by the following properties which are part of the MicroC profile, and are found under `C.CG::Configuration`:

- ◆ `DirectFlowPorts` - boolean property that turns the use of direct flow ports on/off
- ◆ `DirectFlowPortsInitializingStyle` - determines whether the flow ports are initialized at runtime or compile-time

Note that the value of the property `DirectFlowPortsInitializingStyle` affects code generation only if the property `InitializingMode` is set to `ByCategory`. Otherwise, the code generated is determined by the value of the property `InitializingMode`.

By default, the use of direct flow ports is turned on when using the MicroC profile.

Note that the optimization used for direct flow ports can only be applied to atomic (unidirectional) flow ports.

Direct flow ports differ from ordinary flow ports only in terms of the code generated. They are subject to the same constraints as ordinary flow ports: attribute name must match flow port name, type of the attribute in the sending and receiving objects must match.

Direct relations

The MicroC profile contains an option to generate optimized code for relations. When this optimization is used, Rational Rhapsody does not generate setters and getters for relations, these being unnecessary in static systems.

The generation of this optimized code for relations is controlled by the following properties which are part of the MicroC profile, and are found under `C.CG::Configuration`:

- ◆ `DirectRelations` - boolean property that turns the use of direct relations on/off
- ◆ `RelationInitializingMode` - determines whether the relations are initialized at runtime or compile-time

Note that the value of the property `RelationInitializingMode` affects code generation only if the property `InitializingMode` is set to `ByCategory`. Otherwise, the code generated is determined by the value of the property `InitializingMode`.

By default, the use of direct relations is turned on when using the MicroC profile.

This optimization for relation code can only be used for relations where each end has multiplicity of no more than 1. For relations where multiplicity greater than 1 is used, Rational Rhapsody will generate its standard code for relations (not the optimized code) regardless of the values of the relevant properties.

Monitoring of application running on target

The target monitoring feature adapts the Rational Rhapsody animation feature for use with targets that have very limited resources.

This feature allows you to monitor the status of the application running on the target, but does not allow you to provide input to the application.

Since communication is only one-way, this feature requires only minimal instrumentation code in the application

Using target monitoring

Use of the target monitoring feature involves:

1. instructing Rational Rhapsody to generate the appropriate instrumentation code
2. configuring the communication between Rational Rhapsody and the application running on the target
3. launching the monitoring process and viewing status in Rational Rhapsody

Generating instrumentation code

To have Rational Rhapsody generate the code required for target monitoring:

1. Open the Features window for the appropriate Rational Rhapsody configuration.
2. On the Settings tab, set the Instrumentation Mode to Animation.
3. Use the **Advanced** button to open the Advanced Instrumentation Settings window.
4. When the window opens, set Target Monitoring to On.

Now, when code is generated for this configuration, it will include the necessary instrumentation code for monitoring the application on the target.

Configuring communication with application on target

In order to receive updates from the application running on the target, you have to set the following properties to the appropriate values for your target:

- ◆ TargetProtocolBuildFlag - the protocol used by the target to send messages to Rational Rhapsody (for example, RS232 on Star12, RS232 on Windows, TCP on Windows)
- ◆ OnHostMessageReaderDLL - the message reader to be used by Rational Rhapsody for listening to messages from the target. For listening on RS232

port,(\$OMROOT)\DLLs\SerialMessageReader.Dll. For TCP/IP protocol, (\$OMROOT)\DLLs\TcpMessageReader.Dll.

- ♦ OnHostMessageReaderArguments - string of arguments for reading of messages by the host running Rational Rhapsody. For SerialMessageReader.Dll, the argument format is DeviceName:,BaudRate,DataBits,Parity,StopBits (for example, com1:,9600,8,n,1). For TcpMessageReader.Dll, the string consists only of a single integer indicating the port to listen on (if no changes were made to the file TargetMonitor.c, the default port is 24816)

Monitoring the application on the target

When using target monitoring, you should keep in mind that Rational Rhapsody does not control the running application in any way, so any of the animation controls provided only serve to control the display of incoming information by Rational Rhapsody. For example, you can pause the display of status information by Rational Rhapsody and then resume the display of the incoming information, even though the application itself keeps running on the target. The only way to control the running of the application itself is to use the debugger that is controlling the application on the target.

To monitor the application on the target:

1. To have the application pause at specific junctures, set breakpoints using the debugger controlling the application.
2. On the Start Target Monitoring toolbar, click the Start Target Monitoring button. This starts the translation proxy and puts Rational Rhapsody into animation mode. For animation to function properly, you must click this button before starting the application on the target via a debugger or other control.
3. To have Rational Rhapsody pause the display of incoming information at specific junctures (regardless of the progress of the application), use the Breakpoints button on the Animation toolbar.
4. Use the following buttons as required to control the display of data from the application:
 - Animation Break - allows you to pause the display of information coming from the target
 - Go - after pause, allows you to resume updating of information display
 - Go Step - after pause, allows you to resume updating of information display, one step at a time (using standard Rational Rhapsody definition of “step”)
5. Click the Stop Target Monitoring button on the Animation toolbar to have Rational Rhapsody exit animation mode.

Viewing MicroC properties

The MicroC profile uses a set of properties, contained in the file `MicroC.prp`, for controlling its various code optimization features and for controlling the target monitoring feature.

When using the MicroC profile, the list of filters used for viewing subsets of properties will include an entry called `MicroC Settings`. When you select this option, the properties that relate to the MicroC profile are displayed by category, with each category displayed on its own tab. This categorization of the properties is controlled by the `Dialog::All::PropertiesPerspectives` property.

IBM Rational Rhapsody DoDAF Add On

The IBM Rational Rhapsody for *Department of Defense Architectural Framework (DoDAF)* Add On provides industry standard diagrams and notations for developing DoDAF-compliant architecture models. These diagrams and notations are easily communicated and understood by a wide audience, greatly improving the comparability and communicability of architectures while ensuring the interoperability of systems.

Rational Rhapsody for DoDAF Add On is a semantic framework for developing, representing, and integrating architectures in a consistent way for applications for the Department of Defense (DoD). For information on the Department of Defense Architectural Framework (DoDAF) Specification, see the documents at www.defenselink.mil/cio-nii/cio/earch.shtml.

Rational Rhapsody for DoDAF Add On is part of the System Engineering Add-on component that might have been added during the Rational Rhapsody installation process (according to your Rational Rhapsody license). Or, if you purchased the Add On after your initial installation of the Rational Rhapsody product, you must run the Add On separately with a license key. See the Rational Rhapsody installation instructions and system requirements.

Note

If you want to import IBM Rational System Architect for DoDAF Add On diagrams as Rational Rhapsody SysML diagrams, see [Importing DoDAF diagrams from Rational System Architect](#).

Rational Rhapsody for DoDAF Add On and profile

The Rational Rhapsody for DoDAF Add On includes a DoDAF Profile, a number of DoDAF helper utilities, a DoDAF Reporter Template, a Microsoft Word Document Template file, a Rational Rhapsody ReporterPLUS License, an image library with a set of public domain graphics for military applications, and a tutorial.

To provide an effective Model Driven Development Solution for creating DoDAF-compliant architectural models, use the Rational Rhapsody for DoDAF Add On together with Rational Rhapsody in conjunction with a sound Systems Engineering Process and Methodology.

The Rational Rhapsody for DoDAF Add On is an independent process, but it also supports a variation of the Harmony development process targeted at the development of DoDAF-compliant

architecture models. The Rational Rhapsody for DoDAF Add On is a template-driven solution that can be customized and extended to meet specific customer requirements and development processes.

By simulating the Rational Rhapsody model, the ability of an architecture to meet its operational goals can be measured, and its effectiveness in comparison with other architecture models can be observed. The operational scenarios captured as event traces can be executed against the model, and the response of the architecture model can be recorded. In addition, using the automated testing capabilities in Rational Rhapsody, robustness of an architecture model can be analyzed, and a suite of functional verification tests can be generated from the model.

DoDAF views

DoDAF defines the following views:

- ♦ [Operational view](#), which identifies what needs to be accomplished and who does it
- ♦ [Systems view](#), which relates systems and characteristics to operational needs
- ♦ [Technical view](#), which prescribes standards and conventions
- ♦ All View, which encompasses all of the other views as there are overarching aspects of architecture that relate to the Operation, Systems, and Technical views

Operational view

The Operational view is a description of the tasks and activities, operational elements, and information exchanges required to accomplish DoD missions. DoD missions can include both military missions and business processes.

The Operational view contains graphical and textual elements that comprise an identification of the operational nodes, assigned tasks and activities, and information flows required between nodes. It defines the following aspects of communication:

- ♦ Types of information exchanged
- ♦ Frequency of exchange
- ♦ Tasks and activities supported by the information exchange

Operational views can describe activities and information exchanges at any level of detail and to any breadth of scope that is appropriate. The detail level is driven by the information required to perform the intended analyses. The kind of analysis you want to do determines what kind of information and the level of detail you must put into the Operational view.

Systems view

According to the Department of Defense, a “system” might be partially or fully automated and is defined as “any organized assembly of resources and procedures united and regulated by interaction or interdependence to accomplish a set of specific functions.”

The Systems view relates the system resources to the operational capabilities described in the Operational view. Further detail of the information exchanges described in the Operational view is provided in order to:

- ◆ Translate node-to-node exchanges into system-to-system transactions
- ◆ Communicate capacity requirements
- ◆ Show security protection needs

The Systems view describes systems and interconnections providing for, or supporting, DoD functions. DoD functions include both warfighting and business functions.

The systems, shown in the Systems view, can be existing, emerging, planned, or conceptual, depending on the purpose of the architecture effort. This view might be a reflection of the current state, transition to a target state, or analysis of future investment strategies.

Technical view

The Technical view is the minimal set of rules governing system parts and elements. It governs the following aspects of the parts:

- ◆ Arrangement
- ◆ Interaction
- ◆ Interdependence

The purpose of the Technical view is to ensure a system satisfies a specified set of requirements.

The Technical view provides the basis for the engineering specification of the systems in the Systems view and includes technical standards. The Technical view is the engineering infrastructure that supports the Systems view.

All views

All Views encompasses all of the other views as there are overarching aspects of architecture that relate to the Operation, Systems, and Technical views.

Products included in the Rational Rhapsody for DoDAF Add On

The following table lists the products that the Rational Rhapsody for DoDAF Add On includes.

Architecture Product	View	Product Name	Product Description
All Views	Package	AllViews	This optional stereotyped package allows you to add in AV products and other views and packages, if wanted.
AV-1	All	Overview and Summary Information	This product is typically a text (Word, FrameMaker, HTML) document. You can add AV-1 documents and launch them by clicking on them.
AV-2	All	Integrated Dictionary	This is a DoDAF-generated text product (report).
Operational View	Package		This optional stereotyped package is similar to the All View product. It supports all the operational products.
OV-1	Operational	High-Level Operational Concept Graphic	This high-level graphical/ textual description of the operational concept allows you to import pictures and other operational elements, such as Operational Nodes, Human Operational Nodes, Operational Activities and the relations among them.
OV-2	Operational	Operational Node Connectivity Description	This product shows the connections and flows among operational nodes and operational activities. If wanted, the behavior of operational nodes and operational activities can be shown by adding OV-5, OV-6a, OV-6b, and OV-6c diagrams. These diagrams are the primary source of information used by the Rational Rhapsody for DoDAF Add On to create the OV-3 diagram.

Architecture Product	View	Product Name	Product Description
OV-3	Operational	Operational Information Exchange Matrix	This product shows information exchanged between nodes, and the relevant attributes of that exchange. OV-3 is generated from the information shown in OV-2 and other operational diagrams. This information is stored as a CSV file and can be added to any product.
OV-5	Operational	Operational Activity Model	This product details the behavior of operational nodes or more commonly, operational activities.
OV-6a	Operational	Operational Rules Model	This product is a textual description of "business rules" for the operation. It is a controlled file. One of three products used to describe the mission objective.
OV-6b	Operational	Operational State Transition Description	This product is a statechart that can be used to depict the behavior of an operational element (node or activity). One of three products used to describe the mission objective.
OV-6c	Operational	Operational Event Trace Description	This product is a sequence diagram that captures the behavioral interactions among and between operational elements and (in the Harmony process) captures the operational contracts among them. One of the three products used to describe the mission objective.
OV-7	Operational	Logical Data Model	This product is a class diagram that shows the relations among Informational Elements (data classes). This is similar to entity relationship diagrams, but is more powerful.
System View	Package		This optional stereotyped package is similar to other views, but contains system elements.
SV-1	Systems	Systems Interface Description	This product is a diagram that contains System nodes, systems, system parts and the connections between them (links). These can be used with or without ports.

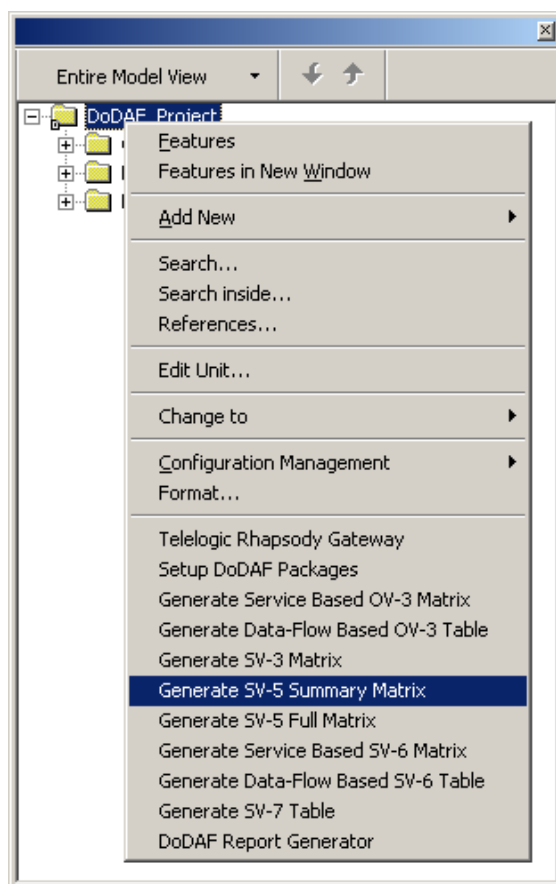
Architecture Product	View	Product Name	Product Description
SV-2	Systems	Systems Communications Description	This product is a diagram that shows the connections among systems via the communications systems and networks.
SV-3	Systems	Systems-Systems Matrix	This product is generated from the information in the other system views. SV-3 assumes that there are links between items stereotyped SystemNode, System, or System Part and represents these in an N ² diagram (system-system matrix).
SV-4	Systems	Systems Functionality Description	This product represents the connection between System Functions and Operational Activities. The connection is made by drawing a Realize dependency line from the System Function to the Operational activity on the diagram. System Functions are mapped onto the system elements that support them by making System Functions parts of the system elements (that is, System Functions are drawn within the other system elements). System elements can also realize system functions. Note that here, as in almost all the other views, you can use Performance Parameters (bound to their constrained elements via anchors) to add performance data. This is summarized in SV-7.
SV-5	Systems	Operational Activity to Systems Function Traceability Matrix	This product is a spreadsheet-like generated view summarizing the relations among system elements (system nodes, systems and system parts), system functions that they support, and the mapping to operational activities.
SV-6	Systems	Systems Data Exchange Matrix	This product shows the information in the flows (information exchanges) between system elements. They might be embedded flows (bound to the links) or they might be flows independent of links. This is a spreadsheet-like generated product.

Architecture Product	View	Product Name	Product Description
SV-7	Systems	Systems Performance Parameters Matrix	This is a generated spreadsheet-like product, showing all the performance parameters and the elements that they constrain.
SV-8	Systems	Systems Evolution Description	This product is the system evolution description. This is an activity diagram (there is a SystemProject element stereotype to serve as the "base" for this activity diagram). SV-8 depicts the workflow for system development, object nodes for products released, and performance parameters for things like start and end dates, slack time, and so on.
SV-9	Systems	Systems Technology Forecast	This product is a text document - a stereotype of a Controlled File.
SV-10a, SV-10b, SV-10c	Systems	Systems Rules Model, Systems State Transition Description, Systems Event Trace Description	These products are similar to the OV-6a, OV-6b, and OV-6c products, but they are separately identified, even though they are structurally identical.
SV-11	Systems	Physical Schema	This product is similar to the OV-7 class diagram. This product uses a class diagram to show physical schema (data representation).

Rational Rhapsody for DoDAF Add On helper utilities

There are a number of helper utilities provided with the Rational Rhapsody for DoDAF Add On. These precompiled helpers assist with common functions. They include helpers to generate the derived products OV-3, SV-3, SV-5, SV-6, and SV-7.

To activate a helper, right-click the applicable model element and select a helper from the pop-up menu. In the following figure, the model element selected is the top-level project folder:



The following table summarizes the helpers and the model elements for which they are available. The **Applicable To** column indicates which model element you must right-click to make the helper appear on the pop-up menu. For more information about the helpers, see [Manually adding the Rational Rhapsody for DoDAF Add On helpers](#).

Helper Name	Applicable To
Setup DoDAF Packages	DoDAF Project
Create OV-2 from Mission Objective	Mission Objective
Create OV-6c from Mission Objective	Mission Objective
Update OV-2 from OV-6c	OV-6c Event Trace
Generate Service Based OV-3 Matrix	DoDAF Project
Generate Data-Flow Based OV-3 Table	DoDAF Project
Generate SV-3 Matrix	DoDAF Project
Generate SV-5 Summary Matrix	DoDAF Project
Generate SV-5 Full Matrix	DoDAF Project
Generate Service Based SV-6 Matrix	DoDAF Project
Generate Data-Flow Based SV-6 Table	DoDAF Project
Generate SV-7 Table	DoDAF Project
DoDAF Report Generator	DoDAF Project

Setup DoDAF packages

The Setup DoDAF Packages helper is used to configure a new DoDAF project with helpers that are useful in building a DoDAF-compliant architecture model. In addition to doing basic configuration of the project, the helper creates a framework for the DoDAF project. This helper also creates two (empty) OV-1s; one is meant to be “semantic free” and the other “semantic rich.”

Create OV-2 from Mission Objective

The Create OV-2 from Mission Objective helper is used to create an OV-2 Operational Node Connectivity Description product, and associates the OV-2 with the selected mission objective. Operational nodes can be dragged onto the diagram to represent the operational nodes in the OV-2.

Create OV-6c from Mission Objective

The Create OV-6c from Mission Objective helper is used to create an initial OV-6c Operational Event Trace Description product, and associates the OV-6c with the selected mission objective. The OV-6c includes the operational nodes associated with the mission objective as lifelines.

Update OV-2 from OV-6c

The Update OV-2 from OV-6c helper is used to add operational activity allocations, along with needline and information exchanges after realizing the messages in an OV-6c diagram. Ports are added to the operational activities, and interfaces created and attached to the ports. These interfaces form the required and provided interface contracts between operational nodes.

Generate Service Based OV-3 Matrix

For information on the Generate Service Based OV-3 Matrix, see [Generating the OV-3 Operational Information Exchange Matrix](#).

Generate SV-3 Matrix

The Generate SV-3 Matrix is used to create a report that is based on the links between system elements in SV-1s and SV-4s, whether or not ports are used or whether interfaces for those ports are formally defined.

Generate SV-5 Summary Matrix

The Generate SV-5 Summary Matrix helper is used to create a summary to show a system function or system element row if and only if there is a dependency to an operational element. The most common dependency is Realization.

Generate SV-5 Full Matrix

The Generate SV-5 Full Matrix helper is used to create a report to show all system functions and element.

Rational Rhapsody for DoDAF Add On Report Generator

The Rational Rhapsody for DoDAF Add On Report Generator helper generates a Microsoft Word Document (.doc) file containing DoDAF architecture products from a Rational Rhapsody model. The process of generating a document is largely push button, and requires minimal user interaction. The content of the document is extracted from the Rational Rhapsody model created by the user. The resulting document is organized as follows:

- ◆ AV-1 Overview and Summary Information
- ◆ AV-2 Integrated Dictionary
- ◆ OV-1 High Level Operational Concept Graphic
- ◆ OV-2 Operational Node Connectivity Description
- ◆ OV-3 Operational Information Exchange Matrix
- ◆ OV-5 Operational Activity Model
- ◆ OV-6a Operational Rules Model
- ◆ OV-6b Operational State Transition Description
- ◆ OV-6c Operational Event-Trace Description
- ◆ OV-7 Logical Data Model
- ◆ SV-1 Systems Interface Description
- ◆ SV-2 Systems Communications Description
- ◆ SV-3 Systems-Systems Matrix
- ◆ SV-4 Systems Functionality Description
- ◆ SV-5 Operational Activity to Systems Function Traceability Matrix
- ◆ SV-6 Systems Data Exchange Matrix
- ◆ SV-7 Systems Performance Parameters Matrix
- ◆ SV-8 Systems Evolution Description
- ◆ SV-9 Systems Technology Forecast
- ◆ SV-10a Systems Rules Model
- ◆ SV-10b Systems State Transition Description
- ◆ SV-10c Systems Event-Trace Description

- ♦ SV-11 Physical Schema
- ♦ TV-1 Technical Standards Profile

For more information on the Rational Rhapsody for DoDAF Add On Report Generator helper, see [Generating the DoDAF report from the architecture model](#).

Rational Rhapsody project for Rational Rhapsody for DoDAF Add On configuration

You specify the Rational Rhapsody model elements that are to form the core views in the generated DoDAF documentation. From these, other DoDAF products are derived. The Rational Rhapsody model elements included in the Rational Rhapsody for DoDAF Add On generated report are specified using stereotypes provided in the Rational Rhapsody for DoDAF Add On profile. The DoDAF profile also provides tags that can be used to specify the location of external data and graphics that should appear in the DoDAF products. The Rational Rhapsody for DoDAF Add On includes a helper utility to create a Rational Rhapsody project with the DoDAF profile preloaded so you.

Creating a Rational Rhapsody for DoDAF project

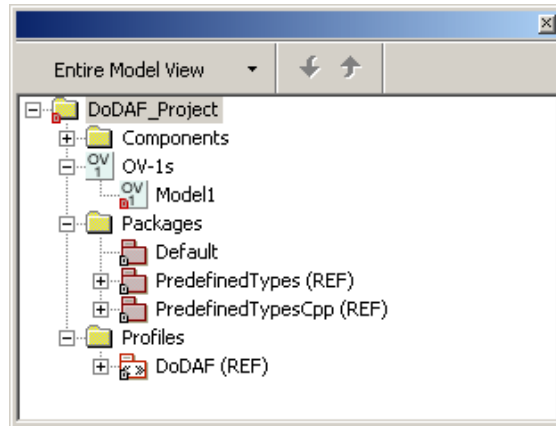
To create a Rational Rhapsody for DoDAF project:

1. Launch Rational Rhapsody and select **File > New**.
2. On the New Project window:
 - ♦ Type in your project name.
 - ♦ Specify a location.
 - ♦ Select DoDAF as the **Project Type** from the list. You might also select one of the **Project Settings**.

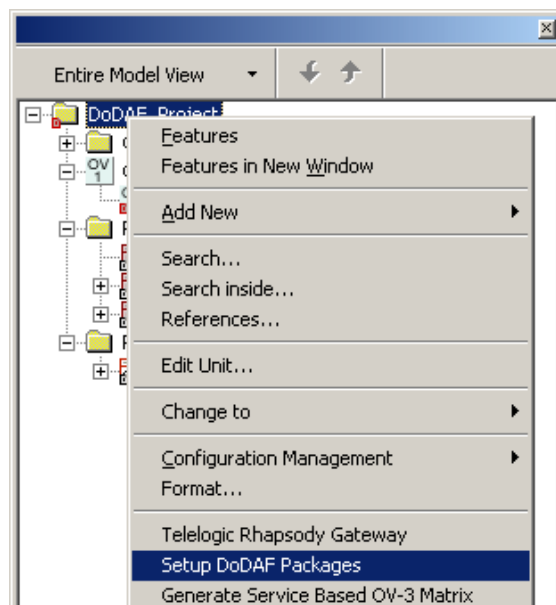
Note: This DoDAF type (or profile) is provided by the Rational Rhapsody for DoDAF Add On in order to help you customize and extend the Rational Rhapsody product to support a Domain Specific Language (DSL), which lets you work with DoDAF terms, diagrams, and artifacts rather than UML terms, diagrams, and artifacts.
3. Click **OK**.
4. If the folder you specified does not exist, you are asked if you want to create it. Click **Yes**.

5. Expand the **Packages** and **Profiles** folders in the Rational Rhapsody browser, as shown in the following figure.

Note: Profiles shown in your browser can vary depending on your site properties and product licensing.

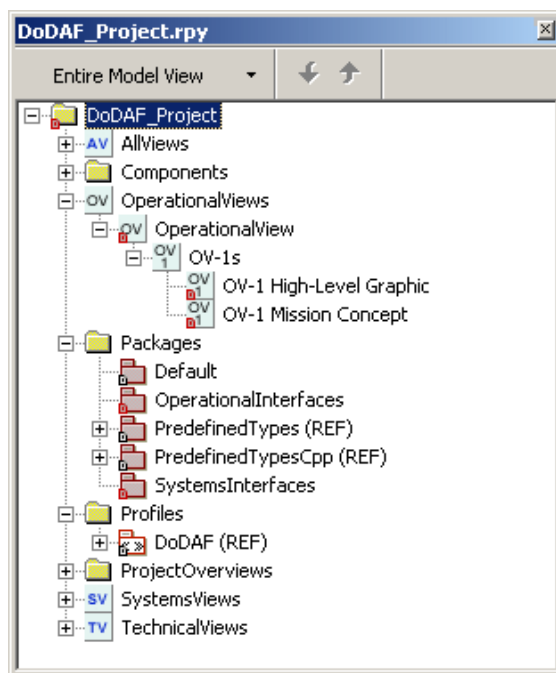


6. To initialize the Rational Rhapsody for DoDAF Add On project, right-click the top-level project name (**DoDAF_Project** in the example) and select **Setup DoDAF Packages**, as shown in the following figure:



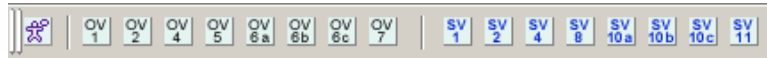
Note: If you do not see **Setup DoDAF Packages**, see [Manually adding the Rational Rhapsody for DoDAF Add On helpers](#).

7. Click **OK**.
8. Look at your Rational Rhapsody browser and notice what **Setup DoDAF Packages** did for your project. For example, expand the new **Operational Views** category until you see the **OV1-High-Level Graphic** and **OV-1 Mission Concept** mission object diagrams, as shown in the following figure:



Diagrams toolbar for a Rational Rhapsody for DoDAF project

The **Diagrams** toolbar, as shown in the following figure, provides quick access to the graphic editors, where diagrams are created and edited. The **DoDAF** profile that you used to create your Rational Rhapsody for DoDAF Add On project displays a **Diagrams** toolbar that is unique to this profile. The available diagrams are represented as icons on the toolbar across the top of the Rational Rhapsody window. To hide or display this toolbar, select **View > Toolbars > Diagrams**.



The following table shows all the diagram types with their icons, as displayed on the **Diagrams** toolbar for a project created with the **DoDAF** profile. Note that there are no icons on the **Diagrams** toolbar for the following diagrams:

- ◆ OV-3 is a derived product and is generated from the model.
- ◆ SV-3, SV-5, SV-6, and SV-7 are derived products and are generated from the model.
- ◆ AVs and TVs are controlled files and are added to the model.



Project Overview Diagram



OV-1: Hi Level Operational Graphic



OV-2: Operational Node Connectivity Diagram



OV-4: Organizational Relationships Diagram



OV-5: Operational Activity Diagram



OV-6a: Operational Rules Model



OV-6b: Operational State Transition Description Diagram



OV-6c: Operational Event-Trace Description Diagram



OV-7: Logical Data Model



SV-1: System Interface Description



SV-2: System Communication Description Diagram



SV-4: System Functionality Description Diagram



SV-8: System Evolution Description Diagram



SV-10a: Systems Rules Model



SV-10b: System State Transition Description Diagram



SV-10c: System Event-Trace Description Diagram



SV-11: Physical Schema Diagram

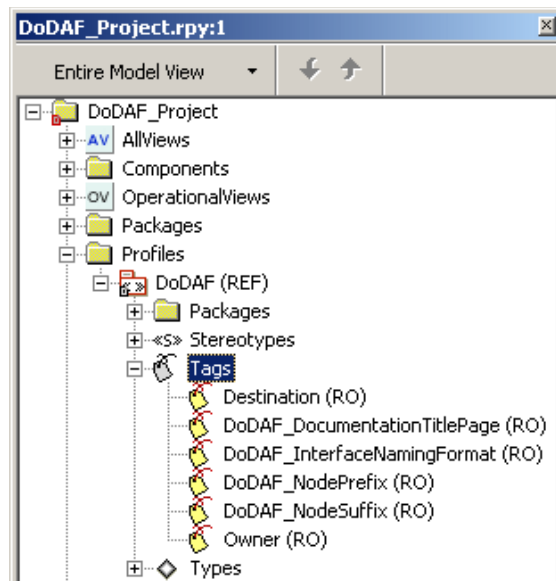
DoDAF tags

The DoDAF profile allows for the application of tags to diagrams, elements, and relations. These tags can be accessed from the Rational Rhapsody Browser or from the diagram, element, or relation itself.

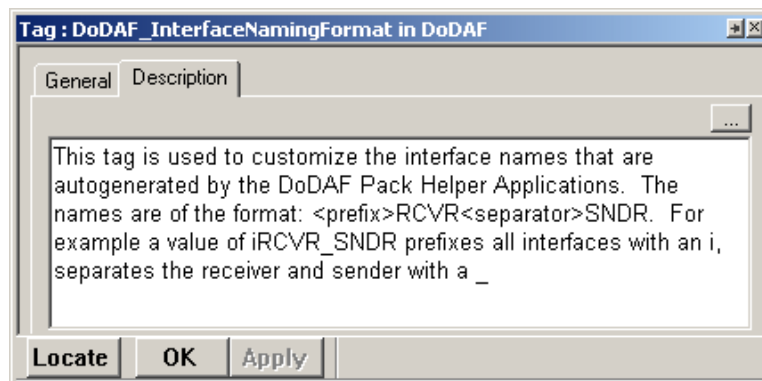
Accessing tags through the Rational Rhapsody browser

To access the tags from the browser:

1. Navigate to the applicable package stereotype by expanding the folders, as shown in the following figure:



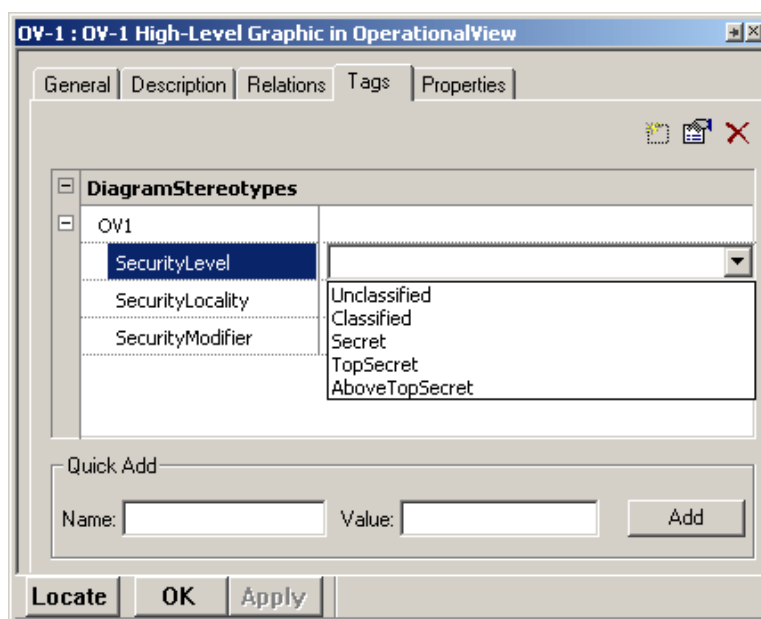
2. Double-click a tag to its Features window where you can view information applicable to it, such as its description, as shown in the following figure:



Accessing tags from a diagram, element, or relation

To access a tag from a diagram, element, or relation itself:

1. Right-click the item you want in the Rational Rhapsody browser and select **Features** to open its Features window.
2. To assign a value to a tag, on the **Tags** tab, click in the cell to the right of the tag select a value from a drop-down list, as shown in the following figure:



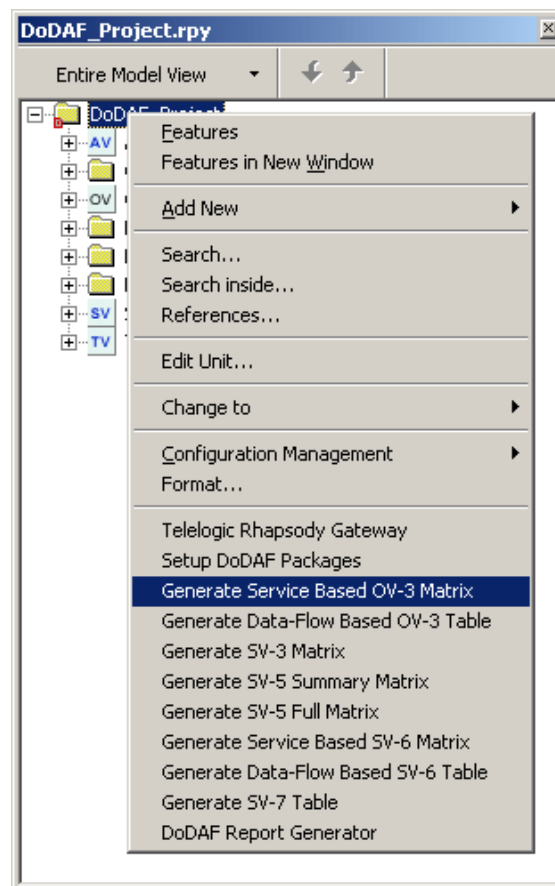
Note: To add a tag locally (meaning for use only by the current element, use the **Quick Add** group: Enter a name for the tag and a value, then click the **Add** button.

Generating the OV-3 Operational Information Exchange Matrix

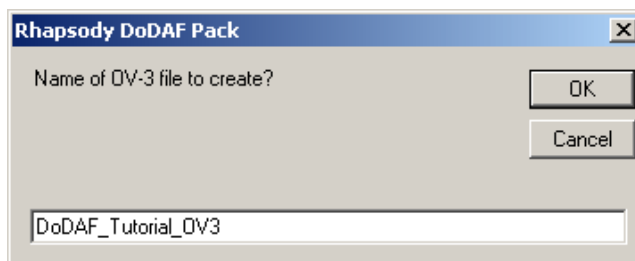
The OV-3 Operational Information Exchange Matrix provides a detailed report of the information exchange between operational nodes.

To automatically generate the OV-3 Matrix:

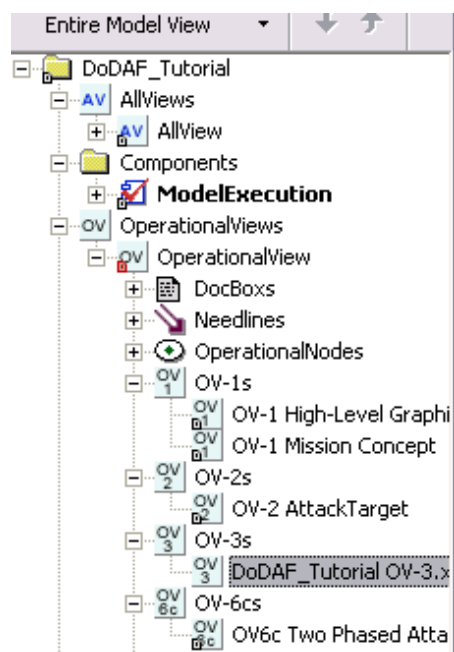
1. Right-click the top-level project folder (**DoDAF_Project** in the example) and select **Generate Service Based OV-3 Matrix**, as shown in the following figure:



2. Enter the name for the OV-3 file, as shown in the following figure, and click **OK**.



3. Wait while the matrix file is generated. Click **OK** to dismiss the confirmation message.
4. The browser now reflects the changes made.



Generating the DoDAF report from the architecture model

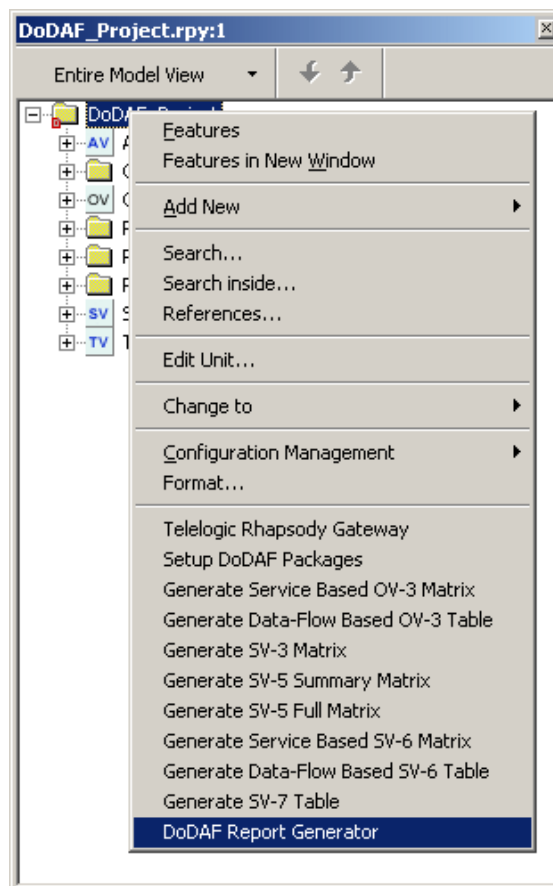
Using another helper, you can generate a Rational Rhapsody for DoDAF Add On report from the architecture model that includes the following architecture products: AV-1, AV-2, OV-1, OV-2, OV-3, OV-5, OV-6b, and OV-6c. The AV-2 is generated for you automatically from the architecture model data. Keep in mind that the Rational Rhapsody model itself is a dynamic and searchable AV-2 including all elements of the architecture model.

Note

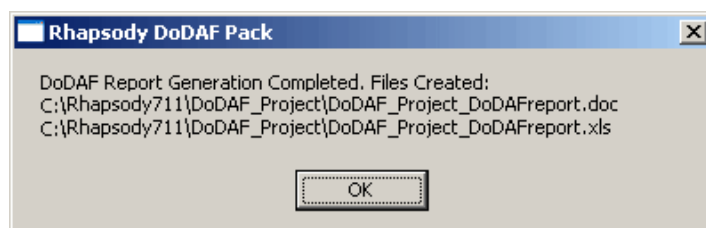
You must already have the Microsoft products mentioned in this section. They are not provide by the Rational Rhapsody DoDAF Add On or the Rational Rhapsody product.

To generate the Rational Rhapsody for DoDAF Add On report:

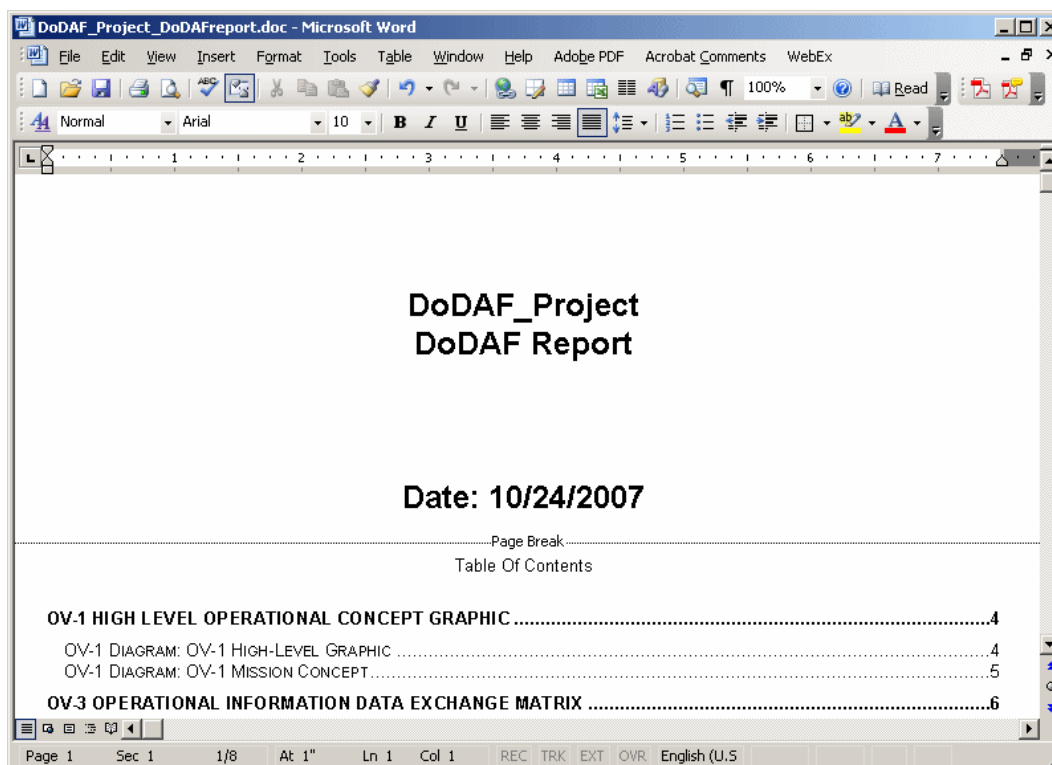
1. Right-click your top-level project folder on the Rational Rhapsody browser and select **DoDAF Report Generator**, as shown in the following figure:



2. Wait while your files are generated.
The Rational Rhapsody for DoDAF Add On Report Generator window displays and ReporterPLUS starts to load the model and generate the files. This process might take a few minutes. When completed, Rational Rhapsody display where the files are stored, which will be in the Rational Rhapsody project directory, as shown in the following figure:



3. Click **OK**.
4. Look at the files generated.
 - ♦ The document is formatted and displayed in Microsoft Word, as shown in the following figure:



- ♦ The OV-3 spreadsheet is saved as a worksheet in an Excel file

It is possible to navigate to the definition of any interfaces displayed in the OV-3 Matrix. Double-click an interface name in the OV-3 Excel file or in the OV-3 Word document will bring you to the corresponding definition of the interface in the AV-2 Data Dictionary. Use the Back button in Word's Web toolbar to return to the OV-3.

Note

The Word document can be converted to other formats including HTML and PDF using third party software (not provided in Rational Rhapsody).

Limitations

The Rational Rhapsody for DoDAF Add On has the following limitation. To use a mapping between Rational Rhapsody Diagrams (artifacts) and DoDAF Architecture products different than the one described here, the Helper Program and ReporterPLUS Template must be modified.

Troubleshooting

This section provides you with information that you can use for troubleshooting purposes.

Verifying the Rational Rhapsody for DoDAF Add On installation

If you are having a problem with the Rational Rhapsody for DoDAF Add On or if it does not appear in the Rational Rhapsody product as mentioned in this section, you should verify that it has been added.

Use any of the following methods to verify that Rational Rhapsody for DoDAF Add On has been added:

- ♦ See if there is a path to the Add On. From the Windows Start menu, select **All Programs > IBM Rational > IBM Rational Rhapsody *version number* > Rational Rhapsody DoDAF Add On**.
- ♦ See if the Add On has been included with the Rational Rhapsody product. Typically that path would be <Rational Rhapsody installation path>\DoDAF Pack.
- ♦ Create a project in Rational Rhapsody using **New > Project** and see if **DoDAF** is available from the Project Type (profile) pull-down menu.

If you find that the Rational Rhapsody for DoDAF Add On is not included in your system, then you must add it. You need a license key for the Rational Rhapsody for DoDAF Add On if it is not already a part of your Rational Rhapsody license.

If the Rational Rhapsody for DoDAF Add On is included in your system but you are having problems with it, you might need to remove and then run the installation program for the Add On to repair your software if it has been damaged.

Manually adding the Rational Rhapsody for DoDAF Add On helpers

Typically, when you add the Rational Rhapsody for DoDAF Add On, the installation program automatically configures the Rational Rhapsody for DoDAF Add On helpers for you. The following table summarizes the helpers and their settings:

Helper Name	Command	Arguments	Applicable To
Setup DoDAF Packages	<Path>\DoDAFPack.exe	-dodaf -i	DoDAF
Create OV-2 from Mission Objective	<Path>\DoDAFPack.exe	-dodaf -uc	MissionObjective
Create OV-6c from Mission Objective	<Path>\DoDAFPack.exe	-dodaf -sd	MissionObjective
Update OV-2 from OV-6c	<Path>\DoDAFPack.exe	-dodaf -d	OV-6c
Generate Service Based OV-3 Matrix	<Path>\DoDAFPack.exe	-dodaf -ov3m	DoDAF
Generate Data-Flow Based OV-3 Table	<Path>\DoDAFPack.exe	-dodaf -ov3t	DoDAF
Generate SV-3 Matrix	<Path>\DoDAFPack.exe	-dodaf -sv3	DoDAF
Generate SV-5 Summary Matrix	<Path>\DoDAFPack.exe	-dodaf -sv5short	DoDAF
Generate SV-5 Full Matrix	<Path>\DoDAFPack.exe	-dodaf -sv5long	DoDAF
Generate Service Based SV-6 Matrix	<Path>\DoDAFPack.exe	-dodaf -sv6m	DoDAF
Generate Data-Flow Based SV-6 Table	<Path>\DoDAFPack.exe	-dodaf -sv6t	DoDAF
Generate SV-7 Table	<Path>\DoDAFPack.exe	-dodaf -sv7	DoDAF
DoDAF Report Generator	<Path>\DoDAFPack.exe	-dodaf -report	Project

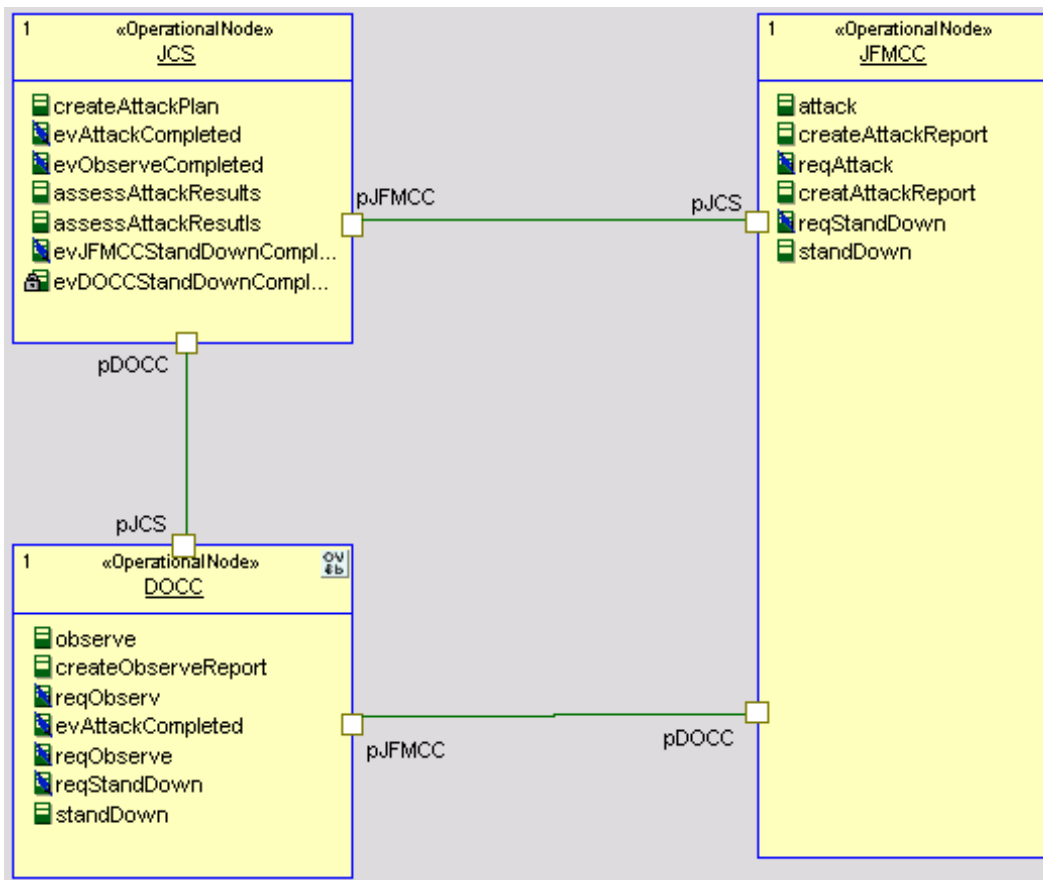
If you find that any of the Rational Rhapsody for DoDAF Add On helpers are missing from the submenus in Rational Rhapsody, first make sure you have verified the Rational Rhapsody for DoDAF Add On installation as described in [Verifying the Rational Rhapsody for DoDAF Add On installation](#). Once you have verified that it has been added, you can manually configure the helpers using the instructions that follow. While verifying the installation, make sure you have noted the path where the Rational Rhapsody for DoDAF Add On is located.

To add the Rational Rhapsody for DoDAF Add On helpers:

1. Open the Helpers window. Choose **Tools > Customize**.
2. On the Helpers window, use the New button  to create a new helper, and enter the appropriate helper name from the table above.
 - a. Set the **Command**, **Arguments**, and **Applicable To** boxes as listed for a helper on the table above. For the command, replace <Path> with the path where your Rational Rhapsody for DoDAF Add On is located.
 - b. Select the **External program** radio button and select the **Show in pop-up menu** check box.
3. Click **Apply**.
4. If needed, configure another helper by using the above steps, or click **OK** to close the Helpers window.

Correcting messages that appear as mission objectives

After updating the OV-2 from an OV-6c, you should check the OV-2 and make sure the messages between nodes are correctly appearing as events. If you find a message going between operational nodes displays in the OV-2 as an operation rather than an event, this indicates the message type was not changed to Event in the OV-6c diagram. The following figure shows an example of evJFMCCStandDownCompleted appearing with the private operation symbol , but should have the event symbol .

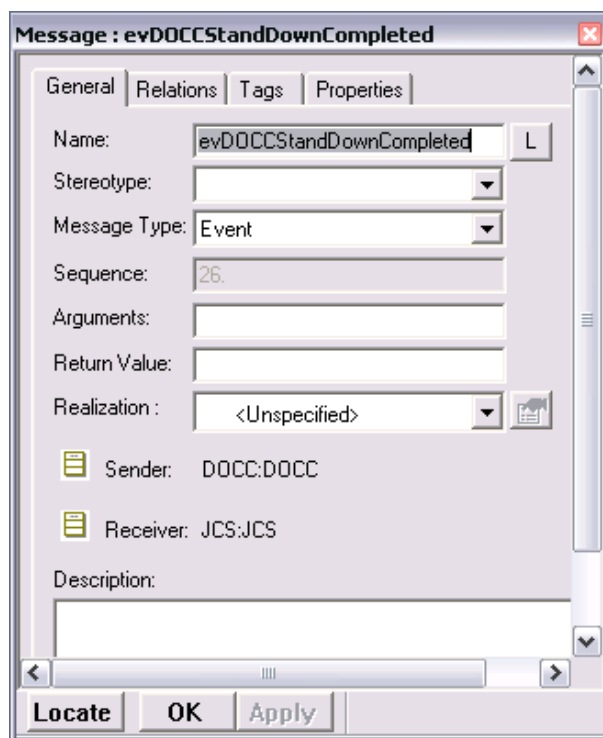


To fix this problem:

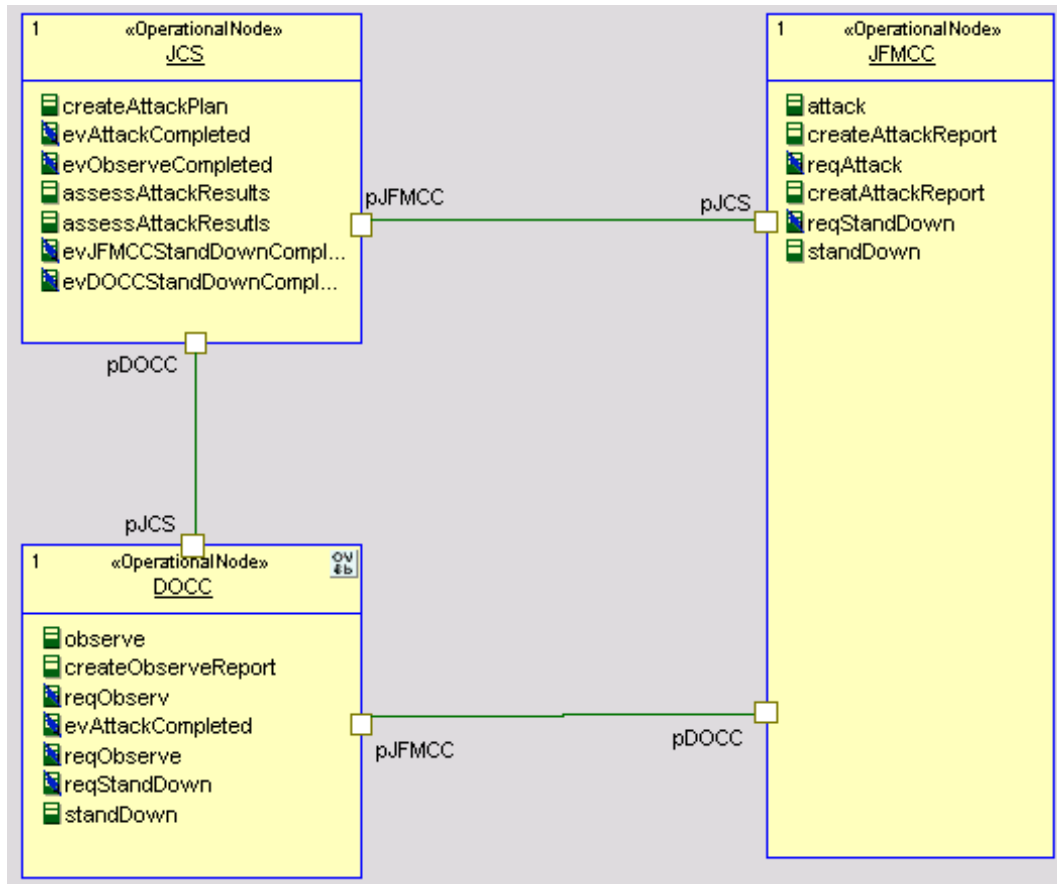
1. Expand the Rational Rhapsody browser to see the operation under the appropriate operational node. For example, expand the folder **Project_1 > OperationalViews > Operational View > Operational Nodes > JCS > Operations > evDOCCStandDownCompleted**.
2. Delete the operation by right-clicking the incorrect operation and selecting **Delete from Model**.
3. Click **Yes** to confirm your action.
4. Open the OV-6c diagram and check all the messages with the message name in question.

Note: To spot the problematic message, make sure messages that connect operational nodes have a hollow arrowhead , and mission objective messages (messages that start and end on the same node) have a solid arrowhead .

5. To change the incorrect message, change the message type:
 - a. Double-click the message to open the Features window.
 - b. On the **General** tab, in **Message Type** box, select **Event**, as shown in the following figure:



6. With the message selected, choose **Edit > Auto Realize** to realize the message.
7. Right-click **OV-6c Two Phased Attack** on the Rational Rhapsody browser and select **Update OV-2s from OV-6c**. You have repaired the OV-6c and OV-2 diagrams, as shown in the following.



View, caption, or table of figures is missing from document

If the final document does not include an OV-3 matrix, figure captions, or table of figures, the macros in the DoDAFReportRTF.dot Word document template file are not being executed. Make sure the security settings for Word are set to medium security to allow macros to be run. In Word, you can change the security setting using **Tools > Macro > Security**.

Microsoft Word might prompt you to enable macros in the generated Rational Rhapsody for DoDAF Add On report when it is opened. You will need to select **Enable Macros** button on this window when you open the DoDAF report in order for the macros to run.

IBM Rational Rhapsody MODAF Add On

The United Kingdom's Ministry of Defence Architecture Framework (known as MODAF) provides standards for enterprise architecture.

Enterprise architecture is the practice of applying a comprehensive and rigorous method for describing a current and/or future structure and behavior for an organization's processes, information systems, personnel, and organization sub-units, so that they align with the organization's core goals and strategic direction. It is effectively a structured approach to describing how a business works or is intended to work so that it can reach its primary objectives. Enterprise architecture is used typically by the military for capability procurement, by governments, and by large businesses.

An *architecture framework* is a specification of how to organize and present an enterprise architecture. It provides a means to present and analyze the enterprises problems. It does not generally tell you how to do something. Architecture frameworks tend to consist of a standard set of viewpoints that represent different aspects of an organization's business as it relates to a particular objective. In the context of MODAF, this implies a systems of systems approach as the analysis is complex and wide-ranging.

Large organizations use MODAF because it enables and facilitates the management of complex enterprise-wide implementations and promotes collaborative architecture development. While MODAF provides an enterprise architecture framework, it is not an architecture. The architecture is the result of using an architecture framework. Therefore, you must use MODAF in conjunction with a knowledge management approach and process that can regulate both the framework and the data.

For all organizations that use MODAF, a key feature of this framework is its goal to help estimate and reduce costs across all projects involved with the enterprise. In the context of military enterprises, the Ministry Of Defence (MOD) sees MODAF as key to the success of its Network Enabled Capability (NEC) goal.

Large frameworks have collections called viewpoints that contain views (also known as products). The viewpoints are inter-related and use elements of each others views. Modeling helps to manage the complexity and retain consistency between the views.

MODAF incorporates aspects of the widely used United States Department of Defense Architectural Framework (known as DoDAF). However, MODAF extends the scope of DoDAF to include viewpoints that reflect the interests of planners and procurement organizations. While DoDAF has four viewpoints (Operational, Systems, Technical, and All Views), MODAF has six with the addition of the Strategic and Acquisition viewpoints. MODAF uses the same views that it shares with DoDAF, though some might work differently in MODAF. For more information on DoDAF, see [IBM Rational Rhapsody DoDAF Add On](#).

MODAF is defined as a UML profile. For details of the MODAF MetaModel (or M3, as it is known), see <http://www.modaf.org.uk/>. Of the 35 views in MODAF, approximately 22 are expressible in UML. The remaining views (except AcV-1 and AcV-2, which are not supported in the IBM Rational Rhapsody for MODAF Add On) are either information that can be extracted from the model using the Tables and Matrices functionality in Rational Rhapsody or are text documents that can be added to the model.

For more information about MODAF, see the following Web sites:

- ♦ For technical and introductory material, go to <http://www.modaf.org.uk/>. This Web site contains the official online documentation for MODAF.
- ♦ For the history of MODAF, go to <http://www.modaf.com>.

Rational Rhapsody for MODAF Add On

The IBM Rational Rhapsody for MODAF Add On includes a MODAF profile, MODAF helper utilities, a model library to enable customization of certain table/matrix views so that you can define your own table/matrix view layouts and add your own custom table/matrix views, a MODAF ReporterPLUS template, a Rational Rhapsody ReporterPLUS license, a set of icons, and an image library with a set of public domain graphics for military applications. In addition, a sample project is available in <Rational Rhapsody installation path>\MODAF Pack\MODAF_ExampleModel.

The Rational Rhapsody for MODAF Add On might have been added during the Rational Rhapsody installation process (according to your Rational Rhapsody license). Or, if you purchased the Rational Rhapsody for MODAF Add On after your initial installation of the Rational Rhapsody product, you must install it separately with a license key. You must use the Rational Rhapsody installation wizard's Modify option to install the Rational Rhapsody for MODAF Add On after the initial Rational Rhapsody installation. See the Rational Rhapsody installation instructions for installation instructions and any system requirements.

When you create a new project in Rational Rhapsody, you identify a project type by selecting a profile, in this case, the MODAF profile. This means that you create your project so that it contains model elements that are customized for your specific domain or purpose. See [Configure a Rational Rhapsody project for MODAF](#).

To provide an effective Model Driven Development Solution for creating MODAF-compliant architectural models, use the Rational Rhapsody for MODAF Add On together with Rational Rhapsody in conjunction with a sound Systems Engineering Process and Methodology.

The Rational Rhapsody for MODAF Add On is process independent, but it can support a variation of the Harmony development process targeted at the development of MODAF-compliant architecture models. The Rational Rhapsody for MODAF Add On is a template-driven solution that can be customized and extended to meet specific customer requirements and development processes. For information about Harmony, see [Harmony process and toolkit](#).

MODAF viewpoints

Large frameworks have collections of common views called viewpoints (for example, the Systems viewpoint). Viewpoints are heavily interrelated as they use elements of each others views (for example, the SV-1 view has an element called a System that can be used or referred to in many other views).

Note that a view is also called a product.

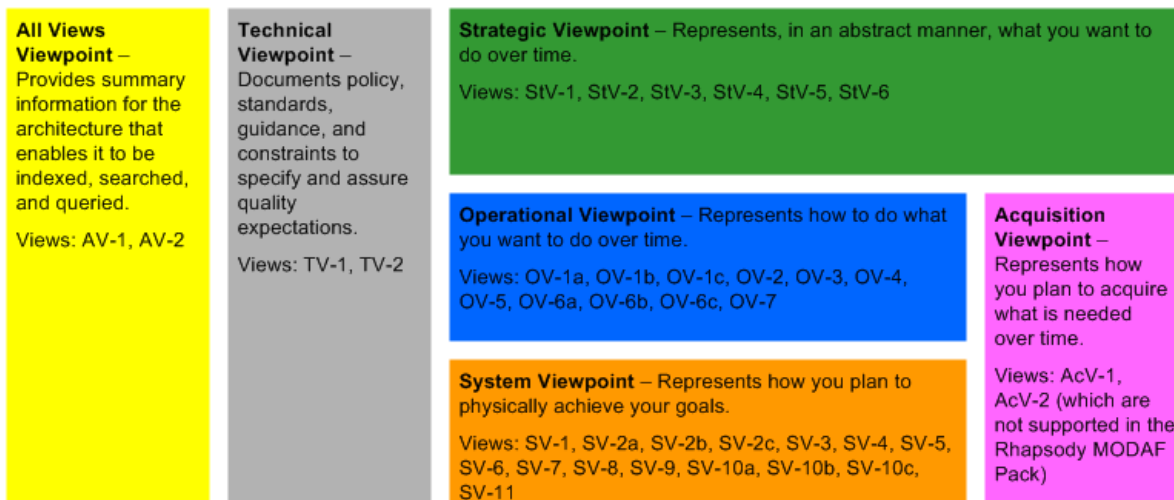
The MODAF viewpoints are:

- ♦ [All Views viewpoint](#). In both DoDAF and MODAF, and along with the Technical viewpoint, All Views (AV) provides summary information for the architecture that enables it to be indexed, searched, and queried. All Views encompasses all of the other views as there are overarching aspects of architecture that relate to the Strategic, Operation, Systems, Acquisition, and Technical viewpoints.
- ♦ [Strategic viewpoint](#). Specific to MODAF, the Strategic viewpoint (StV) represents, in an abstract manner, what you want to do over time. It documents the strategic picture of how a capability (for example, a military capability) is evolving in order to support capability deployment and equipment planning. The Strategic, Operational, and Systems viewpoints have a layered relationship.
- ♦ [Operational viewpoint](#). In both DoDAF and MODAF, the Operational viewpoint (OV) documents the operational processes, relationships, and context to support operational analyses and requirements development. The Operational, Strategic, and Systems viewpoints have a layered relationship.
- ♦ [Systems viewpoint](#). In both DoDAF and MODAF, the Systems viewpoint (SV) represents how you plan to physically achieve your goals. It documents system functionality and interconnectivity to support system analysis and through-life management (meaning it relates systems and characteristics to operational needs). The Systems, Strategic, and Operational viewpoints have a layered relationship.

- ♦ [Acquisition viewpoint](#). Specific to MODAF, the Acquisition viewpoint (AcV) is partly derived from elements of the Strategic viewpoint and provides information for the Operational and Systems viewpoints. The Acquisition viewpoint represents acquisition program dependencies, timelines, and the status of MOD Defence Lines of Development (DLOD, equivalent to U.S. Department of Defense DOTMLFPs) status so that the various MOD programs are managed and synchronized correctly. Note that the AcV-1 and AcV-2 views are not supported in the Rational Rhapsody MODAF profile.
- ♦ [Technical viewpoint](#). In both DoDAF and MODAF, and along with the All View viewpoint, this viewpoint documents policy, standards, guidance, and constraints to specify and assure quality expectations. It also covers all the other viewpoints.

The following illustration shows the MODAF viewpoints and how they relate to each other. In addition, each viewpoint includes a listing of their views.

MODAF Viewpoints and Their Views



You create viewpoints and views as needed. Which viewpoint you start with is up to you. Note that the viewpoints and their views do not have to be and are not expected to be created all at the same time.

All Views viewpoint

The All Views viewpoint encompasses all of the other viewpoints as there are overarching aspects of architecture that relate to the Strategic, Operational, Systems, Acquisition, and Technical viewpoints. All Views records what has happened and what should happen going forward. It provides a dictionary (through the AV-2 Integrated Dictionary view) for the architecture that guides the current developers of the architecture and helps future developers understand the framework going forward. This viewpoint is critical to the future success of the current MODAF architecture and any future architecture. Therefore, you should always use the views in this viewpoint for MODAF.

Typical stakeholders of the All Views viewpoint are enterprise planners.

Strategic viewpoint

The Strategic viewpoint is a description of the strategic picture of how capability (for example, military capability) is evolving in order to support capability management and equipment planning. It contains capability management and shows how capabilities map to operational concepts over time. Its main intent is to analyze the areas of capability gaps, overlaps, and redundancies, and to map capabilities to organizations and platforms. There is no reference to implementation in this viewpoint.

Typical stakeholders of the Strategic viewpoint are high-level planners, policy makers, and analysts.

Operational viewpoint

The views in the Operational viewpoint can describe activities and information exchanges at any level of detail and to any breadth of scope that is appropriate in logical terms. The detail level is driven by the information required to perform the intended analyses. The kind of analysis you want to do determines what kind of information and the level of detail you must put into the Operational viewpoint. The OV views re-use the capabilities defined in the StV views within content of an operation or scenario. The OV views are used during the various point of an enterprise's lifecycle, including the creation of current and future requirements, and during the planning phase for the operation.

Typical stakeholders of the Operational viewpoint are operation planners.

Systems viewpoint

The Systems viewpoint relates the system resources to the operational capabilities described in the Operational viewpoint. The views in the viewpoint describe the resources that help you archive the intended capability. They describe the system resources available and their interactions with each other. This includes the matter of human involvement in the operation of systems.

The systems shown in the Systems viewpoint can be existing, emerging, planned, or conceptual, depending on the purpose of the architecture effort. This viewpoint might be a reflection of the current state, transition to a target state, or analysis of future investment strategies.

A primary use of the views in the Systems viewpoint is to develop system solutions that address user requirements and therefore system requirements.

Typical stakeholders of the Systems viewpoint are members of an organization's acquisition group and its associated suppliers.

Acquisition viewpoint

The Acquisition viewpoint details how the various identified systems will be acquired over time as part of programs. This includes identifying dependencies among projects and capability integration across DLODs (in military endeavors). Note that although you can create views for this viewpoint, the Acquisition viewpoint's AcV-1 and AcV-2 views are not supported in the Rational Rhapsody for MODAF Add On.

Typical stakeholders of the Acquisition viewpoint are those involved in capability management and acquisition.

Technical viewpoint

The purpose of the Technical viewpoint is to ensure a system satisfies a specified set of requirements. The views in this viewpoint cover governance (standards, rules, policy, and so on) for all aspects of the architecture.

The Technical viewpoint provides the basis for the engineering specification of the systems in the Systems view and includes technical standards (though they do not have to be "technical"). The Technical viewpoint is the engineering infrastructure that supports the Systems viewpoint.

Typical stakeholders of the Technical viewpoint are policy makers and those charged with maintaining core interoperability standards.

Views Included in the Rational Rhapsody for MODAF Add On

The following table lists the views (within their viewpoints) included in the Rational Rhapsody for MODAF Add On.

For more information about these views, go to the official online documentation Web site for MODAF at <http://www.modaf.org.uk>.

Architecture Viewpoint/View	View	View Name	Description
All Views Viewpoint	Package		The All Views viewpoint contains views that provide overview and nomenclature.
AV-1	All	Overview and Summary Information	This view is typically a text document (for example, Word, FrameMaker, HTML) that provides overview and summary information for the operations and capabilities being considered for the enterprise. It scopes the architecture and gives it context. You can add AV-1 documents and launch them by clicking on them.
AV-2	All	Integrated Dictionary	This view presents all the elements used in an architecture as a standalone structure, generally using a specialization hierarchy. It should provide a text definition for each one and references the source of the element (for example, MODAF Ontology, IDEAS Model, local, and so on).
Strategic Viewpoint	Package		The Strategic Viewpoint contains views that detail military capabilities and how they evolve to be used by various organizations.
StV-1	Strategic	Enterprise Vision	This view defines the strategic context for a group of enterprise-level capabilities. It takes the overall enterprise vision and goals of the architects and enables them to relate these to realizable capabilities. StV-1 used to be a textual document but is now better represented in a more structured format as UML structure or class diagrams.

Architecture Viewpoint/View	View	View Name	Description
StV-2	Strategic	Capability Taxonomy	This view models capability hierarchies. It enables users to organize capabilities in the context of an enterprise phase, showing required capabilities for current and future enterprises. It specifies all the capabilities that are referenced throughout the current architecture and possibly other reference architectures. StV-2 is realized using UML structure or class diagrams.
StV-3	Strategic	Capability Phasing	This view shows when capabilities are expected to be used. It maps capabilities to time periods. It is also used to perform gap/overlap and redundancy analysis. StV-3 shows a customized table view, with the user defining the time periods.
StV-4	Strategic	Capability Dependencies	Similar to StV-2, this view shows the capability dependencies and logical groupings of capabilities (capability clusters) of the capabilities described in the StV-2. StV-4 is realized using UML structure or class diagrams.
StV-5	Strategic	Capability to Organization Deployment Mapping	This view details what capabilities are mapped to what systems at any particular time and the fulfilment of capability requirements, in particular by network-enabled capabilities. Use this view to perform gap/overlap analysis and interoperability analysis, validate that capabilities have been realized in Systems, and help provide system requirements documents (SRDs) for Systems views. StV-5 is realized using UML class or structure diagrams.

Architecture Viewpoint/View	View	View Name	Description
StV-6	Strategic	Operational Activity to Capability Mapping	This view describes the mapping between the capabilities required by an enterprise and the operational activities that those capabilities support. Use this view to map capabilities to Operations and ensure that all capabilities are fulfilled and can be traced to Operational Activities. StV-6 is derived from UML class diagrams with dependencies. It can be completed once you have done some of the Operational views. StV-6 is a major reason to use tables/matrix layouts and views in Rational Rhapsody.
Operational Viewpoint	Package		The Operational viewpoint contains views that provide a logical view of how an operation is carried out.
OV-1	Operational		OV-1 consists of three parts: OV-1a, OV-1b, and OV-1c. The OV-1 views are realized using UML use case diagrams.
OV-1a	Operational	High-Level Operational Concept Graphic	This high-level graphical presentation of the operational concept allows you to import pictures and other operational elements, such as Operational Nodes, Human Operational Nodes, Operational Activities, and the relations among them. It is intended to be an informal representation of the Concept of Operations (CONOPS).
OV-1b	Operational	Operational Concept Description	This view presents a textual description for OV-1a and it is produced with the associated OV-1a.
OV-1c	Operational	Operational Performance Attributes	This view presents in tabular form the details for the operational performance attributes associated with the scenarios represented in OV-1a and their evolution over time. OV1-1a contains performance parameters that define quality of service requirements.

Architecture Viewpoint/View	View	View Name	Description
OV-2	Operational	Operational Node Relationships Description	This view shows the detailed relationships and flows among operational nodes and operational activities. It also might be used to express a capability boundary, that is, the problem domain. OV-2 is realized using UML class or structure diagrams.
OV-3	Operational	Operational Information Exchange Matrix	This view, presented as a matrix, shows information exchanged between nodes, and the relevant attributes of that exchange. OV-3 can be derived from OV-2 and is generated using the table and matrix functionality.
OV-4	Operational	Organizational Relationships Chart	This view shows an organizational chart for the enterprise. It is divided into two types, a typical chart and an actual chart. OV-4 is realized for using UML class or structure diagrams.
OV-5	Operational	Operational Activity Model	This view shows the flow and ordering of activities required to achieve the operational capability. OV-5 is a lower-down version of OV-2 and is realized using UML activity diagrams.
OV-6a	Operational	Operational Rules Model	The OV-6 views are used to describe the textual rules that control and constrain the mission (for example, doctrine, rules of engagement, and so on). They are represented as operational constraints placed upon operational view model elements. The actual rules and to which elements they are applied are shown in a UML structure or class diagram and then shown in a table.
OV-6b	Operational	Operational State Transition Description	The OV-6 views are used to describe the mission objective. OV-6b view depicts the behavior of an operational element (node or activity). OV-6b is realized using UML statecharts.

Architecture Viewpoint/View	View	View Name	Description
OV-6c	Operational	Operational Event-Trace Description	The OV-6 views are used to describe the mission objective. OV-6c shows the messages and ordering of messages passing between operational nodes. OV-6c is realized using UML sequence diagrams.
OV-7	Operational	Information Model	This view shows the structures of the data that are being passed between elements. OV-7 is realized using UML class diagrams that define the data, its composition and types.
System Viewpoint	Package		The System viewpoint contains views that look at how Operations are physically implemented. They are used as input to SRDs, detail how platforms interact, and are the physical realization of capabilities in StVs.
SV-1	Systems	Resource Interaction Specification	This view shows what your resources are and how they interact with each other. This includes the human elements of your enterprise, such as roles, posts, and organizations; as well as physical elements, such as systems and platforms. This view is generally used for the definition of systems concepts/options, interface definitions, interoperability analysis, and operational planning. SV-1 is realized from using UML structure or class diagrams.
SV-2a	Systems	Systems Port Specification	The SV2 views are all related to communication and can be realized using UML structure and class diagrams. SV-2a specifies the ports (specified points of interaction) that a system has and defines the protocols that a port might use.
SV-2b	Systems	Systems Port Connectivity Description	The SV2 views are all related to communication and can be realized using UML structure and class diagrams. SV-2b shows the interaction of ports between systems. SV-2b is very similar to SV-1 but with protocols and networks included.

Architecture Viewpoint/View	View	View Name	Description
SV-2c	Systems	Systems Connectivity Clusters	The SV2 views are all related to communication and can be realized using UML structure and class diagrams. SV-2c defines how individual connections between systems are grouped into logical connections between parent resources.
SV-3	Systems	Resource Interaction Matrix	This view tells you what communicates with what. It is generated from the information from SV-1. SV-3 shows the lines of communication between systems as an N^2 diagram (system-system matrix). It indicates source (provider of information) and sink (consumer of information) of data flows.
SV-4	Systems	Functionality Description	This view defines the functional decomposition of a system or function. It can be used to show how functions interact to perform a higher-level function. SV-4 provides the functional requirements from the SRDs. SV-4 can be realized by using UML activity diagrams.
SV-5	Systems	Function to Operational Activity Traceability Matrix	This view is a spreadsheet-like generated view that summarizes the mapping of System and Systems functions to Operations, helps identify missing System functions, and provides traceability links between URDs and SRDs. This matrix is derived from UML class diagrams that have dependencies that link systems resources or functions to operations.
SV-6	Systems	Systems Data Exchange Matrix	Similar to SV-3 but for system data, this view details source-sink, protocols, content, and so on, of all data items. It helps specify interoperability requirements. SV-6 is derived from information captured as attributes in the model. It is customizable depending upon the information required by the user of the architecture.

Architecture Viewpoint/View	View	View Name	Description
SV-7	Systems	Resource Performance Parameters Matrix	This is a generated spreadsheet-like view that defines the quality of service requirements expected of each part of the system. SV-7 can be derived from a UML requirements diagram or attributes associated with model elements. This view is customizable by the user.
SV-8	Systems	Capability Configuration Management	This view is the system evolution description. It describes how the system or architecture is expected to evolve over long periods of time. SV-8 is similar to StV-3. SV-8 can be realized as an UML class or structure diagram.
SV-9	Systems	Technology and Skills Forecast	This view is a customizable table (it depends on user-defined time periods) that details what technology will be available in the near future. It touches upon system evolution, capability phasing, and acquisitions.
SV-10a	Systems	Resource Constraints Specification	This view specifies functional and non-functional constraints on the implementation aspects of the architecture (that is, the structural and behavioral elements of the SV viewpoint). These elements are mapped to each other on a class or structure diagram and shown in a set of table views.
SV-10b	Systems	Resource State Transition Description	This view shows state-based behavior, of the system resources to various events. For consistency, the state-based behavior should map to the aggregate behavior of all the flows shown in SV-10c. SV-10b is realized using UML statecharts.
SV-10C	Systems	Resource Event-Trace Description	This view provides a time-ordered representation of all the message and event interaction that occur between the various systems resources. These diagrams are focussed around specific scenarios. SV-10c is realized using UML sequence diagram.

Architecture Viewpoint/View	View	View Name	Description
SV-11	Systems	Physical Schema	This view is similar to OV-7 and it defines data used at the physical level (data relationships, structure, attributes), optimizes data structures, and specifies interfaces and data types. SV-11 is realized in UML class diagrams.
Acquisition Viewpoint	Package		It contains elements associated with the Acquisition viewpoint that are used by other views in the Rational Rhapsody MODAF profile. Note that the AcV-1 and AcV-2 views are not supported in the Rational Rhapsody MODAF profile.
Technical Viewpoint	Package		The Technical viewpoint contains views that detail technical standards that constrain the system development.
TV-1	Technical	Standards Profile	TV-1 presents the current technical and non-technical standards, guidance, and policy applicable to the architecture. Note that TV-1 can be a very large external document that is brought into the model or it can be created as a customizable table that maps elements representing standards to model elements.
TV-2	Technical	Standards Forecast	This view identifies the standards under development with expectations going forward. TV-2 can be seen in a customizable table that relates standards to time periods.

Configure a Rational Rhapsody project for MODAF

You specify the Rational Rhapsody model elements that are to form the core views in the generated MODAF documentation. From these, other MODAF views are derived. The Rational Rhapsody model elements included in the MODAF generated report are specified using stereotypes provided in the MODAF profile.

Creating a Rational Rhapsody for MODAF project

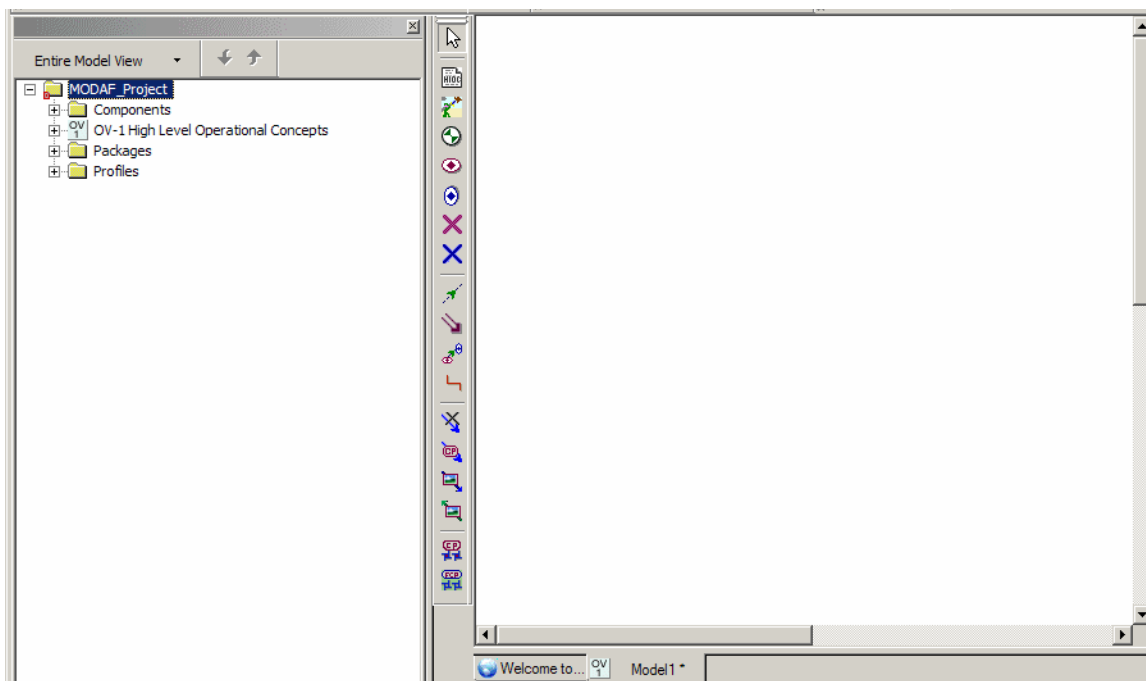
To create a Rational Rhapsody for MODAF project:

1. Launch Rational Rhapsody and choose **File > New** to open the New Project window.
2. On the New Project window:
 - ♦ Enter a project name (for example, MODAF_Project, as shown in the following figure).
 - ♦ Specify a folder location.
 - ♦ As the project type, select MODAF from the **Project Type** drop-down list. You might also want to select one of the **Project Settings**.

Note: The **MODAF** project type (or profile) is provided by the Rational Rhapsody for MODAF Add On in order to help you customize and extend the Rational Rhapsody product to support a Domain Specific Language (DSL), which lets you work with MODAF terms, diagrams, and artifacts rather than UML terms, diagrams, and artifacts.

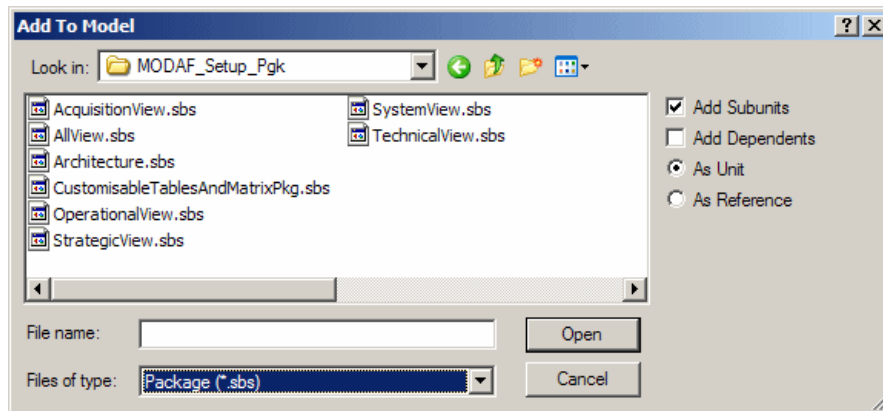
3. Click **OK**.

4. If the folder you specified does not exist, click **Yes** to create it. Rational Rhapsody creates the project and opens with the initial view, as shown in the following figure:

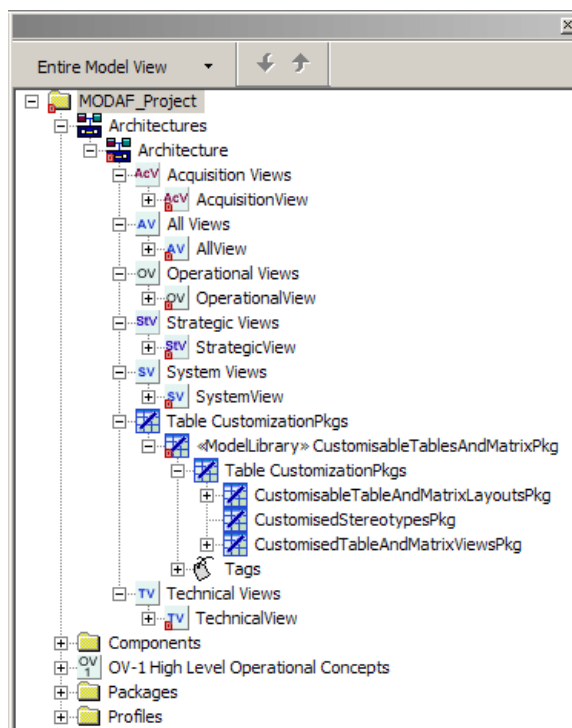


5. Add the primary Architecture Structure. Highlight the top-level project name (**MODAF_Project** in our example) and choose **File > Add to Model** to open the Add to Model window.

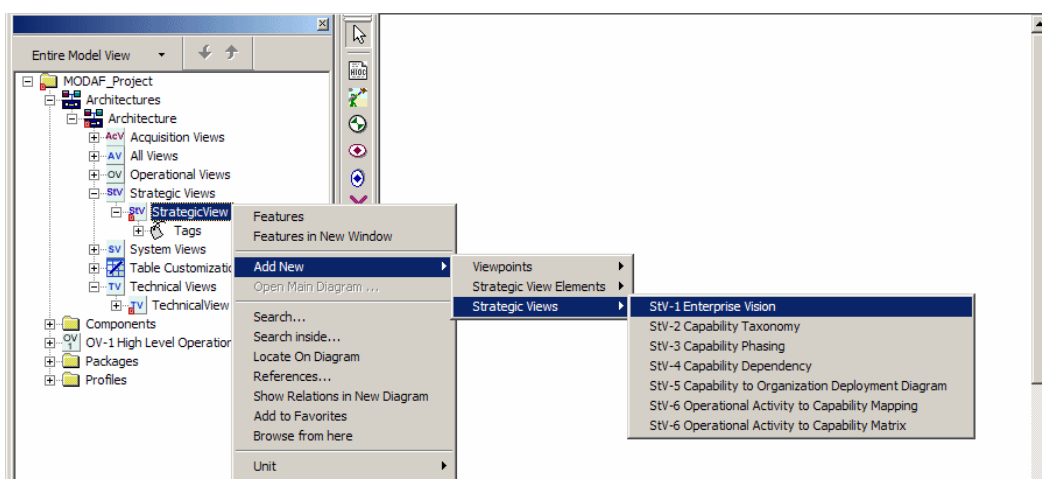
6. Browse to the Rational Rhapsody installation folder and locate **MODAF Pack**.
7. Accept the defaults and double-click **IBM Rational Rhapsody MODAF Add-on** and then locate **MODAF_Setup_Pkg**.
8. Accept the defaults and double-click **MODAF_Setup_Pkg** and then change the files of type to **Package (*.sbs)**, as shown in the following figure:



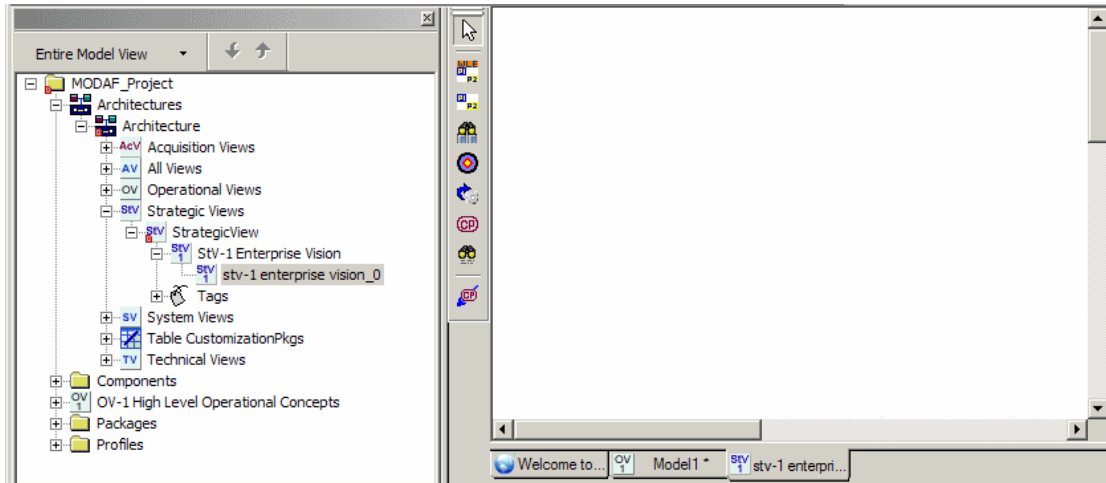
9. Accept the defaults and double-click **Architecture.sbs**. This brings in the main viewpoints for your MODAF project. You now have the basic project structure. Your expanded Rational Rhapsody browser should resemble something like the following figure:



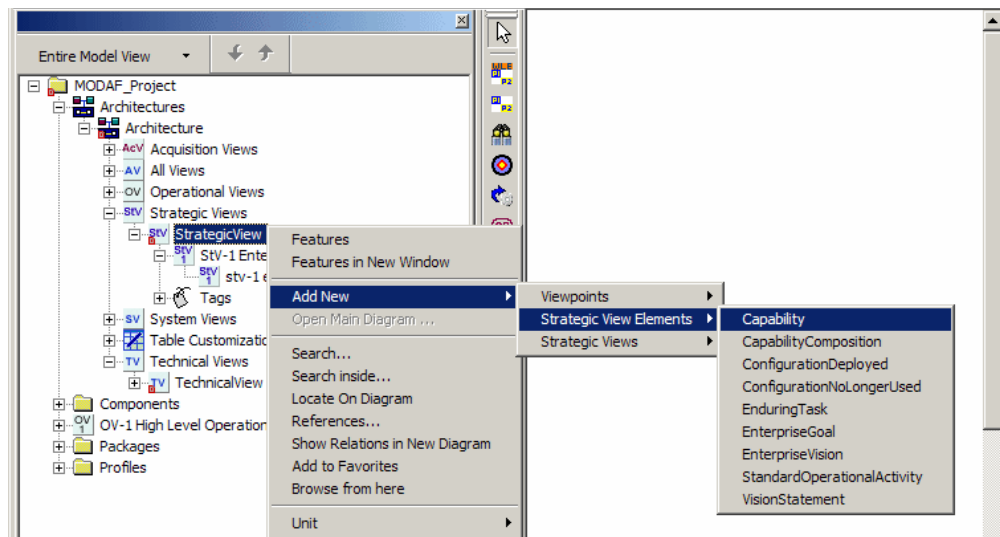
10. To add views to a particular viewpoint, right-click the viewpoint and select **Add New > [Name] Views > [Name of View]**.



11. Rational Rhapsody opens the drawing area with the applicable **Diagram Tools** for the view when appropriate, as shown in the following figure:



12. To add other elements for a particular viewpoint, right-click the view and select **Add New > [Name] View Elements > [Name of Element]**, as shown in the following figure:



Rational Rhapsody provides these additional methods to view model data:

- The screenshot shows the 'Entire Model View' on the left, displaying a hierarchical tree of the model. The tree structure is as follows:

 - Architectures
 - EmailingArchitecture
 - Acquisition Views
 - All Views
 - Operational Views
 - Strategic Views
 - System Views
 - Table CustomisationPkgs
 - «ModelLibrary» CustomisableTablesAndMatrixPkg
 - Table CustomisationPkgs
 - CustomisableTableAndMatrixLayoutsPkg
 - CustomisedStereotypesPkg
 - CustomisableTableAndMatrixViewsPkg
 - Matrix Views
 - AFS_OwningBodies
 - Table Views
 - CustomisedTV1and2ForArchitectural
 - table customisationpkg_30013

On the right, a table lists the service areas:

ServiceArea	Name	ElementType	Description
Military AFs	C4ISR	Standard	First Architectural Framework applied to military applications
Military AFs	DoDAF1	Standard	Successor to C4ISR provided more formality but complications with view definitions
Military AFs	DoDAF15	Standard	Successor to DoDAF1 added Service views (mainly SoA for netcentric operations) reinforced the notion of capability which was underused in DoDAF 1 Stopgap before bringing out DoDAF 2
Military AFs	DoDAF20	Standard	Intended successor to DoDAF 1/1.5 much greater formality. Intended to follow the MODAF views and take into greater capability and procurement
Military AFs	MODAF10	Standard	Based upon DoDAF 1 but expanded to cover the MODs needs for Capability and procurement analysis, introduced Strategic (capability analysis) and Acquisition (Procurement/project management) views.
Military AFs	MODAF11	Standard	Update to MODAF 1, a lot of bug fixes around the M3 metamodel Also gave example views for services which were used by NAF 3
Military AFs	MODAF12	Standard	Bug fixes in MODAF 1.1 to make it more consistent and tidy Brings in Service views properly, These are the same as NAF 3 service views
Military AFs	NAF3	Standard	Based upon MODAF 1.1 with the Service views Acquisition View renamed to Programme View Strategic View renamed to Capability View It uses the same metamodel as MODAF 1.1 with some minor additional views.

-
- The screenshot displays the UML modeling environment with the Package Explorer on the left and the Package Browser on the right.
- Package Explorer (Left):**
- MODAF_CooperatingForcesEmail
 - Architectures
 - EmailingArchitecture
 - AcqV Acquisition Views
 - All Views
 - Operational Views
 - Strategic Views
 - System Views
 - Table CustomizationPkgs
 - <ModelLibrary> CustomisableTablesAndMatrixPkg
 - Table CustomizationPkgs
 - CustomisableTableAndMatrixLayoutsPkg
 - CustomisedStereotypesPkg
 - CustomisedTableAndMatrixViewsPkg
 - Matrix Views
 - AFs_OwningBodies
 - Table Views
 - CustomisedTV1and2ForAchitecturalFramework
 - Technical Views
 - Components

Package Browser (Right):

The Package Browser shows a table of packages and their associated standards.

To: ActualOrganization	Scope: ArchitecturalFrameworkStandards
NATO	US_DepartmentOfDefence
UK_MinistryOfDefence	US_DepartmentOfDefence
US_DepartmentOfDefence	US_DepartmentOfDefence
US_DepartmentOfDefence	US_DepartmentOfDefence
US_DepartmentOfDefence	US_DepartmentOfDefence
MODAF10	UK_MinistryOfDefence
MODAF11	UK_MinistryOfDefence
MODAF12	UK_MinistryOfDefence
NAF3	NATO

These views provide the following development capabilities:

- ◆ Define and run dynamic queries of model content
- ◆ Easy display and analysis of requirements
- ◆ Exportable and printable tables and data lists

For details on how to create the table and matrix views in Rational Rhapsody, see [Table and matrix views of data](#). To be able to supply the data for your table and matrix views, you have to create stereotypes and tags. See [Creating stereotypes and using tags](#).

Creating stereotypes and using tags

A profile hosts domain-specific tags and stereotypes. In addition, you can create your own stereotypes and tags for your Rational Rhapsody project. For more information about stereotypes, see [Stereotypes](#). For tags, see [Use tags to add element information](#).

To be able to create your own custom table and matrix views, you can use the stereotypes and tags provided by the MODAF profile. However, to be able to truly produce table and matrix views of your own design, you will probably find it most useful to create at least one stereotype of your own and then create the tags you want for it. Once set up, you can associate any of your tags with views and operations to produce tables and matrices that are customized for your project needs.

Note

Before you try to create your own custom table and matrix views, see [Table and matrix views of data](#). You should also see [About creating table/matrix views in MODAF](#).

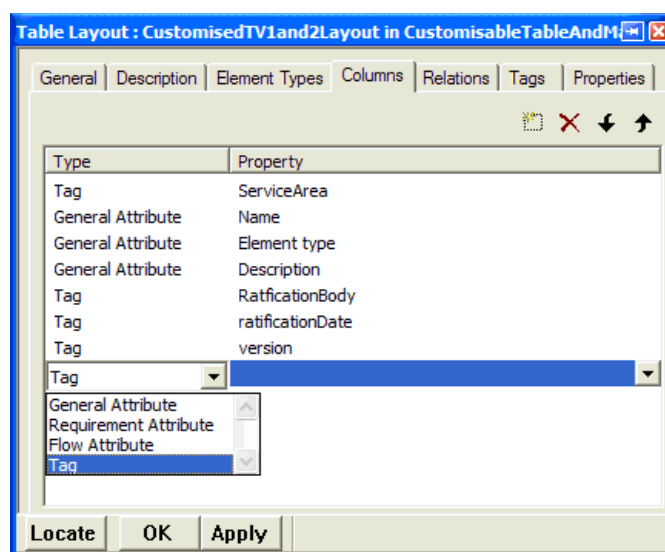
About creating table/matrix views in MODAF

For the Rational Rhapsody for MODAF Add On, the ability to create custom table and matrix views is facilitated by the Table CustomizationPkg, which consists of three subpackages:

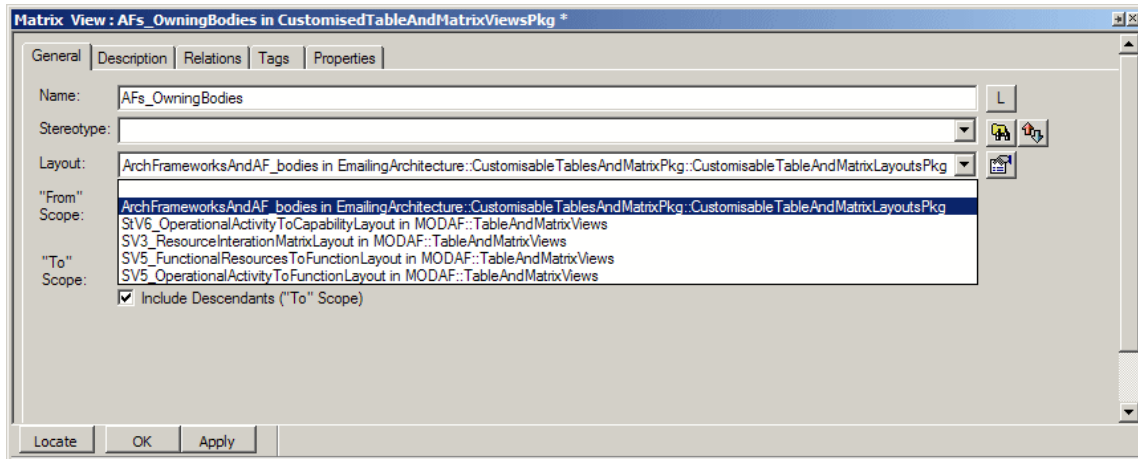
- ♦ [CustomizableTableAndMatrixLayoutsPkg](#)
- ♦ [CustomizedStereotypesPkg](#)
- ♦ [CustomizableTableAndMatrixViewsPkg](#)

CustomizableTableAndMatrixLayoutsPkg

The CustomizableTableAndMatrixLayoutsPkg package contains the table layouts for the MODAF-specific table views that need to be customized by the user. The customization normally consists of setting up the display of the tag elements to be extracted by the user, as shown in the following figure:

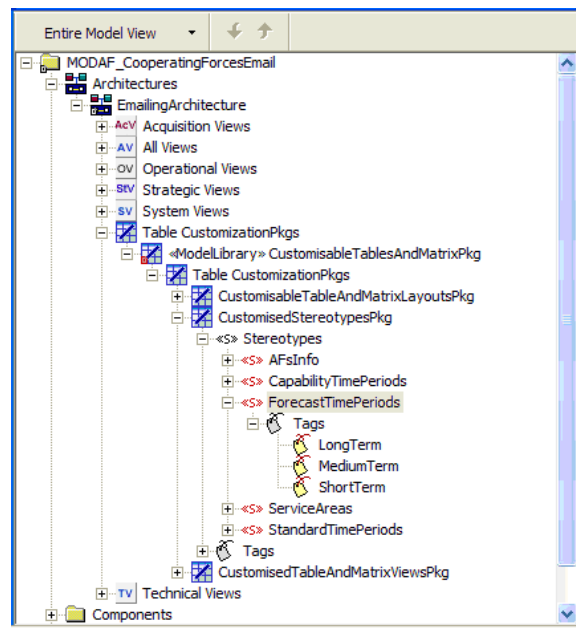


Customized matrix layouts can be added and used by the specific MODAF views by changing the layout and scope to be used by the view, as shown in the following figure:

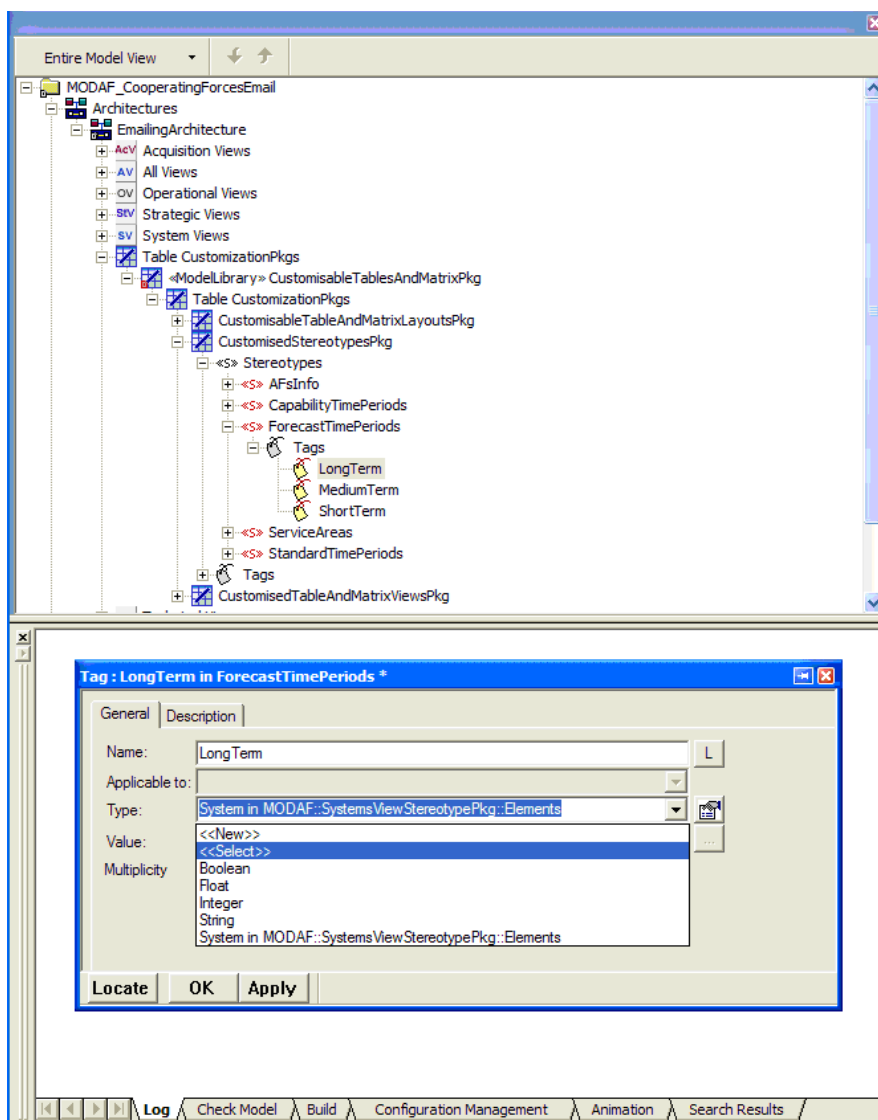


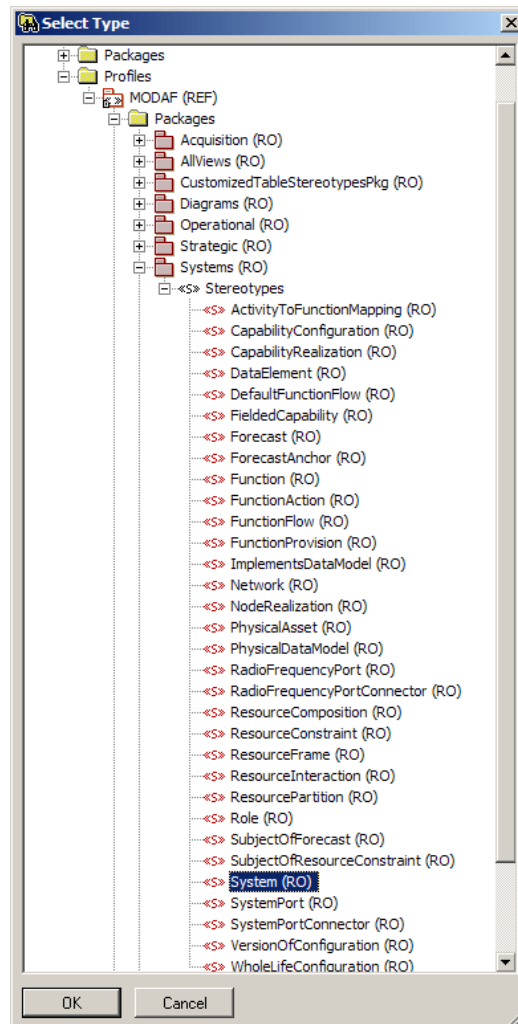
CustomizedStereotypesPkg

The CustomizedStereotypesPkg package provides a location to store the custom stereotypes to be used for custom table layout. Stereotypes can be created with user customizable tags and then applied to specific type of element (typically a class or object).



The tags are typically typed by a specific element (selected from the MODAF profile stereotypes list) type. This ensures that when the tag is populated in the model element that the correct type of information is displayed in the actual table (if defined in the table layout).



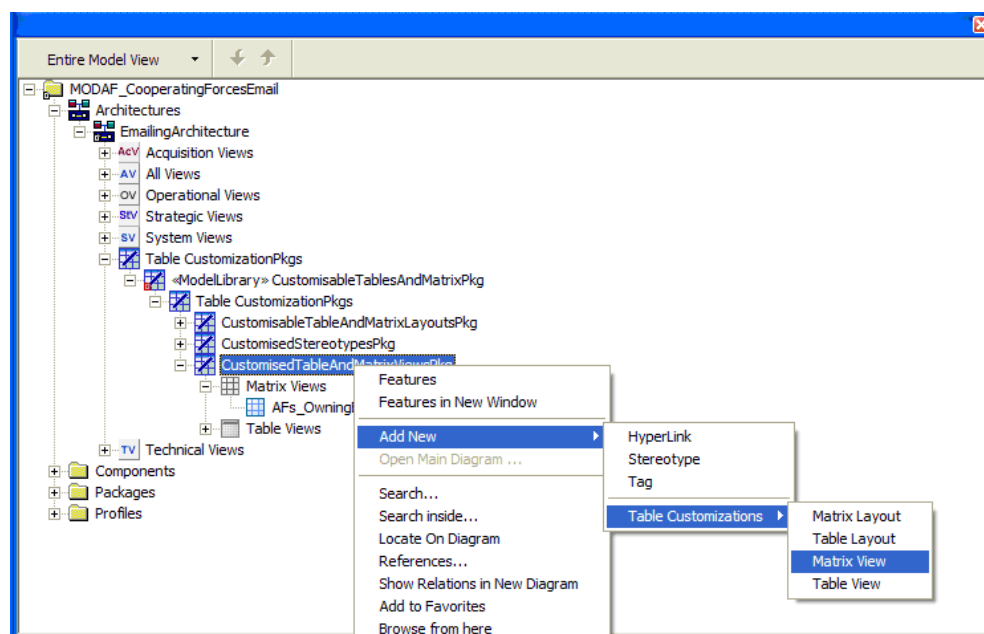


When an element is given a particular stereotype all the tags associated with the stereotype are inherited by the element. Then you have to populate the tag values with the correct information.

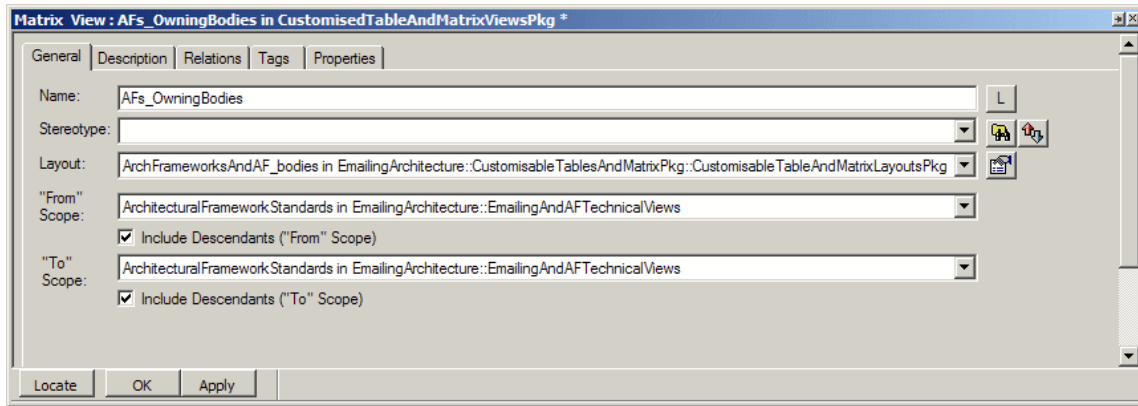
CustomizableTableAndMatrixViewsPkg

The customized MODAF-specific views are still created in their correct places in the Architecture. The CustomizableTableAndMatrixViewsPkg package is intended for custom views outside the scope of the MODAF-specific table and matrix views.

You create a customizable table or matrix by selecting the appropriate package, for example, **Add New > Table Customizations > Matrix View** (as shown in the following figure) or **Add New > Table Customizations > Table View**.



Rational Rhapsody creates a new view. Then you have to open the Features window for the view and select the package scope of the view and the appropriate layout to use (from the CustomizedTableAndLayoutsPkg package), as shown in the following figure:



When the table or matrix view is selected in the browser, it will be brought up in the Graphic Editor area (typically to the right of the browser).

Note: If the view displays empty, incomplete, or has too much information, you might want to review how the scope was set.

For more details about creating the Table and Matrix views in Rational Rhapsody, see [Table and matrix views of data](#).

Note

For the MODAF profile, to create a layout or view for a table or matrix, the menu command path, for example, is **Add New > Table Customization > Table Layout** (instead of **Add New > Table Layout**). Otherwise, the procedures in [Table and matrix views of data](#) are the same.

Create documentation for Your MODAF project with ReporterPLUS

The Rational Rhapsody for MODAF Add On provides you with a ReporterPLUS template that gives you the ability to generate complete electronic documentation with hyperlinks between related and referenced model elements. Output, for example, can be in HTML and Microsoft Word. For information about ReporterPLUS.

This section describes how to set up ReporterPLUS to use this template. It also discusses the structure of the generated document, how to generate the document, and provides some tips on how to use the model structure to its best advantage when generating a document.

Setting up ReporterPLUS

As mentioned earlier, the ReporterPLUS template supplied with the Rational Rhapsody for MODAF Add On can produce hyperlinks between referenced elements. To enable this feature of the template, you must modify the `Rhapsody.ini` file located in the Rational Rhapsody installation folder and then move it to the System (or Windows directory).

To set up ReporterPLUS:

1. Make a copy of the `Rational Rhapsody.ini` file that is located in the Rational Rhapsody installation folder (for example, `<Rational Rhapsody installation path>\Rhapsody730`).

You should now have a `Rhapsody.ini` file and a Copy of `rhapsody.ini` file.

2. Rename Copy of `rhapsody.ini` file to `OriginalRhapsody.ini`.

Having this file ensures that you can return to the original state of the `rhapsody.ini` file when needed.

3. Add the following lines within the [General] section of the Rhapsody.ini file:

```
[ReporterPLUS]
EnableLoadOptions=TRUE
LoadElementReferences=TRUE
```

4. After you save your changes to the Rhapsody.ini file, move it to your System (or Windows directory).

Note: If you use Copy and Paste, be sure to delete the file from where you copied it from.

5. Restart Rational Rhapsody to ensure the changes take effect.

Document structure

The document produced by ReporterPLUS consists of these main sections:

- ♦ A table of contents where all the views and elements are listed and hyperlinked to their location in the documentation.
- ♦ A graphical section that contains all the views, including the embedding of external documents, elements that exist on those views and their descriptions, if they have any. All the elements are hyperlinked to their definition in the Data Dictionary section.
- ♦ A Data Dictionary section that contains the details of all the elements in the model in the viewpoint in which they were created. The model elements from the first section are all hyperlinked to their counterpart in the Data Dictionary as are any connected elements, descriptions and tags for all elements are shown.

Generating a MODAF document

The Rational Rhapsody for MODAF Add On provides you with a ReporterPLUS template called `ModafReport.tpl`. The template is located within the Rational Rhapsody installation path (for example, `<Rational Rhapsody installation path>\MODAF Pack\templates`). You can use ReporterPLUS to create, for example, HTML and Microsoft Word documents from any Rational Rhapsody model.

To use the `ModafReport.tpl` template:

1. Be sure that you have set up ReporterPLUS to use the `ModafReport.tpl` template; see [Setting up ReporterPLUS](#).
2. With your MODAF project open in Rational Rhapsody, choose **Tools > ReporterPLUS > Report on all model elements**.
3. On the Select Task window, select the type of output you want, and click **Next**.
4. On the Select Template window, browse to the MODAF ReporterPLUS template location (for example, `<Rational Rhapsody installation path>\MODAF Pack\templates`) and select the `ModafReport.tpl` file.
5. On the Select Template window, click **Next**.
6. On the Confirmation window, click the **Finish** button.
7. On the Generate Document window:
 - ♦ Enter a document name.
 - ♦ Browse to where you want to locate the files that will be produced.
 - ♦ Click the **Generate** button to generate your document.
8. Wait while your document is generated. ReporterPLUS spends some time loading the template and the model. Then it analyzes the model and the model element relationships.
9. When available, click **Yes** to open your report.

Troubleshooting ReporterPLUS and Rational Rhapsody for MODAF

If the contents of a package do not appear in the document created by ReporterPLUS, it is possible that you might have created an architecture without adding the architecture.sbs package as described in [Configure a Rational Rhapsody project for MODAF](#).

The MODAF profile has two tags that have been added to all the viewpoints, ContainsViews and RootView. By default, they are set to Cleared (meaning False, their check boxes are not selected). However, within the imported architecture.sbs package, these tags have been preset to Checked (meaning True, their check boxes are selected).

ContainsViews is set to Checked when you have established a hierarchy of views within a particular viewpoint that specifically contains views. By default, this is set to Cleared because the main reason why subviews are used is to provide containers for sets of model elements so that a table or matrix view can be scoped correctly.

RootView indicates that this is the Root Viewpoint for this particular set of views, it should only be set to Checked for the viewpoints that sit directly above the Architecture layer.

To override the default settings for a particular viewpoint:

1. Right-click the viewpoint and select **Features**.
2. Select the **Tags** tab on the Features window.
3. Select the **ContainsView** check box (so that it is selected).
4. Select the **RootView** check box if the viewpoint is likely to contain views that you want displayed in the documentation. If a viewpoint just contains elements and if those elements are used by other views external to the owning viewpoint, they will still appear in the document generated by ReporterPLUS in the correct places.

The Dependencies Linker

The Dependencies Linker is an extension to the Link wizard documented in the Systems Engineering Toolkit (for more information, see [Harmony process and toolkit](#)). The Dependencies Linker allows dependencies to be created between particular model elements. It is located under the **MODAF** tab of the Link Wizard window.

The Dependencies Linker is a tool for advanced users who know which dependency relationships are allowed between which elements and can use the browser effectively.

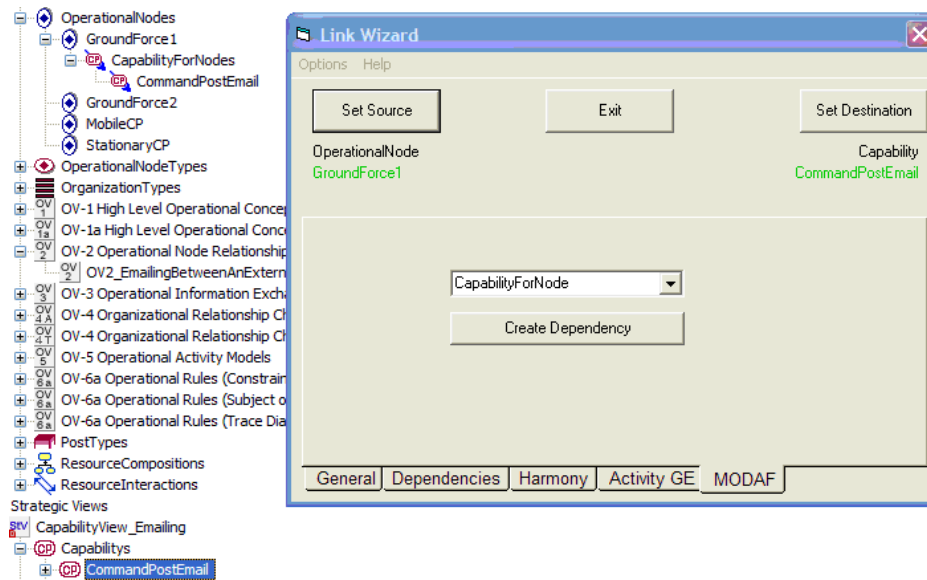
The `modaf_stereotypes.txt` file contains a list of the possible dependencies. The file is located in the <Rational Rhapsody installation path>\Share\Profiles\MODAF path. The profile controls the permissions as to what dependencies can be created by what elements.

Using the Dependencies Linker

To use the Dependencies Linker:

1. Choose **Tools > Dependencies Linker** to open the Link Wizard window.
2. On the Rational Rhapsody browser, select the source element and then click **Set Source** on the Link Wizard window.
3. On the **MODAF** tab of the Link Wizard, select the dependency from the drop-down list.
4. On the Rational Rhapsody browser, select the destination element and then click **Set Destination** on the Link Wizard window.

- Click the **Create Dependency** button on the **MODAF** tab of the Link Wizard.



Note: If the **Create Dependency** button is disabled, this means that you have selected an illegal dependency between the two elements.

Troubleshoot the Dependencies Linker

If the Dependencies Linker tool does not appear, then it is likely that the MODAF .hep file is not located in the same folder as the MODAF .sbs file. Make sure the MODAF .hep file is in the same folder as the MODAF .sbs file.

General troubleshooting

This section provides you with information that you can use for general troubleshooting purposes for Rational Rhapsody for MODAF Add On.

Verify the Rational Rhapsody for MODAF Add On installation

If you are having a problem with the Rational Rhapsody for MODAF Add On or if it does not appear in the Rational Rhapsody product as mentioned in this section, you should verify that it has been added.

Use any of the following methods to verify that the Rational Rhapsody for MODAF Add On has been added:

- ♦ See if there is a path to the Add On. From the Windows Start menu, select **All Programs > IBM Rational > IBM Rational Rhapsody *version number* > IBM Rational Rhapsody MODAF Add On**.
- ♦ See if the Add On has been included in the Rational Rhapsody product. Typically that path would be <Rational Rhapsody installation path>\MODAF Pack.
- ♦ Create a project in Rational Rhapsody and see if the **MODAF** type (profile) is available.

If you find that the Rational Rhapsody for MODAF Add On is not on your system, then you must install it. You need a license key for the Rational Rhapsody for MODAF Add On if it is not already a part of your Rational Rhapsody license. See the Rational Rhapsody installation instructions for more information about installing the Rational Rhapsody for MODAF Add On.

If the Rational Rhapsody for MODAF Add On is on your system but you are having problems with it, you might have to remove it and then run the Installation program again to repair your software if it has been damaged.

Find icons missing from diagram tools

If you notice that there seems to be icons missing from **Diagram Tools**, check to be certain that the \$OMROOT is correct for the Rational Rhapsody for MODAF Add On.

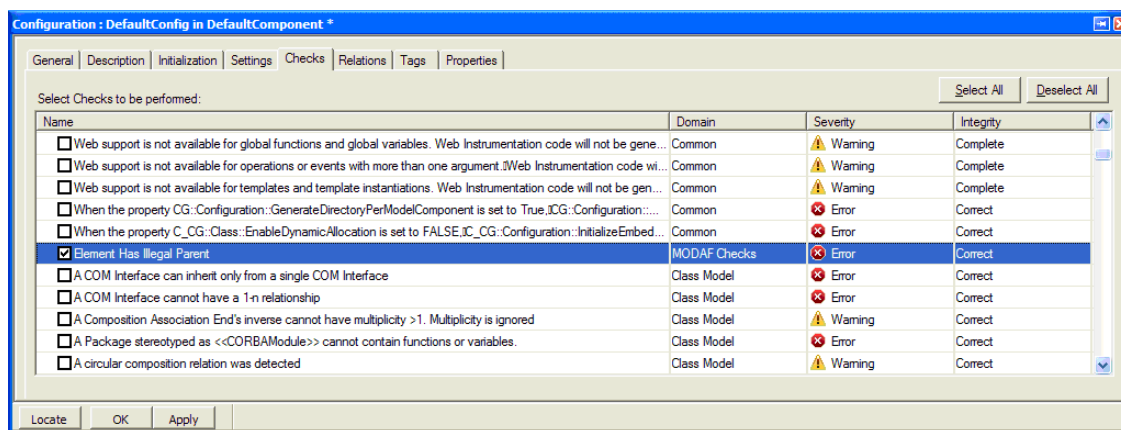
Check your Rational Rhapsody for MODAF model

The Rational Rhapsody for MODAF Add On contains a helper (a Java plug-in) that checks the architectural conformance of your model. It indicates where an element is contained within an illegal parent.

Setting up the Rational Rhapsody Check Model tool for a Rational Rhapsody for MODAF Add On project

To set up the Rational Rhapsody Check Model tool so that it checks your Rational Rhapsody for MODAF Add On model:

1. Choose **Tools > Check Model > Configure** to open the Configuration window for the Check Model tool.
2. Click the **Deselect All** button to clear the check boxes for all the checks (tests).
3. Select the **Element Has Illegal Parent** test (which is in the MODAF Checks domain), as shown in the following figure:

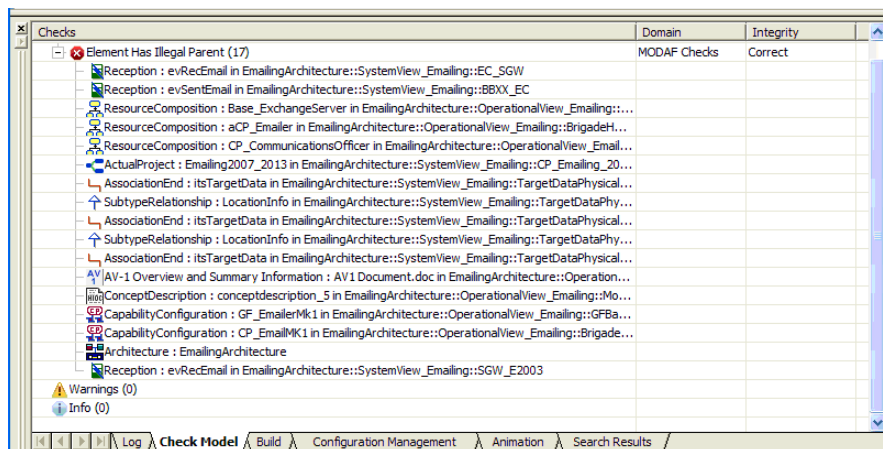


4. Click **OK**.

Running the Rational Rhapsody Check Model tool

To run the Rational Rhapsody Check Model tool:

1. Choose **Tools > Check Model > DefaultConfig** (the name of the default component).
2. The results appear on the **Check Model** tab of the Rational Rhapsody Output window, as shown in the following figure:



Note the following information about the Rational Rhapsody Check Model tool for a Rational Rhapsody for MODAF Add On model:

- ♦ If more than one error is found, the Architecture will always be seen by default. Other results that can be ignored are if Receptions are used on Interfaces and AssociationEnds, and Subtype relationships for data elements.
- ♦ When possible, you can double-click the element highlighted in an error to locate it in the browser to help you understand its parent hierarchy with reference to the M3.
- ♦ You might want to examine the Model::Stererotype::Aggregates property of the stereotype definition in the profile. This contains the elements that should be allowed to be contained by the parent element causing the error.
- ♦ You can examine the structure of an element. Right-click the error element or its parent in the browser and select **Show Relations in New Diagram** to create an object model diagram that is automatically populated with related model elements. For more information on **Show Relations in New Diagram**, see [Showing all relations for a class, object, or package in a diagram](#).
- ♦ The Rational Rhapsody Check Model tool only works with JRE 1.5.

The Rational Rhapsody automotive industry tools

The Rational Rhapsody tools for the automotive industry provide for specification and design of automotive systems and software applications. Rational Rhapsody provides the AUTOSAR-specific [Model-driven Development \(MDD\)](#) environment to leverage both UML and SysML. This makes it possible for engineers to reuse specifications for common vehicle features across multiple automotive lines. The AUTOSAR profiles are used for architectural description of an AUTOSAR model using the native AUTOSAR concepts.

The AutomotiveC profile takes advantage of the capabilities of the MicroC profile, which enables code generation for static systems with limited resources. The AutomotiveC profile also contains a number of features designed specifically for automotive projects.

Rational Rhapsody extends the benefits of MDD to the C developer by allowing designers to work in either a functional or object-oriented environment. Rational Rhapsody includes blocks, flows, graphical files, functions, and data so that C developers can create models using familiar concepts.

AUTOSAR modeling

Rational Rhapsody provides two AUTOSAR profiles (AUTOSAR_21, and AUTOSAR_31) that can be used for modeling components in accordance with the AUTOSAR development process. These profiles conform with the AUTOSAR 2.1, and 3.1 standards.

Note

The AUTOSAR profiles are only visible in the list of profiles if you selected the Automotive option during your installation of Rational Rhapsody.

Information on the AUTOSAR architecture can be found at the AUTOSAR Web site, www.autosar.org

The AUTOSAR workflow

The workflow for AUTOSAR modeling is as follows:

1. Create an AUTOSAR Rational Rhapsody project.
2. Create diagrams using the AUTOSAR elements provided by Rational Rhapsody.
3. Use the Rational Rhapsody Check Model tool to verify that there are no problems with your AUTOSAR model.
4. Export your model to the AUTOSAR XML format.

Creating an AUTOSAR project

To create an AUTOSAR project in Rational Rhapsody:

1. Choose **File > New**.
2. In the New Project window:
 - a. Provide a project name.
 - b. Indicate the folder where the project should be saved.
 - c. From the **Project Type** list, select **AUTOSAR_20**, **AUTOSAR_21**, or **AUTOSAR_31**.
 - d. You might select one of the **Project Settings**.
 - e. Click **OK**.

Creating AUTOSAR diagrams

The Rational Rhapsody AUTOSAR profiles allow you to compose the following types of diagrams:

- ♦ ECU diagram
- ♦ Implementation diagram
- ♦ Internal Behavior diagram
- ♦ SW Component diagram
- ♦ System diagram
- ♦ Topology diagram

Checking an AUTOSAR model

To verify that there are no problems with your model, select **Tools > Check Model > [configuration name]** from the main menu.

Import/export from/to AUTOSAR XML format

Projects based on the AUTOSAR profiles can be exported from Rational Rhapsody to the AUTOSAR XML format.

To export your project:

1. Select **Tools > Generate AUTOSAR XML Document**.
2. Click **Browse** on the When the Export window to select the destination XML file.
3. Click **Export**.

Rational Rhapsody can also import data stored in the AUTOSAR XML format.

To import AUTOSAR data:

1. Select **Tools > Import AUTOSAR XML Document**
2. Click **Browse** on the When the Import window opens to select the source XML file.
3. Click **Import**.

The AutomotiveC profile

The AutomotiveC profile provides the capabilities that you are likely to require for projects in the automotive industry and developed using the C language only.

The AutomotiveC profile automatically loads the MicroC profile, which contains code-generation capabilities designed for static systems.

In addition, the AutomotiveC profile includes a number of features intended specifically for automotive projects, such as:

- ♦ Automotive-specific adaptors, based on the user of configuration-level stereotypes
- ♦ Simulink and StateMate block integration capabilities

Automotive-specific adaptor

The AutomotiveC profile includes an automotive-specific adaptor called OSEK21.

You select the adaptor to use by applying one of the configuration-level stereotypes (see [Configuration stereotypes](#)).

The OSEK21 adaptor

The OSEK21 adaptor provides a Rational Rhapsody development environment for users of the Metrowerks OSEK 21 operating system and tools.

The OSEK adaptor is a set of Rational Rhapsody components which allow you to easily generate code and build applications for either of the following targets:

- ♦ Metrowerks OSEK21 OS/NT Build 2.1.10 (Build env: MSdev)
- ♦ Metrowerks OSEK21 OS/12 Build 2.1.13 (Build env: Metrowerks)

All of the required files are copied to your disk when you choose the *Automotive* add-on option during installation.

Target hardware and software

The OSEK21 adaptor is used to create applications for the following target hardware and software:

Hardware:

- ♦ OSEK21NT target: PC
- ♦ OSEK21HC12 target: Motorola HC12dg128 evaluation board.

Software:

- ♦ OSEK21NT target: Windows XP, Metrowerks OSEK 21 OS for NT, Visual Studio (nmake utility, compiler, linker and so on)
- ♦ OSEK21HC12 target: Windows XP, Metrowerks OSEK 21 OS for HC12, Hiware tools for the Motorola HC12 target (compiler, linker, and so on), Visual Studio (nmake utility)

OSEK21 tasks

The predefined task OS_TASK is responsible for initialization of the model. Using the relevant properties provided, the following characteristics have been assigned to the task: highest priority, non-preemptive, autostart, basic. You can change these characteristics as required.

The predefined task TIMER_TASK is responsible for timeout-handling and dispatching of timed actions. Using the relevant properties provided, the following characteristics have been assigned to the task: second-highest priority, non-preemptive, autostart, basic. You can change these characteristics as required.

Using the OSEK21 adaptor - workflow

To build an application using the OSEK21 Adaptor:

1. Create a new project of type AutomotiveC.
2. Add OSEK tasks to your model: **Add New > AutomotiveC > OSEK21BasicTask** (or **OSEK21ExtendedTask**).
3. For each task, change the **Concurrency** box to **Active**.
4. For each task, use the **Tags** tab of the Features window to set the necessary values for the OIL definition.
5. For each task, add the required attributes and operations.
6. Create a new OSEK21HC12Configuration configuration (**Add New > OSEK21 > OSEK21HC12Configuration**).
7. Set the new configuration to be the active configuration

8. In the configuration, set the property `C.CG:OSEK21HC12:OSEKDIR` to the path to OSEK 21 for HC12.
9. In the configuration, set the property `C.CG:OSEK21HC12: HICROSSDIR` to the path to the Hiware tools.
10. If necessary, modify the value of the following properties:

`C.CG::OSEK21HC12::OsekMainFileDefinition` - determines the content of the C source file that contains the main entry of the OSEK application and the definition of the predefined tasks from the framework (`OS_TASK` and `TIMER_TASK`).

`C.CG::OSEK21HC12::OilDefinitionTemplate` - the content of the OIL file `cfg.oil` that contains include statements to include the model's specific OIL definition.
11. Generate and compile the application.

The workflow described above refers to building an application for the OSEK21 OS/12 target.

To build an application for the OSEK21 OS/NT target, use `OSEK21NTConfiguration` when creating a new configuration, and in the configuration, set the property `C.CG::OSEK21NT::OSEKDIR` to the path to OSEK 21 for NT.

Automotive-specific stereotypes

The stereotypes are available for the C language and in the Automotive C profile only.

Configuration stereotypes

The AutomotiveC profile contains the following stereotypes that can be applied to configurations:

- ◆ `OSEK21NTConfiguration`
- ◆ `OSEK21HC12Configuration`

These stereotypes are “new terms” that are applicable to configurations. They correspond to the following environments:

- ◆ `OSEK21NT`
- ◆ `OSEK21HC12`

These stereotypes set the value of the Environment property. This value specifies the set of Rational Rhapsody properties to use for that environment.

You can create a new configuration using either of the following methods:

- ◆ In the browser, from the context menu for components, select **Add New > AutomotiveC > [configuration]**.
- ◆ From the context menu for components, select **Add New > Configuration**. From the context menu for the newly created configuration, select **Change To > [configuration]**.

Note

For AutomotiveC projects, you must use one of the relevant configuration stereotypes. When you create a new project, the browser will contain a configuration called `DefaultConfig`. This is a generic configuration so you must apply one of the automotive configuration stereotypes to it.

Simulink and StateMateBlock integration capabilities

When you create a new project based on the *AutomotiveC* profile, Rational Rhapsody also loads the *SimulinkInC* profile and the *StateMateBlock* profile.

This allows you to include Simulink and StateMate blocks in your model.

Fixed-point variable support

The AutomotiveC profile provides support for fixed-point variables by automatically loading the Rational Rhapsody *FixedPoint* profile:

For more information regarding the use of fixed-point variables in Rational Rhapsody, see [Using fixed-point variables](#).

AutomotiveC properties

The *AutomotiveC.prp* file is loaded when you create a project based on the AutomotiveC profile.

The file contains properties for the OSEK environments.

StatemateBlock in Rational Rhapsody

The Rational Statemate system, an IBM product, is a high-level graphical development environment. Many companies use Rational Statemate for their modeling needs. However, some Rational Statemate customers requested the flexibility to provide co-execution of models in Rational Statemate and Rational Rhapsody. The integration of these two modeling products allows the seamless code-to-code merging of a Rational Statemate model into a Rational Rhapsody architecture. This integration has the following prerequisites:

- ♦ Rational Rhapsody Developer for C version 7.1.1 or higher
- ♦ Rational Statemate 4.2 MR2 or higher on the computer and licensed
- ♦ License for Rational Statemate MicroC code generator

Preparing a Rational StatemateBlock for Rational Rhapsody

To synchronize a Rational Statemate model with Rational Rhapsody, the Rational Statemate model must have the following characteristics:

- ♦ Only one top-level activity
- ♦ MicroC profile with only one module

Perform the following operations in Rational Statemate to prepare the Rational Statemate model for Rational Rhapsody:

1. Open the Rational Statemate model. If you want to show Rational Statemate animation in Rational Rhapsody, be certain to select the GBA option from the Rational Statemate MicroC profile options.
2. To set the required properties in Rational Statemate before generating code:
 - a. Open the Rational Statemate MicroC Code Generator.
 - b. Select **Options > Settings > Application Configuration > Application Files**.
 - c. In the Application Files window, select both of the **Generate Code in a Single File** and **Generate Code as Statemate Block** items.

- d. Click **OK**.
3. Generate the C code using the Rational StateMate MicroC Code Generator.
4. In the Rational StateMate main interface, select the **Files** tab and:
 - a. Select a Rational StateMate MicroC code generator profile.
 - b. Select the menu item: **Configuration > Create StateMate Block Configuration for Rhapsody > Read mode/Update mode**.

Creating the Rational StateMateBlock in Rational Rhapsody

In order to create a Rational Rhapsody element for the Rational StateMate model:

1. Open Rational Rhapsody.
2. Choose **File > New** to create a new Rational Rhapsody project. Fill in the **Project name** and **In folder** boxes.
3. In the **Type** box of the New Project window, select the **StateMateBlock** profile from the pull-down menu.
4. Rename the automatically created diagram to relate to your project.
5. In the diagram, use the Object  icon in the **Diagram Tools** to create an object with a name that is appropriate for your project.
6. Right-click the object in the diagram and select **Features** from the menu.
7. In the **General** tab of the Features window, select the **StateMateBlock** class stereotype for the object and click **OK** to save the change.
8. Right-click the StateMateBlock object and select **Import/Sync StateMate Model**.
9. In the Import/Sync State Block window, fill in the fields in this order:
 - a. The default in the Rational StateMate Installation path should be the path to your Rational StateMate installation in this format <STM_ROOT>\pm (pm = project management). If the location of the Rational StateMate pm file is not correctly displayed in the default location, click the **Advanced** button. Then enter the correct path to your Rational StateMate installation in the window. Click **OK** to save the path information and return to the Import/Sync State Block window.

Note: This is an important step because the path entered here resets all Rational Rhapsody path references to your Rational StateMate system to use this newly entered path as the default Rational StateMate path. See [Troubleshooting](#)

[Rational StateMate with Rational Rhapsody](#) if you receive error messages while performing this operation.

- b. Select **StateMate Project** from the pull-down list in the next field.
- c. Select **StateMate Block** from the pull-down list in the next field.
- d. Select **StateMate Workarea** from the pull-down list in the next field.
- e. In the **Activation Period (msec)** field, enter the time period between calls to the Rational StateMate Block execution code. This value must be greater than or equal to 50.
- f. Click **Import/Sync** to validate the previous selections and perform the import and synchronization operations if the entries are valid. If one or more of the entries contain errors, the system displays a validation error message. See [Troubleshooting Rational StateMate with Rational Rhapsody](#) for more information.

Connecting and synchronizing Rational Statemate and Rational Rhapsody

The StatemateBlock, created in the previous procedure, operates as a black-box for Rational Statemate code within the Rational Rhapsody architecture once it has been connected and synchronized. The StatemateBlock interface of the top-level flowing data within the Rational Statemate model is specified in Rational Rhapsody using flowports.

To connect the two models:

1. In the Rational Rhapsody diagram from the previous procedure, use the Object  icon to create a object with the appropriately named flowports.
2. Connect the Rational Rhapsody flowports to the ports on the StatemateBlock object via links.
3. To produce an executable, perform code generation and build the entire Rational Rhapsody model.

The StatemateBlock in Rational Rhapsody automatically synchronizes with the Rational Statemate model and adds or removes flowports from the StatemateBlock to reflect any changes made in the Rational Statemate top-level flowing data. The synchronization operation uses a Rational Rhapsody Block Configuration containing the following Rational Statemate data:

- ♦ Rational Statemate MicroC Profile with a single module
- ♦ Rational Statemate charts that are in the scope of the MicroC profile. The top-level chart must have a single top-level (regular) Activity
- ♦ Rational Statemate Panels that are in the scope

Troubleshooting Rational StateMate with Rational Rhapsody

When entering information into the Import/Sync StateMate Block window, you might receive one of these error messages. The table shows the possible error messages and their explanations.

Error Message	Explanation
"Cannot load libraries. Please make sure you are using the correct StateMate installation path."	Rational Rhapsody was unable to locate the stmBlockInterfaceDll.dll in the bin directory of the installation path entered into the window. Correct the Rational StateMate Installation Path so that the DLL can be located.
"PM Filepath not found. Please specify a valid PM path."	The Rational StateMate PM file name entered into the Rational StateMate PM Location field must contain "pm.dat" in the name.
"Invalid StateMate Project. Please select a StateMate Project before pushing OK."	The Import/Sync process has checked the Rational StateMate MicroC profile and the Rational StateMate project was not found or the project name did not match the one entered in the Rhapsody window.
"Invalid Rhapsody Block name. Please select a StateMate Block before pushing OK."	The Import/Sync process has checked the Rational StateMate MicroC profile and cannot locate the StateMateBlock that was entered in the Rhapsody window.
"Invalid StateMate Workarea name. Please select a StateMate Workarea before pushing OK."	The Import/Sync process has checked the Rational StateMate MicroC profile and the Rational StateMate Workarea name was not found that matched the name entered in the Rhapsody window.
"Missing StateMate Block's charts. Would you like to perform check-out and generate code now?"	The Import/Sync process checked for the Rational StateMate code that should have been generated as described in Preparing a Rational StateMateBlock for Rational Rhapsody . If it needs to be generated, click Yes in this error message box.
"Missing generated code for StateMateBlock. Would you like to generate code now?"	The Import/Sync process checked for the Rational StateMate code that should have been generated as described in Preparing a Rational StateMateBlock for Rational Rhapsody . If it needs to be generated, click Yes in this error message box.
"Missing required files. Cannot synchronize with StateMate model."	If you selected No when the system offered to generate the Rational StateMate code (see the two previous error messages), this error message indicates that the Rational StateMate model cannot be synchronized with the Rational Rhapsody until the Rational StateMate code has been generated.

IBM Rational DOORS interface

Software engineers typically need to show traceability of requirements between customer documents and specifications. Moreover, understanding the ramifications of adding or deleting requirements in complex designs, or ascertaining which requirements drove the creation of certain design elements, can be a daunting task.

Rational Rhapsody works with the Dynamic Object Oriented Requirements System (DOORS) to track and manage design requirements throughout the lifetime of a project and to navigate between the design and the requirements, in either direction, online.

The Rational DOORS interface exports design information stored in Rational Rhapsody to the Rational DOORS environment. Design information can include classes, variable and type information, design diagrams, statecharts, and transitions. In Rational DOORS, the information is represented in a logical form as hierarchical requirements inside formal modules reflecting the original hierarchy of the elements in the Rational Rhapsody model. Thus, consistency is maintained between both environments.

The requirements management task is performed within Rational DOORS. Typically, the Rational DOORS tool maintains project documents, user documents, and documentation of changes. System specification and modeling are performed within Rational Rhapsody. The model is built, however, to meet the requirements stored in Rational DOORS, which is the owner of the requirements. Prototyping and analysis done in Rational Rhapsody verify that the model is consistent with your requirements.

The interface works by sharing information between the Rational Rhapsody model and the Rational DOORS database. Requirements are traced by transferring shadow copies of Rational Rhapsody elements into a Rational DOORS formal module, where the shadows are internally linked into the Rational DOORS database.

Note

A “Rhapsody handle” string is attached to each Rational DOORS shadow object to trace the connection from the Rational DOORS shadow to the original Rational Rhapsody element.

Installation requirements

To use the Rational DOORS interface:

- ◆ Copy the `dxlapi.dll` dynamic link library from the `bin` directory in the Rational DOORS installation to your `winnt\system32` directory.
- ◆ Make sure that the file `pc_server.dxl` is in the `$OMR00T\etc` directory of the Rational Rhapsody installation.
- ◆ Rational DOORS must be on the local machine.

Rational Rhapsody reads the locations of your Rational DOORS installation and license from the Windows registry and the `LM_LICENSE_FILE` environment variable. If for some reason Rational DOORS is not registered in the registry in the normal way and your `LM_LICENSE_FILE` variable is not set, you can manually set the `InstallationDir` and `LmLicenseFile` properties under `RTInterface::DOORS`. If these properties are set, they override the registry value or environment variable.

To set the Rational DOORS properties:

1. Click **File > Project Properties** and select `RTInterface::DOORS`.
2. Set the `InstallationDir` property to the location of your Rational DOORS installation. For example:

`d:\doors`

3. Set the `LmLicenseFile` property to the location of your Rational DOORS license. For example:

`d:\doors\lib\license.dat`

If you are using a client license for Rational DOORS, you can enter a port number and server using the following format:

`<port>@<server>`

Rational DOORS version 7.0

By default, Rational DOORS 7.0 does not show the links module. To show the links module (named “Rhapsody_links” by default), select **View > Show Link Modules** in the main Rational DOORS window.

Solaris-specific information

Rational Rhapsody on Solaris supports Rational DOORS 4.1.4 only. Use the following settings:


```
setenv DOORSHOME /tools/DOORS4.1.4
setenv SERVERDATA /tools/DOORS4.1.4/data
setenv PORTNUMBER 36677
setenv DOORSDATA /tools/DOORS4.1.4/data
set path = ( $path $DOORSHOME/bin )
setenv LM_LICENSE_FILE 7192@lily
```

In these settings, `lily` is the name of the Rational DOORS license server.

To use Rational DOORS 6.0 on Solaris systems, use Rational Rhapsody and the following settings:

```
setenv DOORSHOME /tools/doors6.0
setenv SERVERDATA /tools/doors6.0/data
setenv PORTNUMBER 36677
setenv DOORSDATA $PORTNUMBER@bee
setenv LOCALDATA /tools/doors6.0/data
set path = ( $path $DOORSHOME/bin )
setenv LM_LICENSE_FILE 7192@lily
```

In these settings:

- ◆ `lily` is the name of the Rational DOORS license server.
- ◆ `bee` is the name of the Rational DOORS data server.

Using Rational Rhapsody with Rational DOORS

The general process for using Rational Rhapsody with Rational DOORS is as follows:

1. Set up a project within Rational DOORS.
2. Capture requirements and other design information in Rational DOORS.
 - Note:** Rational DOORS is the owner of the requirements. If you need to make changes to requirements, make them in Rational DOORS.
3. Capture the high-level use cases, structure, sequences, and behavior of the system in Rational Rhapsody.
4. Select elements from the Rational Rhapsody model that you want to trace to elements in Rational DOORS.
5. Export the selected elements as shadow copies to the Rational DOORS database.
6. Navigate to the exported elements in Rational DOORS from the Rational Rhapsody browser.
7. Create links within Rational DOORS between the requirements and shadow copies.

8. Run the Check tool to verify the consistency of shadows in the Rational DOORS database with elements in the Rational Rhapsody repository and the completeness of the links between the two.
9. Navigate from the Rational DOORS database to the respective elements in Rational Rhapsody, as wanted.

Configuring Rational Rhapsody and Rational DOORS with the Gateway wizard

The Rational Rhapsody Gateway synchronizes Rational Rhapsody models with Rational DOORS and other requirement management tools. The Gateway DOORS wizard simplifies the set up process to link Rational Rhapsody and Rational DOORS. You can use the wizard to create new Rational DOORS documents or open the wizard for an existing Rational DOORS document in order to make some customizations. The wizard provides a step-by-step configuration of the Rational DOORS document capture.

The two types of customizations are Rational DOORS Basic and Rational DOORS Advanced. The common configuration for Rational DOORS capture, Rational DOORS Basic type, uses the Object ID as the requirement identifier. Rational DOORS Advanced type uses a ReqID attribute as the requirement identifier.

To configure Rational DOORS using the Rational Rhapsody Gateway wizard:

1. Open Rational Rhapsody Gateway.
2. Choose **File > DOORS Wizard**. The wizard inserts a new Rational DOORS document in the project editor.
3. Enter the following information for the new document:
 - ◆ Document name
 - ◆ Rational DOORS Type to used directly or as a template for new type. Enter the New Type name when creating a new type.
 - ◆ Types from library such as Rational DOORS Advanced or Rational DOORS Basic
4. In the Type Customization window, configure the Requirement and Attribute capture in the **Kind** column for **ReqID**. Select one of the following options:
 - ◆ **Ident** use the attribute as Requirement identifier.
 - ◆ **Label** use the attribute as Requirement label.
 - ◆ **Text** use the attribute as Requirement text.
 - ◆ **Present** check that attribute is present on the Rational DOORS object being captured, a **Condition** can be provided to test the value of the attribute (regular expression).

- ♦ **Absent** check that attribute with optional **Condition** is missing on the Rational DOORS object being captured.
 - ♦ **Attribute** create an Attribute type to capture the Rational DOORS attribute in Rational Rhapsody Gateway.
 - ♦ **Ignore** keep this choice when the Rational DOORS attribute is not necessary for the Rational Rhapsody Gateway.
5. In the Set Links to capture window from the list of link modules, select the links to capture.
 6. You can also select one of the two global settings that can be configured for the Rational DOORS document:
 - ♦ **Capture diagrams** allows capture of pictures that are in-lined in the Rational DOORS Object Text.
 - ♦ **Extract only defined attributes** allows saving conversion time by not converting attributes that are not used by the Rational DOORS type.

Requirements synchronization in Rational DOORS and Rational Rhapsody

You can edit requirements in both Rational Rhapsody and Rational DOORS without regard to which tool created the original requirements.

Based on the configuration settings, the requirements synchronization process then compares changes to the Requirement class. The synchronization operation compares important class features such:

- ♦ GUID of Model Element (if any)
- ♦ Identifier of the requirement
- ♦ Label
- ♦ Text
- ♦ Attributes
 - ♦ Name and values
 - ♦ Pictures

The Rational Rhapsody Gateway synchronization preview displays the differences in the Rational Rhapsody and Rational DOORS versions of the requirements so that you can select the appropriate action.

Navigating from Rational DOORS to Rational Rhapsody

You can select an object in the Rational DOORS exported elements module and navigate to the matching element in the Rational Rhapsody model. The navigation operation highlights the selected element in Rational Rhapsody. When navigating to a browser element, such as a class, the element is highlighted in the Rational Rhapsody browser. In the case of a state or transition, the matching statechart opens and the appropriate state or transition is highlighted. In the case of diagrams, the diagram will open in Rational Rhapsody.

To navigate from Rational DOORS to Rational Rhapsody:

1. Select any shadow object in the Rational DOORS formal module.
2. Select **Rhapsody > Navigate to Rhapsody**. The Rational Rhapsody application loads with the matching element highlighted.

Rational DOORS projects

The project must already exist in Rational DOORS before you can connect to it from Rational Rhapsody. In Rational DOORS, create the project to which you will export Rational Rhapsody data if the Rational DOORS project does not already exist.

Invoking the Rational DOORS interface

To start the Rational DOORS interface in Rational Rhapsody:

1. Open the Rational Rhapsody project you want to export.
2. Choose **Tools > DOORS Interface**. The DOORS Interface window opens.
3. In the **Project Name** box, type the name of the Rational DOORS project to which you want to connect.
4. Select the mode using the **Run in Batch Mode** option. The default value is interactive mode (unchecked).

To be able to navigate to Rational DOORS objects from the Rational Rhapsody browser, you must use interactive mode.

Note

If you want to launch Rational DOORS, click **Login** and provide your Rational DOORS username and password.

Set export options

Export options enable you to control:

- ◆ Which Rational Rhapsody design elements are exported
- ◆ Whether all elements are exported to one Rational DOORS formal module or to several modules
- ◆ Whether each package is exported to a separate formal module
- ◆ Which metatypes (such as packages or classes) to export

The **Export All** option enables you to export all design elements in the current Rational Rhapsody project as shadows to Rational DOORS.

To export all design elements in the current Rational Rhapsody project as shadows to Rational DOORS, select the **Export All** check box. When you select **Export All**, the browser tree is unavailable, indicating that you can no longer select individual packages or diagrams.

To explicitly select individual packages or diagrams, clear the **All types** check box and select one or more element metatypes to export using the browser tree.

You can export nested packages to Rational DOORS. When you select a package to export, that package and all the packages nested under it are exported.

Identify which formal modules to create

The **Create module per package** option enables you to export design elements from the Rational Rhapsody project to formal modules with the same names as their packages in the Rational DOORS project. If this option is disabled, all elements are exported to a single formal module named RHAPSODY_MODULE.

Note the following information:

- ◆ When models are exported using the **Create module per package** option, you cannot navigate to Rational DOORS from a class or top-level object model diagram (one that is not in a package). In both cases, you will receive a message stating that the element's package could not be found.
- ◆ While you are in **Create module per package mode**, a new empty package will not be detected; in this mode, the package has no shadow in Rational DOORS, only a module.

Selecting Rational DOORS export options

Export options enable you to select the types of elements that can be exported, rather than the individual elements that actually are exported. For example, if you choose to export elements of metatype Package, you can still opt to export one particular package but not another. If you choose not to export elements of metatype Package, no individual elements of that type will be exported.

Examples of exportable metatypes include packages, classes, attributes, object model diagrams, statecharts, relations, operations, states, activity flows, and constraints. Generally, all of the metatypes that you see in the Rational Rhapsody browser are exportable, and more. States and transitions of statecharts are also exportable metatypes.

To select which metatypes to export:

1. Click **Options**. The Export Options window opens.
2. To export all metatypes in the Rational Rhapsody model to Rational DOORS, select the **All types** check box. The browse tree is unavailable, indicating that you can no longer select individual metatypes.

To explicitly select individual metatypes to export, clear the **All types** check box and select one or more element metatypes to export using the browser tree.

3. To export the graphics of diagrams and statecharts to Rational DOORS, select the **Diagram images** check box, or set the `RTInterface::ExportOptions::ExportPictures` property to Checked. When you have turned on this option, every diagram that you export to Rational DOORS has an OLE object inserted in its shadow. The OLE object holds the diagram graphics as an RTF file stored in the Rational Rhapsody project directory.

Note: If WORD is not on the machine, the OLE object will not be created.

4. To export element labels instead of names, select the **Export Labels** check box.
5. There are two kinds of deletion:
 - a. **Hard delete** means the element and its link is deleted from the Rational DOORS database.
 - b. **Soft delete** means the element is marked as deleted, but remains in the database so it can be recovered. The link is deleted.

The property `RTInterface::ExportOptions::PurgeOnDelete` controls the deletion type used in Rational DOORS. By default, this property is set to Checked (hard delete).

To permanently delete the element, select the **Purge on Delete** check box.

Note the following information:

- When there is an extra element in Rational DOORS that does not exist in Rational Rhapsody, the system asks whether you want to delete it.
- If you soft delete an element and later create an element with the same name, a new shadow element is created in Rational DOORS. The old one is not used.

The metatypes are shown in hierarchical order, with packages and diagrams at the top of the tree. This is analogous to the way metatypes are shown in the browser. The same information hierarchy is maintained in the Rational DOORS formal module as in the Rational Rhapsody model.

Note the following information:

- ♦ You can deselect a subordinate metatype only if its higher-level metatype is selected. For example, you must select packages in order to export classes, events, types, globals, use cases, or actors.
- ♦ When you export only particular types of data, change those elements and then re-export the model without exiting Rational Rhapsody, Rational DOORS “sees” only those elements that were originally exported and updates them in Rational DOORS.
- ♦ If you have exited Rational Rhapsody and re-opened it before re-exporting, the default setting in the Export Options window returns to **All Types**.
- ♦ States are organized according to their position in the state hierarchy in the source statechart. Transitions are organized under their source state. The hierarchy of transitions and states is the same as in the Rational Rhapsody reporter tool.
- ♦ Constraints can only be exported to Rational DOORS with their owners. Constraint shadows become the children of their owner’s shadow.
- ♦ Diagram shadows are created in the shadow of the module for the package to which they belong.
- ♦ Generalization of classes, use cases, and actors are mentioned in the derived shadow’s attributes in Rational DOORS. The short text attribute in Rational DOORS holds the following code:

```
<Super ELEMENT_TYPE> : <object_name>
```

In this syntax, <object_name> is the name of the parent relating to that shadow. To view the short text attribute in Rational DOORS, right-click the shadow of the element and select **Edit**.

Linking the Rational DOORS data

You link Rational Rhapsody elements to Rational DOORS requirements by first exporting information that identifies the elements to Rational DOORS. The information is inserted into a Rational DOORS module as an object. You can link any exported element that is not undefined or unresolved in Rational Rhapsody.

Once you have set the wanted export options, you are ready to export the project.

To export the project:

1. In the DOORS Interface window, click **Export**. The Login window opens.
2. Type your Rational DOORS user name and password.
3. Click **OK**.

If Rational Rhapsody cannot connect to the Rational DOORS project, the specific Rational DOORS error message displays. If Rational DOORS cannot be run at all, Rational Rhapsody displays an error message.

Once you have successfully logged in, Rational Rhapsody exports design information to the Rational DOORS project. For each Rational Rhapsody element for which the Rational DOORS project does not yet have a shadow, one is created. If the Rational DOORS project already has a shadow for a particular element, the shadow is updated with current information.

After all shadow information is updated in the Rational DOORS project, Rational Rhapsody automatically checks the data to verify that all elements were correctly exported.

Rational Rhapsody remains connected to Rational DOORS for the duration of the Rational Rhapsody session. Therefore, if you close the DOORS Interface window, you do not need to reconnect to Rational DOORS during the same Rational Rhapsody session.

Together with each shadow element, Rational Rhapsody exports structure information that allows Rational DOORS to mirror the hierarchy of information shown in the Rational Rhapsody browser. Rational Rhapsody also maintains internal information on exported elements.

Links in Rational DOORS between use case and sequence diagrams

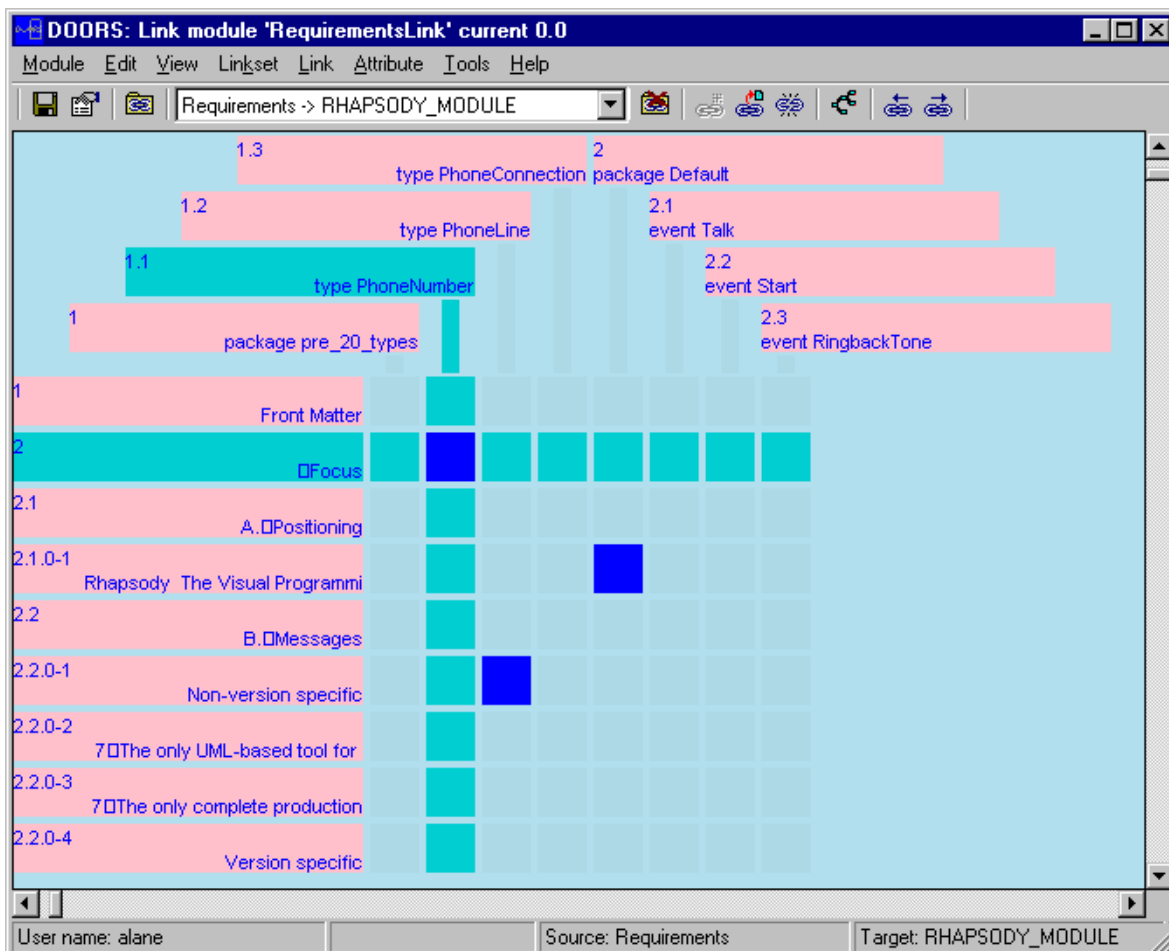
Certain element types in Rational Rhapsody are connected in logical relationships with each other. For example, a use case diagram in Rational Rhapsody can hold references to the sequence diagram that describes it. Rational Rhapsody creates a link between the relational shadows (for example, the use case shadow and its corresponding sequence diagram shadow) during the export operation. These connections are expressed in the Rational DOORS formal module, like any other Rational Rhapsody model information.

Linked elements in Rational Rhapsody include:

- ◆ Use cases and their sequence diagrams
- ◆ Use cases and their collaboration diagrams
- ◆ Sequence and collaboration diagrams, and classes that participate in each.

A link module named Rhapsody_Links (default name) is added to the Rational DOORS project to describe the links between the shadows. You can control the name of the link module using the property `RTInterface::DOORS::LinkModuleName`.

The link module describes the links as entry points in a matrix of the shadows. Links can be between shadows in the same module or between shadows from different modules. The blue squares specify a link between the use case “arming and disarming” and sequence diagram “Arming the a...” and between the use case “changing code” to the sequence diagram “Changing the c...”



A link is graphically specified in the Rational DOORS formal module as an arrowhead, red for outgoing links and orange for incoming links.

To navigate from a shadow to its links shadows:

1. Position the cursor on the arrow specifying the link.
2. Click the right mouse button.
3. Select the wanted shadow from the shadows links list.

Information stored in Rational DOORS

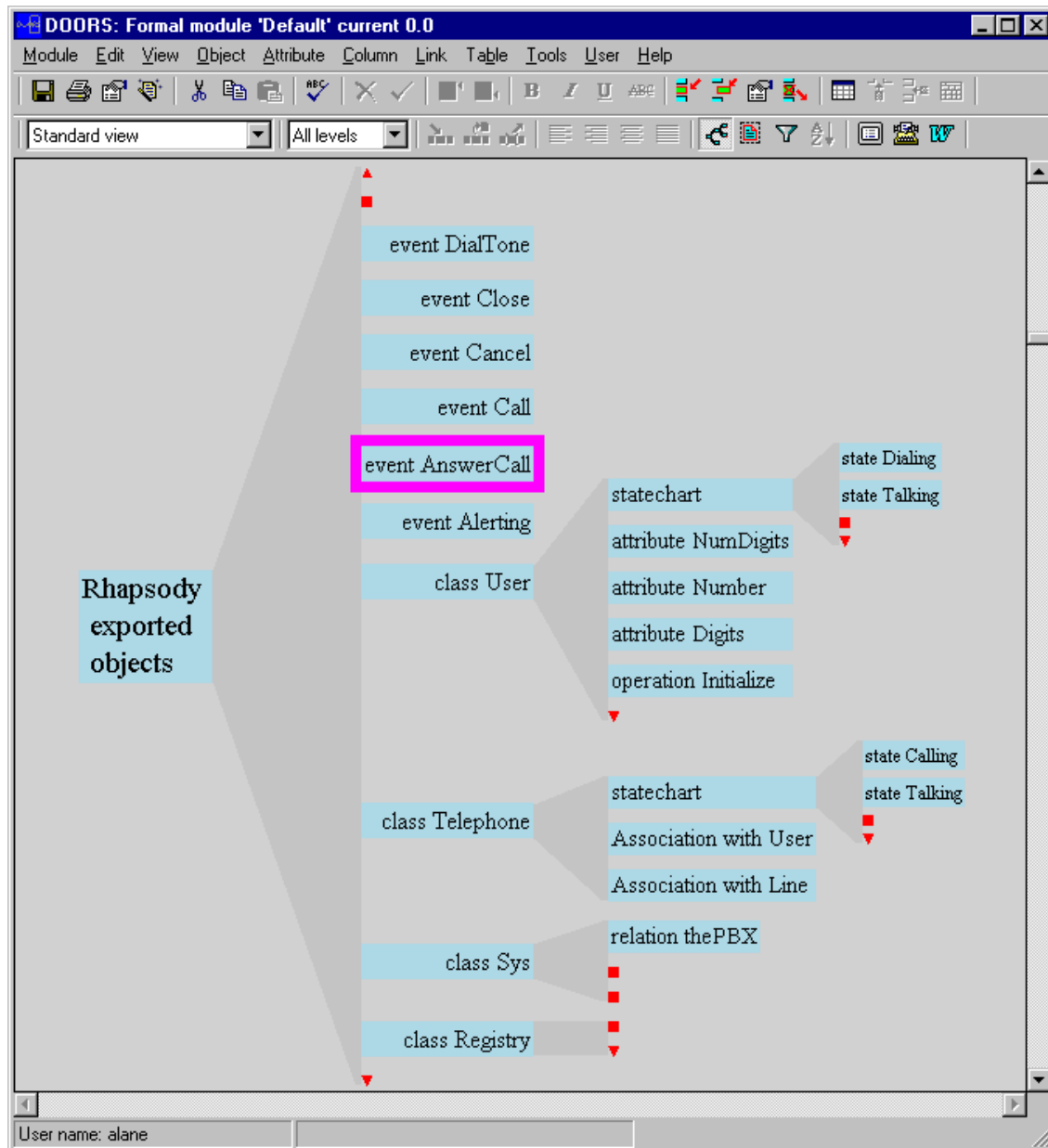
Each exported object is a Rational DOORS shadow element containing the following information:

- ♦ The name of the Rational Rhapsody element.
- ♦ The package or diagram to which the element belongs in Rational Rhapsody.

Note the following information:

- If you selected the option **Create module per package**, each exported element belongs to a formal module corresponding to the package in Rational Rhapsody. Otherwise, the Rational DOORS object is hierarchically located under the object representing the package in the `RHAPSODY_MODULE`.
 - If you want the Rational DOORS project name to reflect the Rational Rhapsody project name, set the `RTInterface::DOORS::ModuleNameFromProject` property to `Checked`. If you set the property to `Checked`, the elements are exported to a formal module matching the name of the Rational Rhapsody project, plus a leading “R”. For example, if the project name is `Pbx`, the formal module name will be `RPbx`.
- ♦ The type of the Rational Rhapsody element.
 - ♦ The description of the Rational Rhapsody element.
 - ♦ In special cases, descriptive information is added to the shadow element in Rational DOORS. For relations, for example, information is added similar to that contained in a Rational Rhapsody report.

The following figure shows the graphical view of a Rational Rhapsody project that has been exported to Rational DOORS. It shows the Default package, which has been exported to a Rational DOORS formal module of the same name, and the hierarchy of some of the shadow objects, such as events, classes, statecharts, and states, that are part of the package.



Rational DOORS information stored in Rational Rhapsody

Rational Rhapsody maintains unique identification information for each design element that was successfully exported to Rational DOORS so Rational Rhapsody can retrieve the Rational DOORS internal data for a shadow at any time.

Data checking

The Check Data operation checks for inconsistencies between the Rational Rhapsody source elements and their related shadows in the target Rational DOORS project. Rational Rhapsody automatically starts the Check Data operation after each successful export operation. You can also check data independently of an export.

To start the Check Data operation, click **Check Data**.

If the Check Data operation finds an inconsistency, the Problem Resolution window opens, which lists any inconsistencies found and offers various ways to deal with the problem.

If you try to re-export the model after closing and then reopening Rational Rhapsody, because the Export Options window reverts back to the **All Types** selection, you might receive errors about elements that have not been exported because they were not among those selected for the earlier export. To ensure that the export does not give you errors of this kind, be sure to reselect all the same metatypes as you selected in previous exports.

Problem Description window

The Problem Description window lists inconsistencies discovered by the Check Data operation. To view the next problem, choose one of the operation buttons, such as Ignore or Update, which are active when a problem is selected.

Each problem description includes the following information:

- ◆ Definition of the problem
- ◆ Name of the element in Rational Rhapsody, if applicable
- ◆ Type of the element in Rational Rhapsody, if applicable
- ◆ Package to which the element belongs in Rational Rhapsody
- ◆ Name of the shadow element in Rational DOORS, if applicable
- ◆ Type of the shadow element in Rational DOORS, if applicable
- ◆ Rational DOORS formal module to which the shadow was exported

To handle a particular consistency problem, select the problem in the Problem Description list. Each type of problem has an appropriate resolution. Rational Rhapsody offers a default resolution and alternatives, depending on the type of problem. The following table summarizes the types of problems detected by the Check Data operation and their possible resolutions.

Problem	Description	Possible Resolutions
DOORS element is outdated.	Invalid name in Rational DOORS for an element that was renamed or otherwise edited in Rational Rhapsody. The message lists the element's name, type, and package in both Rational DOORS and Rational Rhapsody for comparison.	Default: Update Rational DOORS to match the newer information in Rational Rhapsody. Alternate: Ignore.
DOORS element is not connected to any Rational Rhapsody element.	Rational DOORS shadow element points to a non-existent element in Rational Rhapsody.	Default: Delete shadows. Alternate: Ignore.
Package/diagram missing.	The package, diagram, or statechart that existed in the Rational Rhapsody project when the information was initially exported to Rational DOORS no longer exists in the Rational Rhapsody project.	Default: Ignore Alternate: Delete shadows for all related elements
Missing link in Rational Rhapsody element.	The Rational Rhapsody element is not linked to a shadow in Rational DOORS, although a shadow exists in Rational DOORS that matches the element's unique Rational Rhapsody identification information.	Default: Update both the Rational Rhapsody and Rational DOORS information. Alternate 1: Ignore Alternate 2: Delete the element in Rational DOORS and reexport the Rational Rhapsody project.
Rational Rhapsody element is not connected to any DOORS element.	The Rational Rhapsody element has no link to a Rational DOORS shadow element.	Default: Export the element. Alternate: Ignore the error and export later.

The following buttons are available depending on the appropriate solution for a selected problem:

- ♦ **Update** updates the Rational DOORS project with the current Rational Rhapsody information so it is consistent in both places. The Update operation also renews the shadow ID on the Rational Rhapsody side to keep the link to the shadow element current.

Updating is the preferred way to deal with stale data or missing links.

- ♦ **Delete Shadow(s)** deletes the shadow elements in Rational DOORS that are related to the selected problem.

Deleting shadows is the preferred way to deal with shadows that have invalid names.

- ♦ **Delete & Create New** deletes an existing, erroneous shadow and creates a new, accurate one.
- ♦ **Ignore** tells Check Data to proceed with the next problem to be resolved without making any changes to either the Rational Rhapsody or the Rational DOORS project.

Ignoring the inconsistency is the preferred way to deal with missing packages or diagrams.

Mapping Requirements to imported elements

You map the imported shadow elements to requirements by creating a link module in Rational DOORS. To create a link module:

1. In the Rational DOORS Project Manager, select **New > Link Module**.
2. Use the New Link Module window to specify a the name for the link module, its description, and a mapping value.
3. Click **OK** to apply your changes and dismiss the window. The Link Module window opens.
4. Select **File > New > Linkset**. The New Linkset window opens.
5. Specify the source and target modules, then click **OK** to create the link.

Rational DOORS creates a new link module in which requirements from the source module line the left margin of a blank table in the middle of the window, and shadow objects from the target module line the top margin of the table.

To link requirements to shadow objects:

1. Right-click a blank square in the table and select **New Link**.
2. In the Edit Link Object window, edit any of the link attributes.
3. Click **Close**.

That square is marked in blue to indicate that a requirement on the Y-axis is linked to a shadow object on the X-axis.

Ending a Rational DOORS session

To end a Rational DOORS session, do any of the following actions:

- ◆ Click **Logout** in the Rational DOORS Interface window.
- ◆ Close the Rational Rhapsody project.
- ◆ Exit Rational Rhapsody.

If you are running Rational DOORS in interactive mode, you can exit Rational DOORS from the Rational DOORS window.

Rational DOORS with Rational Rhapsody summary

The objective of the Rational DOORS interface is to represent a Rational Rhapsody model in a Rational DOORS module. The formal module must always contain the most current information about Rational Rhapsody model elements. Thus, you can treat a Rational Rhapsody project as a special kind of requirements file filled with model elements. This enables you to link requirements to actual Rational Rhapsody model elements that fulfill those requirements. Remember that Rational DOORS is the owner of the requirements. If you need to make changes to requirements, make them in Rational DOORS.

You can transfer information about complete Rational Rhapsody models or subsets of models into Rational DOORS. You select elements to transfer by constructing a list using the Rational Rhapsody browser. In this way, you can only update subsets of the model if it takes too long to transfer the entire model.

Rational Rose models

The IBM Rational Rose Importer utility imports models created in IBM Rational Rose into Rational Rhapsody. Components of the Rational Rose Logical View, such as packages, classes, and relations between classes, are mapped to similar entities in Rational Rhapsody. Class and state diagrams from Rational Rose are imported as object model diagrams and statecharts, respectively, in Rational Rhapsody. In addition, you can import activity, component, sequence, and use case diagrams, along with templates and template instantiations.

Note

The Rational Rose Importer imports the Rational Rose Logical View, Use Case View, and Component View. It does not import the Deployment View.

In addition, Rational Rose must be on the Rational Rhapsody host machine for import to work. It is not sufficient to simply import a model created in Rational Rose without Rational Rose actually being on the Rational Rhapsody machine.

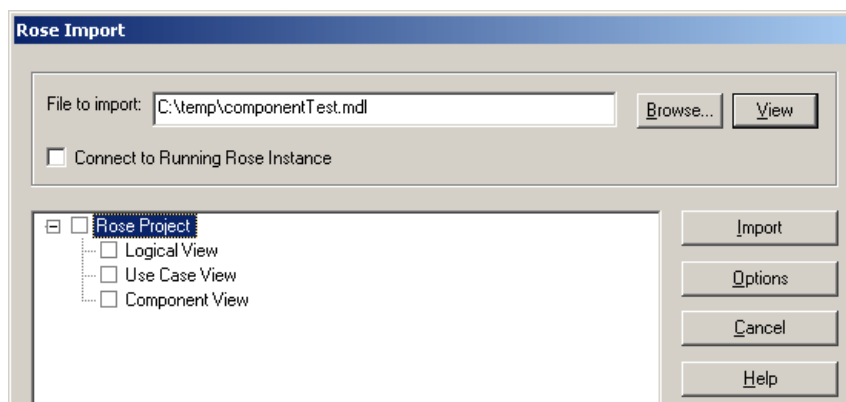
As well, you can (separately) import the code for your imported Rational Rose model and then merge the operation and function bodies from that code into the corresponding imported Rational Rose model. This involves importing the model from Rational Rose (referred to as the imported Rational Rose model), importing the code (using the Rational Rhapsody Reverse Engineering tool), and then merging the imported code with the imported Rational Rose model.

Importing a Rational Rose model

You should ensure that your Rational Rose model is correct from the Rational Rose perspective before you import it into Rational Rhapsody. In addition, a target project must first exist in Rational Rhapsody before you can import a Rational Rose model.

To import a Rational Rose model into Rational Rhapsody:

1. Before importing a Rational Rose model, verify that the model is correct from the Rational Rose perspective. Use the Rational Rose check model function and clear all reported errors in the model before importing it. Attempting to import a model with errors might result in problems using the Rational Rose Importer.
2. With Rational Rhapsody running, create the new project. For example, choose **File > New**. For more information about creating a new Rational Rhapsody project, see [Creating a project](#).
3. To start the process to import your Rational Rose model, select **Tools > Import from Rose > Import Model**.
Notice that Rational Rhapsody automatically opens the Output window for you.
4. To select a Rational Rose model to import, do whichever of the following action is applicable for you to fill in the **File to import** box:
 - ♦ If you have the Rational Rose environment and the Rational Rose model you want to import open, select the **Connect to Running Rose** check box to fill in the **File to import** box.
 - ♦ If you do not have the Rational Rose environment open, use the **Browse** button to locate the Rational Rose .mdl file you want to import. Or you can type the name, including the full path, of the Rational Rose model in the **File to import** box.
5. Once the Rational Rose .mdl filename displays in the **File to import** box, the Logical View, Use case View, and Component View branches for the Rational Rose model to be imported are displayed on the Rose Import window.



6. Expand the contents of a view choice and select the elements you want to import. Note the following information:
 - ♦ Clicking the check box for the main (top) branch selects or clears all sub-branches and their elements.
 - ♦ Clicking the check box for a sub-branch selects or clears that sub-branch and all its elements.
 - ♦ Right-clicking a check box either clears or selects that specific element, depending on its current state.

7. To select your import options, click the **Options** button to open the Import Options window:

- ♦ **Import statecharts and activity diagrams, Import object model diagrams, Import Associations with no names.** Decide (select/clear the check boxes) if you want to include statecharts and activity diagrams and/or object model diagrams and/or associations with no names.
- ♦ If you want the imported Rational Rose project to have the look-and-feel of a Rational Rose project, select the **Use Rose Look-and-feel** check box.

Note: This **Use Rose Look-and-feel** check box is disabled on the re-import of a model if the check box was selected on the original import of the model.

- ♦ If you want to import properties from the Rose model, select the **Import Properties** check box and use the **Browse** button to point to the needed property XML map file. See [Setting up the XML map file for importing Rational Rose properties](#).

Note: Rational Rhapsody will automatically use these settings the next time you do an import. For example, if you select the **Import statecharts and activity diagrams** check box and clear the **Import object model diagrams** check box, this setting will be used for all subsequent imports until you change the settings again.

8. Before you import, you might want to be sure of or do the following information:

- ♦ If you are re-importing the same packages from Rational Rose, remember that the names in Rational Rhapsody and in Rational Rose must be exactly the same.
- ♦ If necessary, move the Rational Rose Import window away from the Output window before you start the import so that you can see any messages as they occur.

Note: While Rational Rose allows names with spaces, Rational Rhapsody does not. Rational Rhapsody approximates spaces in names by replacing them with underscores. For example, a package named “Course roster” in Rational Rose becomes “Course_roster” when imported into Rational Rhapsody. There are other characters not allowed in Rational Rhapsody names (such as &, #, \$, and %). For these characters, Rational Rhapsody will use underscores or truncate the names.

9. To close the Import Options window, click **OK**.
10. On the Rose Import window, to do the import, click the **Import** button.
11. If a top-level package with the same name as one you are importing already exists in the Rational Rhapsody model, the following message displays:

Packages Logical_View, Use_Case_View, Component_View already exist. Do you want to continue?

To continue with the import, click **Yes**. This means that any package that is re-imported will be totally overwritten.

12. The import process begins. Progress meters and possible messages regarding “lost data” are written to the Output window. The following examples show types of messages:

Error: Can't import association itsTerminal from IControlDevice. It has only one role.

...

Error: Can't add operation GetPropertyValue to class IControlDevice, there is a name or signature clash.

...

Error: Can't override statechart for derived class IAlarm.

Note

The import process creates a log file, **importRose.log**, which is located in the folder of the active project.

Setting up the XML map file for importing Rational Rose properties

You can include the property data for a Rational Rose model when you import it into Rational Rhapsody. You must define which Rational Rose properties you want to import based on the tool, metaclass, and property name, which you define in an XML map file. During the set up for the importing process, you point to the XML map file on the Import Options window (in Rational Rhapsody).

The Rational Rhapsody product includes sample XML map files that you can use to design the needed XML map file. The sample XML files, which you can edit with any text editor, are located in `<Rational Rhapsody installation path>/Share/etc`.

To design your XML map file:

1. Decide which XML map file you want to use.
 - `rose_properties_import.xml` is the basic map file and within it is a list of the common values for attributes and a list of the Rose tools.
 - `rose_properties_import_java.xml` is specifically for importing a Java Rational Rose model.
2. Using any text editor, modify the XML map file. For each property in the XML map file, you must include its metaclass, tool, and property name for the Rational Rose model, as shown in the following example:

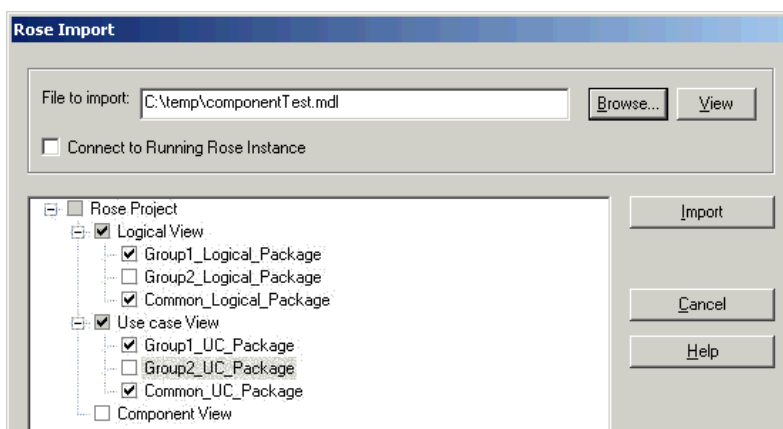
```
<property_map>
  <property>
    <rose metaclass="class"
      tool="Java"
      property_name="Final"/>
  </property>
  <property>
    <rose metaclass="class"
      tool="MSVC"
      property_name="Type"/>
  </property>
```

Note

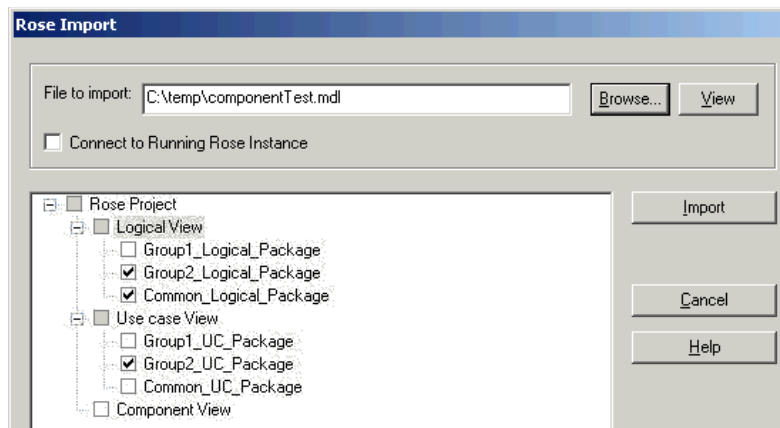
The default location and name for the XML map file is in the `<Rational Rhapsody installation path>/Share/etc/rose_properties_import.xml` path. You can change this by modifying the `RoseInterface::Import::PropertiesXMLPath` property to point to another path for the XML map file.

Incremental import of Rational Rose models

Rational Rhapsody allows you to import Rational Rose models incrementally. This makes it easier to import large models according to your workflow processes. For example, for a very large model, you might have more than one team, with each team assigned a specific part of the model. You can import each part of the model in separate import sessions. For example, you might import all of Group1, as shown in the following figure:



Then you could import Group2, as shown in the following figure. Notice also in this example you can re-import a Rational Rose package (in the Logical View, the Common_Logical_Package has been selected again for importing). When you import a Rational Rose package that has already been imported, the incremental import process will completely override the contents of that package in Rational Rhapsody.



Incrementally imported parts of a Rational Rose model will be integrated correctly in Rational Rhapsody when possible. An example of when it is not possible: A class has an association with another class in the original Rational Rose model. During incremental import, only the first class is imported. The other class is located in another package, which will be imported later. The association between these classes will remain unresolved (that is, incomplete) until you import the second class.

Before the import process starts

Prior to the import process, for performance reasons, Rational Rhapsody will close all diagram windows that might be open (neither saving or unloading diagrams). In this event, the following message displays:

All opened diagrams will be closed prior to Rose Import.
Please click the OK button to proceed or the Cancel button to cancel import.
Click **OK** to continue.

About processing time and project size

The internal steps required for this incremental importing process result in slightly longer processing time as well as slight increases in the size of the project. The size increase is due to the data that must be saved to allow the importing of further increments of a model. You can use **Tools > Clean Project Import Data** to delete the data that was stored in order to allow these incremental imports. By default, this menu item is not visible. To make this menu item visible, add the following line to the [General] section of the `rhapsody.ini` file:

```
ShowCleanImportData=TRUE
```

Note

This data is necessary for importing further increments of the model you have imported. Use this menu option only after you have completely finished importing all of the Rational Rose model. Once import data is destroyed, incremental import of the particular Rational Rose model is no longer possible.

Code import

You can import code from a Rational Rose model into your imported Rational Rose model in Rational Rhapsody. Importing code from a Rational Rose model means creating a temporary reverse-engineered package from the source code that was generated from Rational Rose and manually changed after generation.

Note the following points about code import:

- ♦ Before doing code import, turn off DMCA and, if open, close the Active Code View window.
 - DMCA (dynamic mode-code associativity) is the function in Rational Rhapsody that changes the code of a model to correspond to the changes made to a model in Rational Rhapsody. To turn off DMCA, choose **Code > Dynamic Mode Code Associativity > None**.
 - To close the Active Code View window, choose **View > Active Code View**. (A check mark should not appear next to this menu command when the window is closed.)
- ♦ After you have imported the code for the imported Rational Rose model, do not generate code for the imported Rational Rose model in Rational Rhapsody while the reverse-engineered package still exists in the model. You want to avoid situations where, for example, two classes of the same name try to generate code into the same source file, which could corrupt the model if the file is then roundtripped. To avoid this problem, generate code only after the operation bodies have been merged into the imported model and after the reverse-engineered package has been deleted from the model.
- ♦ Be careful as to which classes/objects are assigned to the existing components before code generation after deleting the imported-code/reverse-engineered package.

To import code from a Rational Rose model, use the Rational Rhapsody Reverse Engineering tool. To open this tool, choose **Tools > Import from Rose > Import Code**. For information on how to use the Reverse Engineering tool, see [Reverse engineering](#).

Once you have imported the code, you can merge it into your imported Rational Rose model. See [Merging imported code to the imported Rational Rose model](#).

Merging imported code to the imported Rational Rose model

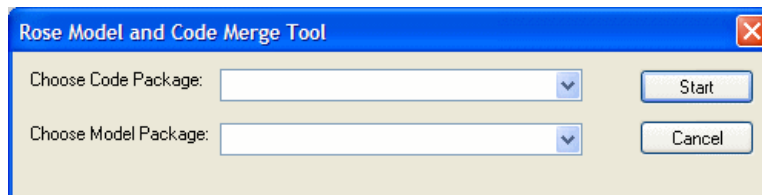
Once you have imported your Rational Rose model into Rational Rhapsody and separately imported the code for that imported Rational Rose model, you can merge the operation and function bodies from the imported code into the corresponding imported Rational Rose model.

Note

This merging process copies the code from all the operations/functions in one top-level package to another. The merging process does not work for operations/functions that might be in subpackages (packages under a top-level package). In addition, the destination package must contain the same class/operation structure as the package that contains the imported code (though the destination package must have operations/functions with empty bodies waiting for the imported code).

To merge imported code to its corresponding imported Rational Rose model:

1. With the imported Rational Rose model opened in Rational Rhapsody, choose **Tools > Import from Rose > Merge Model and Code**.
2. On the Rose Model and Code Merge Tool window, as shown in the following figure, choose the code package (the imported code from your Rational Rose model) from the drop-down list and the model package (in the imported Rational Rose model) to which you want the imported code.



3. Click **Start**.
4. Notice that a log of the merge progress prints on the **Log** tab of the Output window. Upon successful completion of the merge process, you should see the code for the imported operations/functions in the applicable package in your imported Rational Rose model.

How Rational Rose constructs and options map into a Rational Rhapsody model

The following table shows how various Rational Rose constructs and options map into a Rational Rhapsody model. For ease of use, the Rational Rose elements are listed in alphabetical order.

Rational Rose Element or Option	Rational Rhapsody Element	Notes
Abstract class		Not imported.
Action	Action	
Activity diagram	Activity diagram	
Actor	Actor	
Anchor note to item	Anchor	
Association	Link, linktype = association	See Imported association classes .
Cardinality of classes	Part	Class cardinality refers to the number of instances of a class that can be created at run time. A class with exactly one instance has a cardinality of one. In Rational Rhapsody, a class's cardinality is referred to as its <i>multiplicity</i> . The Multiplicity box reflects the cardinality of the class in the original Rational Rose model.
Category	Package	
CategoryDependency	Dependency	
Class	Class	
Class type	Type = class	All types of classes are mapped to classes.
ClassifierRoles	ClassifierRoles	
Collaboration diagram	Collaboration diagram	Not imported.
Component Package	Package	
Component	Component	Since Rational Rhapsody does not allow components to be contained in packages, any imported components will be included under the project level.
Component diagram	Component diagram	
Concurrency—sequential, active, guarded, or synchronous	Concurrency—sequential or active	Operation concurrency is not imported.
Condition	Guard	
Constraints		Not imported.

Rational Rose Element or Option	Rational Rhapsody Element	Notes
Containment—by value, reference, unspecified		Not imported.
Dependency (UCD)	Dependency	
Deployment diagram	Deployment diagram	Not imported.
Derived attributes and relations		Not imported.
End state	Termination connector	
Event	Event	Events trigger transitions from one state to another. Events are imported as classes whose behavior includes triggering state transitions.
Export control		Not imported.
Friend	Property	
Global package		Not imported.
HasRelationship	Link, linktype = aggregation	
Inheritance (use cases)	Inheritance	
InheritRelationship	SuperClass, superevent	
Initial value of attribute		Not imported.
Interface	Class	Interface classes are imported into Rational Rhapsody as classes with virtual operations.
IsConstant (Rational Rose property)		Not imported.
Link Attribute		Not imported.
Link Element		Not imported.
Messages	Messages	
Multiplicity of relations	Multiplicity	
Navigable relation	Feature (from class to class)	In Rational Rhapsody, you cannot add a Navigable feature if there is a navigation (cannot have both Navigable and aggregation).
Nested class		Not imported.
Note	Note	
Operation type—virtual, static, friend, abstract, common	Virtual, static	
OperationIsConst (Rational Rose property)		Not imported.
Parameter	Argument	
Persistence		Not imported.
Private implementation	Private implementation	

Rational Rose Element or Option	Rational Rhapsody Element	Notes
Protected implementation	Protected implementation	
Public implementation	Public implementation	
Qualifier/keys	Qualifier	In Rational Rose, a qualifier might not be a class attribute. In Rational Rhapsody, a qualifier must be a class attribute. Rational Rhapsody approximates qualifiers depending on whether they are also attributes in Rational Rose. If the qualifier is an attribute in Rational Rose, it is mapped to an attribute in Rational Rhapsody. Otherwise, Rational Rhapsody creates an attribute, adds it to the class, and makes it the qualifier. Rational Rose allows multiple qualifiers, whereas Rational Rhapsody allows only one. Therefore, when you import an association with multiple qualifiers, Rational Rhapsody randomly takes the first one it sees.
Qualifier type	Attribute	If the qualifier is not a class attribute, create it.
Relation	MetaLink	Abstract class.
	Relations in UCDs are imported as relations.	
Relation type—by value, by reference, unspecified	All three types map to By reference.	
RealizeRelation	SuperClass	
Role	Role	
Send argument	Action	
Send event	Action	
Send target	Action	The Rational Rose Send event/argument/target are mapped to the Rational Rhapsody action using the following format: <pre>Sendtarget ->GEN(Sendevent(Sendarguments))</pre>
Sequence diagram	Sequence diagram	When Rational Rhapsody imports sequence diagrams from Rational Rose, the Rational Rose ClassifierRoles are converted to Rational Rhapsody ClassifierRoles and Classifiers, and messages are converted to actual operations on the target (receiving) class.

Rational Rose Element or Option	Rational Rhapsody Element	Notes
Space of class		Not imported.
Start state	Initial Connector	Combined with outgoing transition.
State	State	If there is more than one view for a single state in Rational Rose, when imported into Rational Rhapsody, the additional views will be converted into new states in the model with the same characteristics (like you would get with the Copy with Model feature). The name will indicate that it is a new state, but the label will be the same.
Static attributes	Static attributes	
Static relation	Static (relation is a static class member)	
StereoType		Not imported.
Substate	State (with parent)	If there is more than one view for a single substate in Rational Rose, when imported into Rational Rhapsody, the additional views will be converted into new substates in the model with the same characteristics (like you would get with the Copy with Model feature). The name will indicate that it is a new substate, but the label will be the same.
Templates and template instantiations	Templates and template instantiations	
Text box	Note	Same as object model.
Transition	Transition	The format for an activity flow in the diagram is as follows: <Event>[<Guard>]/<Action> If there is more than one view for a single transition in Rational Rose, when imported into Rational Rhapsody, the additional views will be converted into new transitions in the model with the same characteristics (like you would get with the Copy with Model feature). The name will indicate that it is a new transition, but the label will be the same.

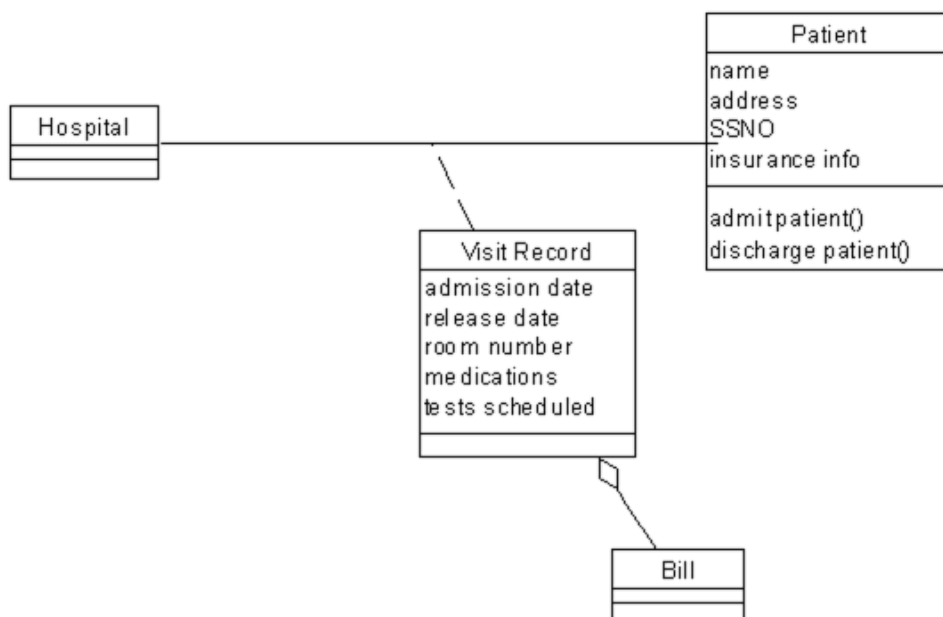
How Rational Rose constructs and options map into a Rational Rhapsody model

Rational Rose Element or Option	Rational Rhapsody Element	Notes
Types—predefined (such as <code>int</code> or <code>float</code>), user-defined, or class.	Type	When you create a user-defined type in Rational Rose, you can give it a name but no declaration. Rational Rhapsody approximates a user-defined type by adding an on-the-fly type with the new type name as its declaration. In Rational Rose, you can also assign a class type, such as <code>ParameterizedClass</code> or <code>InstantiatedClass</code>. Rational Rhapsody approximates class types by creating an on-the-fly type with the class as its declaration.
Use cases	Use cases	
UseRelation (ClassDependency)	Dependency between packages is saved only in the graphical interface.	

Imported association classes

If a class does not have associations or a statechart, it is imported as an association class; otherwise, it is imported as a regular class.

Consider the following hospital model:



In this example, Visit Record is a class associated to the Hospital_Patient association. Therefore, it could be imported as association class.

If the Visit Record class has a statechart or associations with other classes, it will not be imported as an association class, but will be imported as a class. As shown in the figure, because Visit Record has an association with class Bill, it will be imported as a regular class. However, the association Hospital_Patient will have a hyperlink to this class.

If Visit Record does not have associations or a statechart, it is imported as association class. That means:

- ♦ The name of the association Hospital_Patient will be Visit Record.
- ♦ The attributes and operations of Visit Record will be displayed under the association class.

XMI exchange tools

XMI (XML Metadata Interchange) is a format specification produced by the Object Management Group (OMG). The XMI format allows the interchange of objects and models through an XMI formatted file. This is commonly used to exchange UML models between other tools or software.

In addition to XMI, Rational Rhapsody provides additional tools for developers to examine the models, such as:

- ◆ ReporterPLUS reports in Word, PowerPoint, and HTML format. Reports are created without any conversion to another format.
- ◆ Model simulation capabilities show how the model components work together.
- ◆ The COM API exports a set of COM interfaces representing the metamodel objects and application operational functions.
- ◆ Write macros provide a means to examine the model within Rational Rhapsody.

Using XMI in Rational Rhapsody development

The Rational Rhapsody XMI export and import feature facilitates the following development tasks:

- ◆ Export an entire Rational Rhapsody model to XMI to be closely examined as a whole
- ◆ Export the whole model to XMI to be searched in an HTML browser
- ◆ Export the model to XMI in order to parse the entire model with another UML tool or a non-UML tool
- ◆ Imports XMI models or pieces of other XMI models into Rational Rhapsody models
- ◆ Exchange models to or from the Tau system

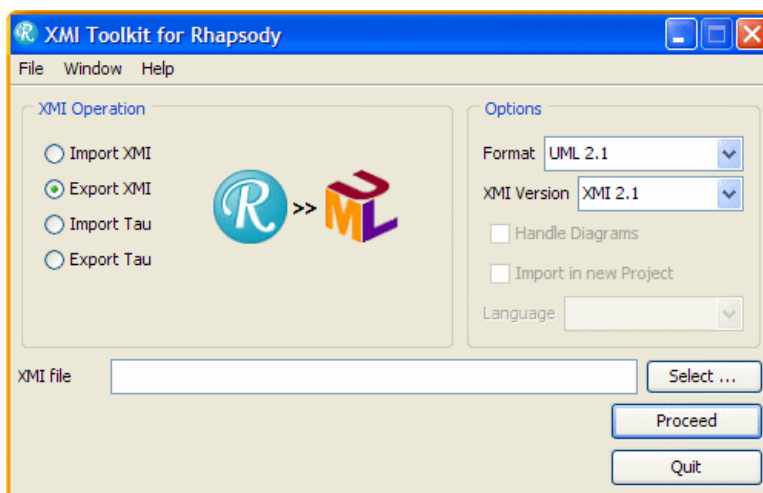
Exporting a model to XMI

Engineers, designers, or architects might export a Rational Rhapsody project to an XMI file for any of the following reasons:

- ◆ Share a model with another UML tool
- ◆ Create a text file for your model so that it can be parsed
- ◆ Create a file that can be searched in an HTML browser

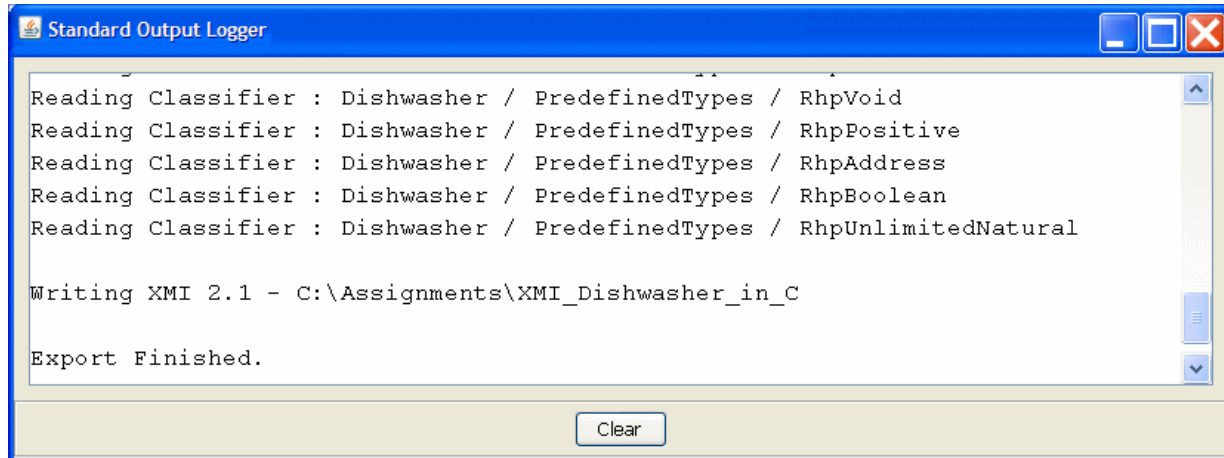
To export a Rational Rhapsody model to an XMI file:

1. Open the model for export in Rational Rhapsody.
2. Choose **Tools > Export XMI from Rhapsody**.
3. In the **XMI Operation** area, select whether you are exporting to XMI or to Tau.
4. Then select the UML **Format** of your project files as either 1.3 or 2.1.
 - If you select UML 2.1, the **XMI Version** is automatically set to 2.1.
 - If you select UML 1.3, you can select 1.0, 1.1, or 1.2 as the XMI Version for your exported file.



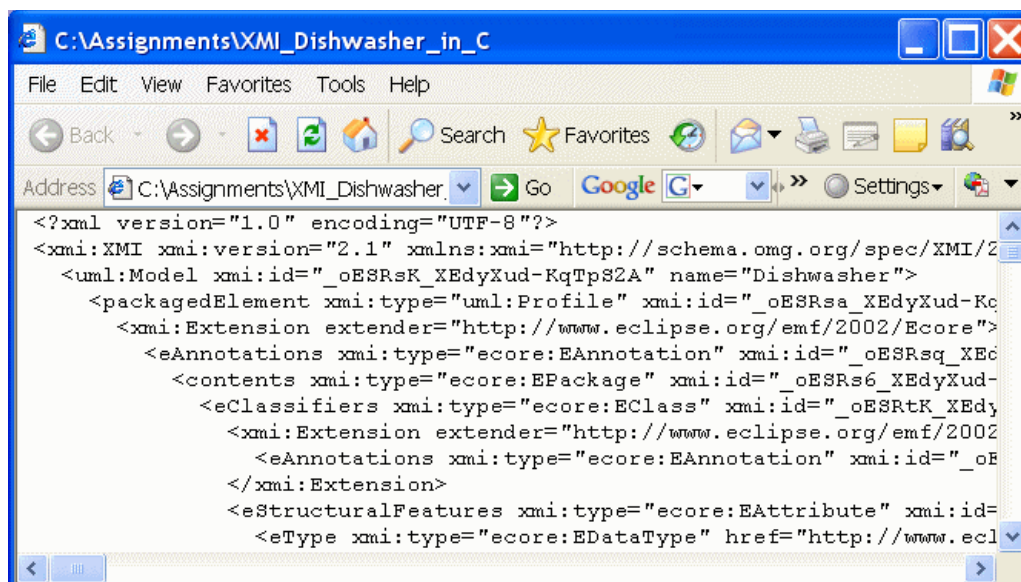
5. If you select the UML 1.3 format, you can decide to **Handle Diagrams** and output the diagrams for the model in the UNISYS extensions format during the export operation.
6. In the **XMI file** box, select a directory for the exported file.
7. Click **Proceed** to export the model as defined.

8. The system displays any messages relating to the export in a window, as shown in the following figure:



Examining the exported file

After exporting a model, you can open the exported file in any standard browser to examine the details of the model. The following example shows an exported Rational Rhapsody model displayed in the *Windows Internet Explorer* browser.



Note

If the **UML 1.3** and **Handle Diagrams** options are selected, the exported file includes the Rational Rhapsody model diagrams in the UNISYS extensions format.

Importing an XMI file to Rational Rhapsody

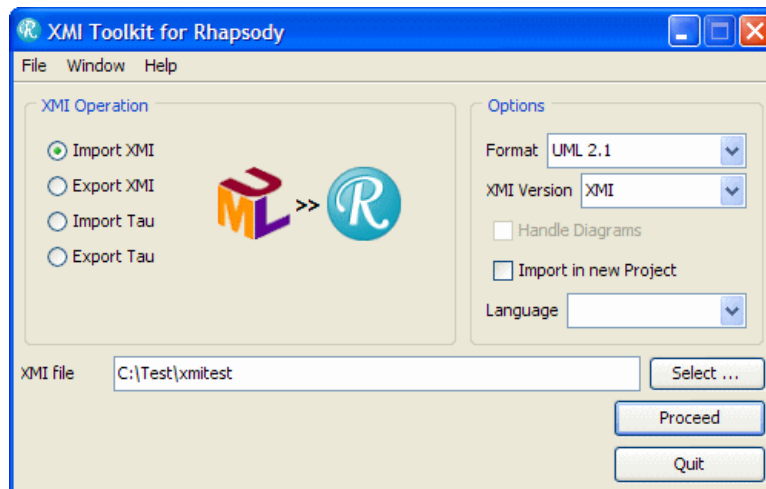
Rational Rhapsody has the ability to import a model from an XMI file into Rational Rhapsody for either of these reasons:

- ♦ A file from another UML tool needs to be brought into Rational Rhapsody. For example, a TAU file could be brought into Rational Rhapsody this way.
- ♦ A file from a non-UML tool needs to be brought into Rational Rhapsody.

Important: Normally you would not export and import XMI files between Rational Rhapsody users. XMI is intended as a means for vendor-neutral data sharing.

To import an XMI file from another source:

1. Create a new Rational Rhapsody project or open an existing Rational Rhapsody project.
2. Choose **Tools > Import XMI into Rhapsody**.
3. In the **XMI Operation** area, select whether you are importing from XMI or from Tau.
4. In this window, select the UML **Format** for the imported file as either 1.3 or 2.1.



5. If you select the UML **Format** for the imported file as UML 1.3, you can also select whether or not the import operations should **Handle Diagrams**.
6. Select the **Language** for your imported file as C, C++, Ada, or Java.
7. In the **XMI file** box, select the directory from which to import the file.
8. Click **Proceed** to import the XMI file as defined and add it to the Rational Rhapsody project.

The system displays messages relating to the import in a window, as shown in [Exporting a model to XMI](#).

More information

For more information about the XMI toolkit features, see the following documentation in <Rational Rhapsody installation path>\Sodius\XMI_Toolkit\doc:

- ◆ User Guide
- ◆ Mapping Rules Overview
- ◆ Rational Rhapsody Rational Tau Integration

Integrating Simulink components

Rational Rhapsody can be used in conjunction with Simulink, the MATLAB extension that allows modeling of continuous processes using block diagrams.

Rational Rhapsody allows you to integrate Simulink models into Rational Rhapsody designs. Simulink models are represented as “Simulink blocks” in the UML model, and these blocks can interact with Rational Rhapsody objects/parts or other Simulink blocks. The integration of Simulink blocks into Rational Rhapsody uses a “black box” approach, in which only the input/output ports of the Simulink blocks are exposed, appearing as flowports in the Rational Rhapsody model. To send/receive data to/from a Simulink block, you use links to connect these flowports to flowports of other Simulink blocks or of other Rational Rhapsody objects. When code is generated for a Rational Rhapsody model containing Simulink blocks, the code generated by Simulink is wrapped into the Rational Rhapsody-generated code.

If changes are made to the Simulink model, you can synchronize the representation of the Simulink model in your Rational Rhapsody project with the updated model.

In general, the process for including such Simulink components in a Rational Rhapsody model is as follows:

1. Build the Simulink model using Real-Time Workshop.
2. Import the model into Rational Rhapsody as a *SimulinkBlock*. The Simulink input and output ports will appear as atomic flowports on the *SimulinkBlock* element. (See [Flow ports](#).)
3. Connect the flowports of the *SimulinkBlock* element to the flowports of the relevant elements in the Rational Rhapsody model.

The following software is required for integrating Simulink components into a Rational Rhapsody model:

- ♦ Matlab must be available and licensed (Matlab 7), with Simulink (version 6) and the Real-Time Workshop component (which generates C and C++ code from Simulink models).
- ♦ Rational Rhapsody 7.0 or greater

Note

The `.. \Samples` directory contains a sample Rational Rhapsody model that includes Simulink integration.

Importing Simulink components

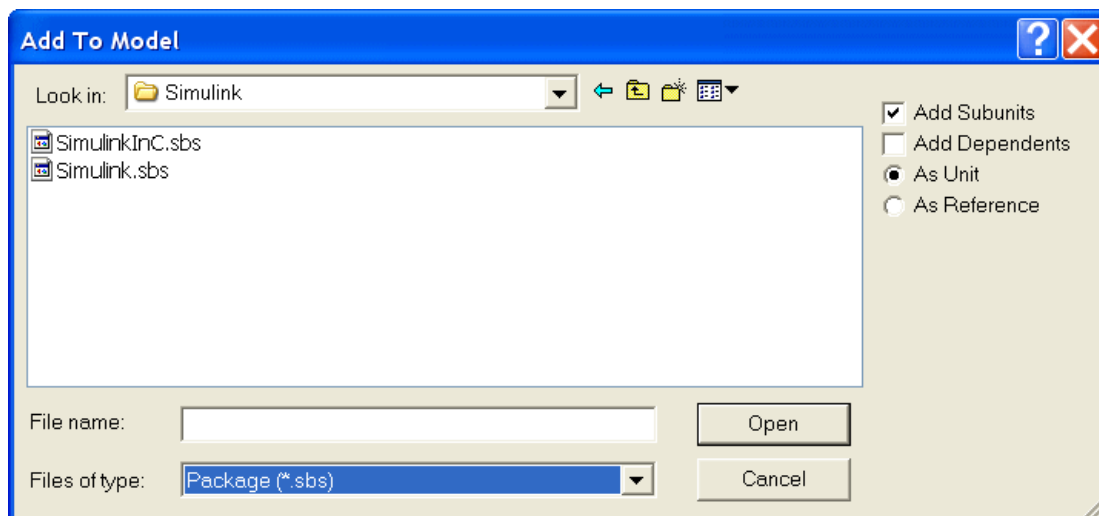
To import a Simulink component, carry out the following steps in Simulink and in Rational Rhapsody:

In Simulink

1. Create a Simulink model, or open an existing one, and save it in your working directory, preferably in the same working directory as your Rational Rhapsody model.
2. For generating code, use the following settings (most are the default settings) in the Real-Time Workshop. You can view the settings by selecting **Tools > Real-Time Workshop > Options**.
 - ◆ Hardware Implementation->Device Type - Unspecified (assume 32bit Generic)
 - ◆ Real-Time Workshop->System target file - ert.tlc
 - ◆ Real-Time Workshop->Language - C or C++ (note that the default setting is C)
 - ◆ Real-Time Workshop->Make command - make_rtw
 - ◆ Real-Time Workshop->Template makefile - ert_default_tmf
3. Generate code for the Simulink model (**Tools > Real-Time Workshop > Build Model**).

In Rational Rhapsody

1. Create a new Rational Rhapsody project.
2. Right-click the project name in the browser and select **Add to Model > Package**.
3. Navigate to your Rational Rhapsody installation's Share/Profiles/Simulink directory. Select the Package (*.sbs) for **Files of type**, as shown in this example.



4. Select the `SimulinkInc.sbs` profile if you are using C and `Simulink.sbs` if you are using C++. Click **Open** to add the selected profile to the project. Check the `Profiles` section in the browser to be certain that the selected Simulink profile is now displayed.
5. Create an object in an object model diagram, and apply the *SimulinkBlock* stereotype to it (in the `Features` window).
6. Right-click the object and select **Import/Sync Simulink Model**.
7. In the window that is displayed, provide the following information:
 - ♦ **Simulink Model File.** The location of the Simulink model file
 - ♦ **Simulink Generated Source Code.** The location of the *.cpp files generated by the Real-Time Workshop (add all files except `ert_main.cpp`).
 - ♦ **Simulink Model Sample Time.** The interval (in milliseconds) at which Rational Rhapsody should activate the Simulink engine.
8. Click **Import/Sync** and wait until Rational Rhapsody creates flowports on the block representing the input and output of the Simulink model.
9. Once the flowports have been created, you can connect the Simulink block to the flowports on other Rational Rhapsody blocks.

Integration of the Simulink-generated code

When Simulink components are imported into a Rational Rhapsody model, the .cpp files generated from the Simulink model using Real-Time Workshop are included as source files in the Rational Rhapsody-generated makefile.

In terms of Rational Rhapsody-generated code, *SimulinkBlock* elements in Rational Rhapsody are classes that are based on a framework class called `OMSimulinkBlock`. The superclass periodically calls the method `doStep()`, which is implemented by the derived class. This method initializes the input port, calls the step function in the Simulink-generated .cpp file, and sets the value of the output after the step. (The output is then relayed via the output flow port.)

The `doStep()` function will be generated once you assign the *SimulinkBlock* with a Simulink model and use the **Import/Sync Simulink Model** context menu command. Note that an Embedded Coder License (ERT) is required for this operation.

Troubleshooting Simulink integration

- ♦ If after importing or synchronizing with your Simulink model, you get an error message about a missing file, *langeng.dll*, verify that MATLAB's \bin\win32 folder is in your PATH environment variable. After adding it, you will have to restart Rational Rhapsody and try reimporting.
- ♦ If you get compilation errors regarding missing include files, look for them in the MATLAB installation directory. After locating them, you can add them to the include search path for the Rational Rhapsody configuration.

Creating Simulink S-functions with Rational Rhapsody

Using Rational Rhapsody in conjunction with Simulink

Rational Rhapsody can be used to create Simulink S-functions that can then be plugged into Simulink models.

The specific steps to carry out to create S-functions in Rational Rhapsody are described in [Creating a Simulink S-function](#).

For a broader picture of S-function creation in Rational Rhapsody, see [S-function creation: behind the scenes](#).

Note

This feature is only applicable in Rational Rhapsody Developer for C.

Creating a Simulink S-function

To create a Simulink S-function in Rational Rhapsody and then use it in Simulink:

1. Create a new Rational Rhapsody project.
2. Create a new configuration and apply the stereotype *S-FunctionConfig* to it.
3. Set the newly-created configuration to be the active configuration.
4. Create a new class, and apply the stereotype *S-FunctionBlock* to it.
5. Add incoming flowports to the class to represent incoming data.
6. Add outgoing flowports to the class to represent outgoing data.
7. For each of the flowports you added, add an attribute to the class to represent the flowport. The attribute must have the same name and be of the same type as the corresponding flowport.
8. Implement a statechart for the class.
9. Generate code for the configuration you created.

10. The output directory for the configuration will include the following items:
 - ♦ generated source files for the model
 - ♦ Rational Rhapsody framework files (from the Rational Rhapsody IDF framework)
 - ♦ a Simulink C template file called *RhapsSFunc_*(the name you gave to the block).c
 - ♦ a mex options file called *MexOpts.txt*
 - ♦ a Simulink model file, representing the S-function block, called *RhapSFunc_*(the name you gave to the block)*_Model.mdl*
11. Open MATLAB and go to the output directory containing the Rational Rhapsody code.
12. Run the command `mex @MexOpts.txt`.

S-function creation: behind the scenes

When you generate code for an *S-FunctionConfig* configuration, Rational Rhapsody performs the following actions:

- ♦ Completes the *sfuntmpl_basic.c* template provided by Simulink, and renames it to reflect the name you assigned to your S-function block.
- ♦ Takes the information you have entered for the S-function block in your project and creates a corresponding Simulink model file, using the name you assigned to your S-function block.
- ♦ Generates a mex options file, containing the necessary compiler switches and list of source files to use.

When you run the mex command using the mex options file generated by Rational Rhapsody, MATLAB's MEX compiler creates a binary file that can be used by Simulink.

Timing and S-Functions

For time-related events, Rational Rhapsody uses the timing mechanism of the target operating system. Since Simulink has its own timing mechanism, Rational Rhapsody takes this into account when generating the S-function code. The Simulink clock is added as an input to the S-function. This is not visible to the user in Rational Rhapsody, but when the resulting files are imported into Simulink, you see a clock element in addition to the element representing the defined S-function.

Limitations

When creating Simulink S-functions in Rational Rhapsody, keep the following information in mind:

- ◆ You can only have one S-function per configuration.
- ◆ The Rational Rhapsody animation feature does not work with S-function blocks.

The Rational Rhapsody command-line interface (CLI)

Rational Rhapsody provides command-line options for individual system features, such as for the [DiffMerge tool functions](#). You can run the full version of Rational Rhapsody (Rhapsody.exe) from the command line. To assist with command-line operation, Rational Rhapsody includes a lightweight non-GUI version of the program (RhapsodyCL.exe), which allows you to use a subset of the full set of Rational Rhapsody command-line options.

Note

RhapsodyCL.exe is located in the same directory as Rhapsody.exe.

RhapsodyCL

RhapsodyCL allows you to use the Rational Rhapsody code-related functions, such as generate and make, in contexts where you do not require the GUI elements, for example, as part of a nightly build procedure. Since RhapsodyCL is designed for tasks such as code generation, it does not support options relating to diagrams, for example, populating a diagram from the command line. It also does not support the commands relating to configuration management and running macros.

You can send the RhapsodyCL application commands using any of these four methods:

- ◆ Command line
- ◆ Batch file
- ◆ Interactive mode
- ◆ Socket mode

Interactive mode

In this mode, RhapsodyCL, switches to a “shell mode” using a prompt to enter commands. You can use either of the following techniques to employ the interactive mode for the Rational Rhapsody command-line interface:

- ◆ Adding the `-interactive` switch in the command line
- ◆ Executing RhapsodyCL with no commands

For every command the user enters in interactive mode, RhapsodyCL performs the following actions:

1. Executes the command.
2. Wait for more commands from the user.
3. Stop when an `exit` command is received.

Note

If any commands exist in the command line when the `-interactive` switch is entered, the existing commands are executed first, and then RhapsodyCL enters interactive mode.

Socket mode

In this socket mode, RhapsodyCL listens on a socket port (supplied by the user), and any commands that arrive on that socket are executed immediately. RhapsodyCL stops only when it receives an `exit` command. To start the Rational Rhapsody command-line interface in socket mode, enter this command:

```
RhapsodyCL.exe -socket <Socket_Port>
```

The `<Socket_Port>` is the number of the port that the RhapsodyCL listens to for commands.

Note

If any commands exist in the command line when the `-socket <Socket_Port>` switch is entered, the existing commands are executed first, and then RhapsodyCL enters socket mode.

Command-line syntax

The syntax for using command-line options is the same for both Rational Rhapsody and RhapsodyCL.

The options are in the forms of switches and commands, and the syntax is slightly different for the two groups, as described in the following sections.

Note

Any path names within these commands should not contain spaces. If spaces must be included in a path, enclose the entire path in quotation marks to direct the command to the correct location.

Switches

For switches, the general syntax is as follows:

```
-switchName=parameter
```

Example

```
Rhapsody.exe -lang=cpp ...
```

Commands

For commands, the general syntax is as follows:

```
-cmd=commandName parameter
```

Example

```
Rhapsody.exe -cmd=open modelName.rpy -cmd=generate
```

For both switches and commands, parameters are separated from the command name or previous parameter by a space. No quotation marks are used.

Switches and commands are not case-sensitive but parameters are.

Note

In general, the switches refer to global configuration settings such as language, while the commands represent common actions in Rational Rhapsody such as open or generate.

Order of commands

All commands must be issued in a logical order. For example, since you must open a project before you can modify and save it, the open command must precede the save command in the command-line.

There is no significance to the order of parameters.

Include commands in a script file

The `-f` switch can be used to call a script file consisting of a number of commands. Within a script file, there is no need for the “-cmd” before the command name. Comment lines in script files begin with a pound sign (#).

Sample Script File

```
# This is a sample script file
setlog d:\log.txt
open d:\rhapsody\samples\Dishwasher dishwasher.rpy
generate EXE gui
save
make
```

Calling a Script File

```
rhapsody -f script.txt
```

Exit after use of command-line options

For Rhapsody.exe, you must close Rational Rhapsody after using the options on the command line in order to close the process, for example:

```
C:\> rhapsody -f script.txt -cmd=exit
```

With RhapsodyCL, however, this is not necessary. RhapsodyCL exits as soon as it has finished carrying out the specified commands.

Note

For Rhapsody.exe (but not RhapsodyCL.exe) make is an asynchronous command and should be the last command included in a script. You therefore cannot follow a make command with an exit command to close your project and exit Rational Rhapsody. If you do so, the make process will end prematurely.

Return codes

The following return codes are used for command-line options for both RhapsodyCL.exe and Rhapsody.exe:

- ◆ 0: success, no errors occurred
- ◆ 100: failed to open the project file
- ◆ 101: license not found
- ◆ 102: code generation failed
- ◆ 103: failed to load the project
- ◆ 104: failed to create or write to the code generation folder
- ◆ 105: errors were found in check model
- ◆ 106: unresolved elements in scope
- ◆ 107: error in the name of the component or configuration specified
- ◆ 108: build failed

Examples

The following examples show command-line usage with Rational Rhapsody.

```
C:\> rhapsody d:\rhapsody\samples\Dishwasher\Dishwasher.rpy -cmd=setlog  
d:\log.txt -cmd=generate EXE gui -cmd=save -cmd=make
```

This sample command line performs the following actions:

1. Starts Rational Rhapsody.
2. Opens the Dishwasher sample.
3. Directs the output to the file d:\log.txt.
4. Generates code for an executable component using the gui configuration.
5. Saves the project.
6. Builds the component.

The above example specifies the project to open as a parameter immediately following “rhapsody”. You can perform the same action using the -cmd=open command.

```
C:\> rhapsody -cmd=setlog d:\log.txt -cmd=open  
d:\rhapsody\samples\Dishwasher\Dishwasher.rpy -cmp EXE -cfg gui -cmd=generate  
-cmd=save -cmd=make
```

The following example illustrates the use of RhapsodyCL.

```
RhapsodyCL.exe -lang=cpp -cmd=open  
d:\rhapsody\samples\Dishwasher\Dishwasher.rpy -cmd=generate
```

Command-line switches

The switches that can be used on the command line with Rational Rhapsody are listed here. Unless otherwise noted, switches can be used with both Rhapsody.exe and RhapsodyCL.exe.

-animport=<Number>

(cannot be used with RhapsodyCL.exe)

Instructs Rational Rhapsody to use an animation port other than the one defined in the `rhapsody.ini` file. Using this option allows you to use animation in a number of Rational Rhapsody instances simultaneously. (See [Running on a Remote Target](#) for more details.)

-architect

Runs the architect version of Rational Rhapsody.

-dev_ed (default)

Runs the developer version of Rational Rhapsody (this is also the default value if a specific version is not specified).

-f

Runs the script provided as a parameter.

-hiddenui

(cannot be used with RhapsodyCL.exe)

Hides the Rational Rhapsody user interface. Can be used for tasks such as generating code.

Note: This switch predated RhapsodyCL. For tasks such as code generation, use RhapsodyCL rather than running the full version of Rational Rhapsody with the `-hiddenui` switch.

-interactive

Switches to a “shell mode” using a prompt to enter commands. See [Interactive mode](#) for more information.

-lang=<language>

Specifies the code language.

-noanimation

(cannot be used with RhapsodyCL.exe)

Disables animation by not attempting to open the TCP/IP animation port. This is useful for running more than one Rational Rhapsody instance without having to deal with the modal window that pops up when the animation port is unavailable.

-nodiagnostics

(cannot be used with RhapsodyCL.exe)

Loads the specified Rational Rhapsody model without the diagrams it contains.

-profile=<profile name>

Starts Rational Rhapsody with the specified profile.

-root=<pathname>

Specifies the root directory of the Rational Rhapsody installation.

-socket <Socket_Port>

RhapsodyCL listens on the socket port, and commands that arrive on that socket are executed immediately. See [Socket mode](#) for more information.

-system_architect

Runs the System Architect edition of Rational Rhapsody.

-system_designer

Run the Designer for Systems Engineers edition of Rational Rhapsody.

-verbose

Use this switch when you want RhapsodyCL to notify a user about a wrong syntax or unsupported commands.

Command-line commands

Unless otherwise noted, commands can be used both with Rhapsody.exe and RhapsodyCL.exe. In general, the following types of commands cannot be used with RhapsodyCL: diagram commands, configuration management commands, commands for running macros.

Note also that if you try to use a non-supported command with RhapsodyCL, the following actions will happen depending on if you have set the -verbose switch:

- ♦ If you have set the switch, RhapsodyCL will notify the user and ignore the command.
- ♦ If you have not set the switch, RhapsodyCL will simply ignore the command without any notification to the user.

Note: When making changes to projects under source control, check out the project before running RhapsodyCL.

-cmd=addtomodel <file location> <withdescendants|withoutdescendants>

Adds to the current model from the specified file location. The default value is <withoutdescendants>.

-cmd=arccheckout <file name> <label/revision> <locked|unlocked> <recursive|nonrecursive>

(cannot be used with RhapsodyCL.exe)

Checks out a file from the archive.

If you do not want to specify a <label/revision>, use NULL.

-cmd=call <plugin> <parameters for plug in>

(cannot be used with RhapsodyCL.exe)

Calls one of the Rational Rhapsody plug-ins and forwards the provided parameters to the plug-in.

In contrast to all the other commands, the parameters for this command are provided as a single string enclosed in quotation marks. The first parameter in the string should specify the plug-in that is being called. The remainder of the string contains the parameters that should be sent to the plug-in.

The following examples show the using of this command to run Test Conductor.

- ♦ -cmd=call "rtc run <listname>" will execute every test included in the batchlist with the name <listname>
- ♦ -cmd=call "rtc run all" will execute all the test defined in the TC.

- ◆ -cmd=call "rtc run <testpath>" will execute only the test which is in the path <testpath>

For example, Rhapsody D:\RhapsodyModels\Pbx\PBX.rpy -cmd=call "rtc run all"

-cmd=checkin <unit name> <label/revision> <locked|unlocked> <recursive|nonrecursive> <description> (cannot be used with RhapsodyCL.exe)

Checks in a unit to an archive. If you do not want to specify a <label/revision>, use NULL.

For example, -cmd=checkin p1.sbs NULL locked recursive "my description"

-cmd=checkmodel

Starts a Check Model operation.

Set the current configuration before issuing this command.

-cmd=checkout <unit name> <label/revision> <locked|unlocked> <recursive|nonrecursive> (cannot be used with RhapsodyCL.exe)

Checks out a unit from the archive. If you do not want to specify a <label/revision>, use NULL.

-cmd=close <NoSave>

Closes the open Rational Rhapsody model. By default, Rational Rhapsody will automatically save any changes you have made to the model before closing. If you do not want Rational Rhapsody to save changes upon closing, use the NoSave parameter.

-cmd=closediagram <diagram type><diagram name>

(cannot be used with RhapsodyCL.exe)

Closes the specified diagram.

The first parameter specifies the type of diagram. This parameter can take one of the following values: omd, ucd, msc, collaboration, component.

The second parameter is the name of the diagram in the model.

Connecttoarc <archive location>

(cannot be used with RhapsodyCL.exe)

Connects to an archive. <archive location> includes the full path.

-cmd=creatediagram <diagram type><diagram name>

(cannot be used with RhapsodyCL.exe)

Creates a new diagram.

The first parameter specifies the type of diagram. This parameter can take one of the following values: omd, ucd, msc, collaboration, component.

The second parameter is the name of the diagram in the model.

-cmd=exit

Closes the project and exits Rational Rhapsody.

-cmd=forceroundtrip

Performs a roundtrip regardless of the timestamps of the files.

-cmd=generate <component> <configuration>

Generates code for the specified component and configuration.

<component> and <configuration> are optional parameters. If not specified, the active component and configuration are used. Like the generate option in the GUI, this only generates code for modified elements. To regenerate all code, use the -regenerate command.

For example, -cmd=generate EXE Acme

If you want to generate code for more than one component, or for more than one configuration for a given component, you must repeat the generate command for each component/configuration combination, for example:

```
-cmd=generate compA cfg1 -cmd=generate compA cfg2 -cmd=generate compB cfg1
```

If you want to generate code for a nested component, use the syntax outerComponent::innerComponent, for example:

```
-cmd=generate def::abc DefaultConfig
```

Note: This command should not be used with RhapsodyCL.exe if you are using “customized code generation” or if you are generating code for the INTEGRITY operating system. Use the command with Rhapsody.exe instead.

-cmd=gmr

Performs generate/make/run.

-cmd=import

Imports classes according to the reverse engineering settings stored in the current configuration. This is equivalent to selecting the Rational Rhapsody command **Tools > Reverse Engineering**.

-cmd=make

Builds the application, using the current configuration.

Make is an asynchronous command and should be the last of all commands in a script.

Because exit is a synchronous command, you cannot follow a make command with an exit (to close your project and exit Rational Rhapsody); doing so will cause the make to cease prematurely.

If you plan to run the application right after the make, use -syncmake instead of -make. This waits for the make to complete before running any additional commands.

-cmd=new <project location> <project name>

Creates a new project in the specified location and assigns it the specified name.

-cmd=open <project name>

Opens the specified project. (RhapsodyCL.exe can only open projects. Rhapsody.exe can open units as well.)

-cmd=opendiagram <diagram type><diagram name>

(cannot be used with RhapsodyCL.exe)

Opens the specified diagram.

The first parameter specifies the type of diagram. This parameter can take one of the following values: omd, ucd, msc, collaboration, component.

The second parameter is the name of the diagram in the model.

-cmd=populatediagram <diagram type><diagram name>

(cannot be used with RhapsodyCL.exe)

Populates the specified diagram.

The first parameter specifies the type of diagram. This parameter can take one of the following values: omd, ucd, msc, collaboration, component.

The second parameter is the name of the diagram in the model.

-cmd=printcurrentdiagram

(cannot be used with RhapsodyCL.exe)

Prints the open diagram.

-cmd=regenerate <component> <configuration>

Generates code for the specified component and configuration.

<component> and <configuration> are optional parameters. If not specified, the active component and configuration are used. Like the regenerate option in the GUI, this regenerates all the code, not just the code for changed elements.

If you want to generate code for more than one component, or for more than one configuration for a given component, you must repeat the regenerate command for each component/configuration combination, for example:

```
-cmd=regenerate compA cfg1 -cmd=regenerate compA cfg2 -cmd=regenerate compB  
cfg1
```

If you want to regenerate code for a nested component, use the syntax outerComponent::innerComponent, for example:

```
-cmd=regenerate def::abc DefaultConfig
```

Note: This command should not be used with RhapsodyCL.exe if you are using “customized code generation” or if you are generating code for the INTEGRITY operating system. Use the command with Rhapsody.exe instead.

-cmd=report <format> <name + location>

Generates a report.

<format> is the report format (RTF or ASCII). The file extension is added automatically (.rtf for RTF and .txt for ASCII).

<name + location> specifies the name and location of the report. These parameters are optional.

If you do not specify a name, the default file name is used (RhapsodyRep.rtf).

If you do not specify a location, the default location is used (the project directory).

Set the current configuration before issuing this command.

For example, `-cmd=report RTF myReport`

For RhapsodyCL, the report command uses the Rational Rhapsody internal reporter and does not extract diagrams.

-cmd=roundtrip

Roundtrips code changes back into the model.

Set the current configuration before issuing this command.

-cmd=runexternalprogram

Runs the specified external program.

(with RhapsodyCL.exe, cannot be used to run COM-based programs.)

-cmd=runvbamacro <module name> <macro_name>

(cannot be used with RhapsodyCL.exe)

Runs the specified VBA macro outside an active project. The VBA script must already exist within a Rational Rhapsody model and be compiled so the file <model>.vba exists.

You can copy a .vba file into your project directory (where the .rpy file is located). For example, if you have the project file abc.rpy, copy your macro .vba file as abc.vba then use the runvbmacro command to run the macro.

In VBA, if you do not specify a module name, the default is module1. You cannot pass parameters to the module.

For example, rhapsody -cmd=open d:\rhapsody\models\hhs.rpy -cmd-runvbamacro module1 first

-cmd=save

Saves the open project. Can be used after making changes like roundtrip, reverse engineering.

-cmd=saves <project name>

Saves the project to a specified location. <project name> can include the path.

-cmd=setcomponent <active component name>

Sets the active component.

If you want to make a nested component the active component, use the syntax outerComponent::innerComponent, for example:

-cmd=setcomponent def::abc

-cmd=setconfiguration <active configuration name>

Sets the active configuration.

For example, -cmd=setconfiguration AcmeDebug

-cmd=setlog <log file>

Redirects the output normally sent to the output window to the specified log file. If the parameter does not specify the path, the log file is put in the "current" Rational Rhapsody directory. If a log file is specified, output is not sent to the standard output.

-cmd=setomroot <alternative OMROOT>

Sets the variable OMROOT to a new location. This variable specifies the root directory of the Rational Rhapsody installation.

For this command to take effect, this must be the first option specified in the command line.

-cmd=syncmake

Builds the application using the current configuration.

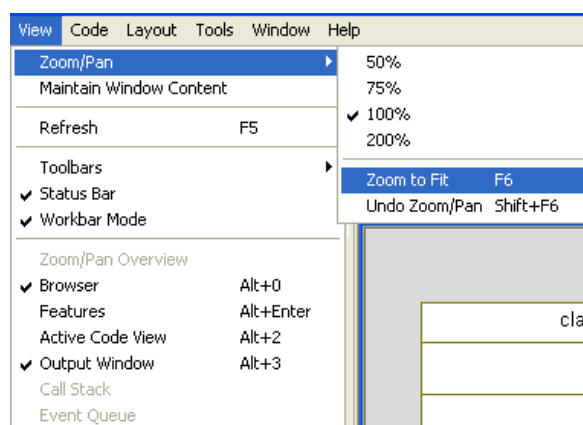
As opposed to the make command, the syncmake command will wait until the make has completed before running any additional commands. So if you plan to run the application right after building it, use syncmake instead of make.

Rational Rhapsody shortcuts

Rational Rhapsody supports the following kinds of keyboard interaction: accelerator keys, mnemonics, modifiers, and standard Windows shortcuts.

Accelerator keys

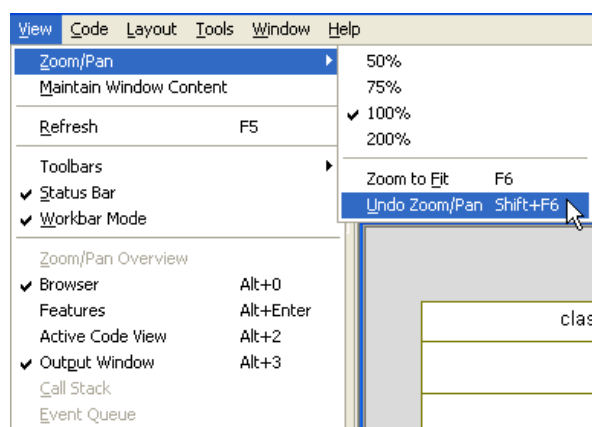
An *accelerator key* is a keyboard key (or combination keys) designated to achieve a specific action. For example, the **F6** keyboard key is an accelerator for **Zoom to fit** of a diagram, whereas **Ctrl+A** is an accelerator key for **Select All**. In most cases, the menu option that serves the same purpose as the accelerator lists the corresponding accelerator, as in this example:



Mnemonics

Mnemonics are small indicators marked as an underline under a letter in a menu name or on a button name. This indicator means that clicking Alt and the designated mnemonic letter will result in activating this menu.

For example, the following figure shows that you can achieve the **Zoom to Fit** functionality not only with the accelerator key **F6**, but also using mnemonics by clicking **Alt+{V, Z, F}**: **Alt+V** opens the View menu, **Alt+Z** opens the Zoom menu, and **Alt+F** executes the **Zoom to Fit** command.



This document does not describe the complete set of mnemonics because it is clearly visible on the menus.

Note

Sometimes the underlining is not visible in the menus. If this occurs, try pressing the Alt key until they are displayed. If you still cannot see them, follow the instructions in [Changing settings to show the mnemonic underlining](#).

Keyboard modifiers

A *modifier* is a keyboard key applied to a command to slightly modify its behavior or meaning. For example, when using your mouse for resizing a shape in a Rational Rhapsody graphic editor, you can use the Alt key to change the operation behavior from “resize” to “resize without contained.”

Standard Windows keyboard interaction

Windows applications have an extensive list of standard keyboard shortcuts to common interactions. For example, **Ctrl+Tab** toggles through the open windows within an application. Using this keyboard shortcut in Rational Rhapsody will navigate through the open diagrams and code editors that are currently open.

Rational Rhapsody accelerator keys

Accelerator keyboard keys can be further broken down into three types:

- ◆ **Application accelerators** activate menu commands.
- ◆ **Accelerators and modifiers within diagrams** assist with drawing activities.
- ◆ **Accelerators in the code editor** assist in coding activities.

Application accelerators

Action	Shortcut
Accelerators for mapping and navigation	
Locate in Browser	Ctrl+L
Search in Model	Ctrl+F
Show References	Ctrl+R
Animation	
Go	F4
Go Event	F10
Quit animation	Shift+F5
Browser navigation	
Expand all	Numeric block *
Navigate	Numeric lock keys
Code	
Build	F7
Generate	Ctrl+F7
Run	Ctrl+F5
Stop Make Execution	Ctrl+Break
Project	

Action	Shortcut
New Project	Ctrl+N
Open Project	Ctrl+O
Print	Ctrl+P
Redo	Ctrl+Y
Save Project	Ctrl+S Although this usually saves the model, if you are focused on code (for example, using Edit Code), this shortcut saves the file, not the model.
Undo	Ctrl+Z
VBA	
Macros	Alt+F8
Visual Basic Editor	Alt+F11
Window management	
Arrange Options	Ctrl+W
Show/Hide Active Code View	Alt+2
Show/Hide Browser	Alt+0 (zero)
Show/Hide Features	Alt+Enter
Show/Hide Output Window	Alt+3
Help	
Help	F1

Accelerators and modifier usage in diagrams

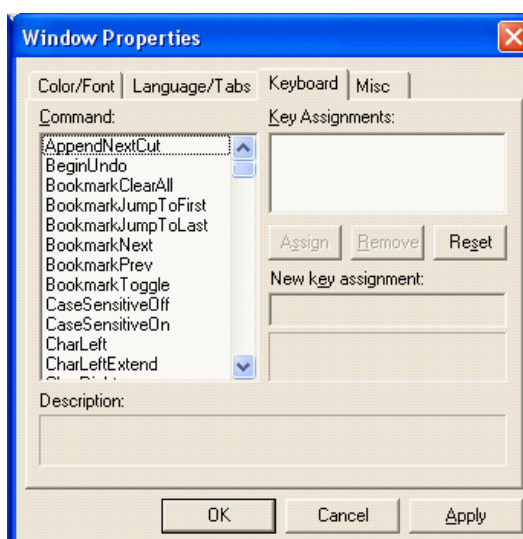
Action	Shortcut
Add a new item to a list compartment (for example, the attributes compartment of a class box).	Insert Note that this shortcut works only if you already have a list. It does not work for the first item in a list.
Change the selection anchor	Select + Ctrl
Copy	Ctrl+C; Ctrl+Drag You can also use this shortcut in the Rational Rhapsody browser.
Create more space in an SD (assuming you are using a mouse)	Click+Shift and drag the mouse to display a dashed, horizontal bar. Move the bar down to create more space, or move it up to eliminate unnecessary space.
Create or resize elements (by mouse dragging) with a symmetrical shape	Shift (while dragging) Note that using the corner anchor and Shift creates a "lock aspect ratio" sizing.
Create straight lines and arrows that are parallel to the axis.	Use the Ctrl key while drawing the lines.
Cut	Ctrl+X
Delete from Model	Ctrl+Delete
Insert a new user point in a line or arrow.	Ctrl+Click Note that this does not apply to rectilinear lines or to SD messages).
Paste	Ctrl+V
IntelliVisor	Ctrl+Spacebar
Move shape to the ARROW direction ("nudge").	Ctrl +ARROW (where ARROW could be the up, down, left, or right arrow key)
Move shape to the ARROW direction without its contained elements ("nudge").	Ctrl+Alt +ARROW (where ARROW could be the up, down, left, or right arrow key)
Refresh	F5 Although this usually refreshes the model, if you are focused on code (for example, using Edit Code), this shortcut performs a roundtrip.
Remove from View.	Delete Note that sometimes the Delete key deletes the element from the model, depending on the diagram context (for example, statecharts).
Removes the selected (clicked) element from the selection.	Select + Shift

Action	Shortcut
Resize box to fit contained.	Ctrl+E
Resize without contained elements (assuming you are using a mouse).	Use the Alt key while stretching the shape.
Scale from the center of the element (instead of stretching).	Scale + Ctrl
Zoom to Fit.	F6
Select All.	Ctrl+A
Select next shape (by proximity).	Ctrl+Alt+N
Undo Zoom.	Shift+F6

Code editor accelerators

When you are using the built-in Rational Rhapsody code editor, you can use not only the predefined accelerators, but an extended set of accelerator keys.

Open the Properties window for the code editor and click the **Keyboard** tab to view the complete list of commands supported by accelerator keys, and to extend this list.

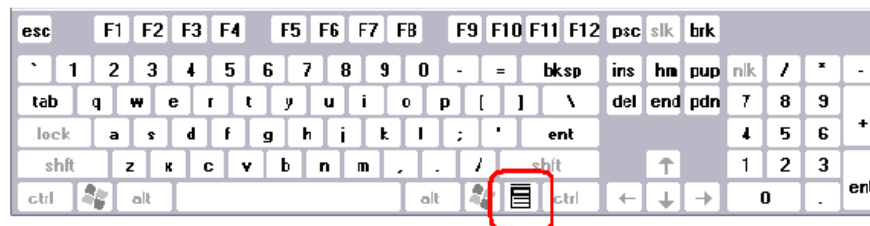


Useful Rational Rhapsody Windows shortcuts

The following table lists some of the more common and useful Windows keyboard shortcuts available in Rational Rhapsody.

Action	Shortcut
Close the currently active window.	Ctrl+F4 If you hold down Ctrl+F4 , all the Rational Rhapsody windows close in succession. Note: This method will not work if you have a diagram that needs to be saved (for example, an animated SD).
Enable or disable check boxes.	Space key
Get to the Windows menu.	Alt+Space
Start IntelliVisor when editing code or names of graphic elements.	Ctrl+Space
Navigate between open diagrams.	Ctrl+Shift+Tab
Navigate between boxes when in a window.	Tab To do the same in reverse order, use Shift+Tab .
Navigate between tabs when in a window (for example, the Features window).	Ctrl+Tab To do the same in reverse order, use Ctrl+Shift+Tab .
Navigate to items in lists (such as the list of element types in the Search/Replace window) or in the browser.	Type its name on the keyboard.
Navigate within the browser.	Use the up and down keys to move between nodes. Use the left and right arrow keys to expand or collapse tree nodes.
Navigate within the Properties tab for any item.	Up and down arrow keys
Toggle between all open windows inside the application (diagrams, code windows, and so on).	Ctrl+Tab To do the same in reverse order, use Ctrl+Shift+Tab .

In addition, whenever an item is selected (such as a node in the browser or a class in a graphic editor), you can use the standard Windows keyboard key designated to activate the menu of the selected item. The following figure highlights this button (but the location might vary, depending on your keyboard).

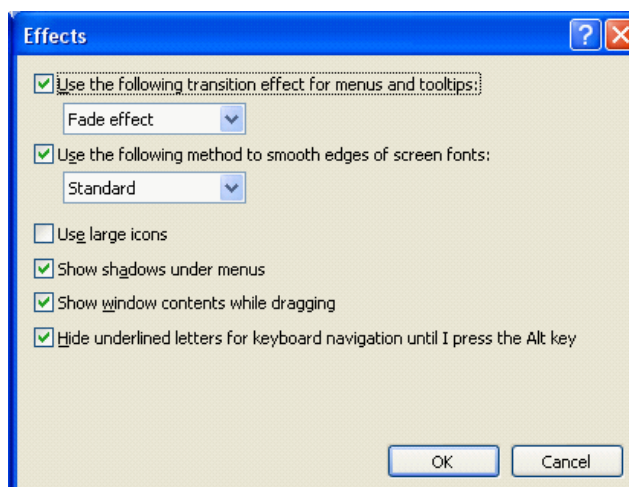


The resultant menu can also be used with mnemonics.

Changing settings to show the mnemonic underlining

To expose mnemonic underlining in Rational Rhapsody menus:

1. On your desktop, right-click and select **Properties**. The Display Properties window opens.
2. On the **Appearance** tab, click **Effects**. The Effects window opens, as shown in the following figure.



3. Clear the last check box to show underlined letters for keyboard navigation.
4. Click **OK** twice.

Technical support

All IBM Rational Rhapsody customers receive support from IBM Rational Software Support and resources.

Contacting IBM Rational Software Support

If the self-help resources have not provided a resolution to your problem, you can contact IBM[®] Rational[®] Software Support for assistance in resolving product issues.

Prerequisites

To submit your problem to IBM Rational Software Support, you must have an active Passport Advantage[®] software maintenance agreement. Passport Advantage is the IBM comprehensive software licensing and software maintenance (product upgrades and technical support) offering. You can enroll online in Passport Advantage from <http://www.ibm.com/software/lotus/passportadvantage/howtoenroll.html>.

- ♦ To learn more about Passport Advantage, visit the Passport Advantage FAQs at http://www.ibm.com/software/lotus/passportadvantage/brochures_faqs_quickguides.html.
- ♦ For further assistance, contact your IBM representative.

To submit your problem online (from the IBM Web site) to IBM Rational Software Support, you must additionally:

- ♦ Be a registered user on the IBM Rational Software Support Web site. For details about registering, go to <http://www.ibm.com/software/support/>.
- ♦ Be listed as an authorized caller in the service request tool.

Contacting Support

To contact IBM Rational Software Support:

1. Locate your **ICN** (IBM customer number). It is required for support requests.
2. Determine the business impact of your problem. When you report a problem to IBM, you are asked to supply a severity level. Therefore, you need to understand and assess the business impact of the problem that you are reporting.

Use the following table to determine the severity level.

Severity	Descriptions
1	The problem has a <i>critical</i> business impact: You are unable to use the program, resulting in a critical impact on operations. This condition requires an immediate solution.
2	This problem has a <i>significant</i> business impact: The program is usable, but it is severely limited.
3	The problem has <i>some</i> business impact: The program is usable, but less significant features (not critical to operations) are unavailable.
4	The problem has <i>minimal</i> business impact: The problem causes little impact on operations or a reasonable circumvention to the problem was implemented.

3. Describe your problem and gather background information. When describing a problem to IBM, be as specific as possible. Include all relevant background information so that IBM Rational Software Support specialists can help you solve the problem efficiently. To save time, know the answers to these questions:
 - ◆ What software versions were you running when the problem occurred?

To determine the exact product name and version, use the option applicable to you:

 - Start the IBM Installation Manager and choose **File > View Installed Packages**. Expand a package group and select a package to see the package name and version number.
 - Start your product, and choose **Help > About** to see the offering name and version number.
 - ◆ What is your operating system and version number (including any service packs or patches)?
 - ◆ Do you have logs, traces, and messages that are related to the problem symptoms?
 - ◆ Can you recreate the problem? If so, what steps do you perform to recreate the problem?

- ♦ Did you make any changes to the system? For example, did you make changes to the hardware, operating system, networking software, or other system components?
 - ♦ Are you currently using a workaround for the problem? If so, be prepared to describe the workaround when you report the problem.
4. Submit your problem to IBM Rational Software Support. You can submit your problem to IBM Rational Software Support in the following ways:
- ♦ **From the Support Web site:** Go to the IBM Rational Software Support Web site at <https://www.ibm.com/software/rational/support/> and in the Rational support task navigator, click **Open Service Request**. Select the electronic problem reporting tool, and open a Problem Management Record (PMR), describing the problem accurately in your own words.
 - ♦ **Request assistance through e-mail:** send the e-mail to the support address for your region:
 - sw_support_emea@nl.ibm.com
 - sw_support@us.ibm.com
 - sw_support_ap@au1.ibm.com
 - ♦ For more information about opening a service request, go to <http://www.ibm.com/software/support/help.html>
 - ♦ You can also open an online service request using the IBM Support Assistant. For more information, go to <http://www.ibm.com/software/support/isa/faq.html>.
 - ♦ **By phone:** For the phone number to call in your country or region, go to the IBM directory of worldwide contacts at <http://www.ibm.com/planetwide/> and click the name of your country or geographic region.
 - ♦ **Through your IBM Representative:** If you cannot access IBM Rational Software Support online or by phone, contact your IBM Representative. If necessary, your IBM Representative can open a service request for you. You can find complete contact information for each country at <http://www.ibm.com/planetwide/>.

If the problem you submit is for a software defect or for missing or inaccurate documentation, IBM Rational Software Support creates an Authorized Program Analysis Report (APAR). The APAR describes the problem in detail. Whenever possible, IBM Rational Software Support provides a workaround that you can implement until the APAR is resolved and a fix is delivered. IBM publishes resolved APARs on the IBM Rational Software Support Web site daily, so that other users who experience the same problem can benefit from the same resolution.

About Rational Rhapsody

Select **Help > About Rhapsody** (or **Help > About DiffMerge** if you have launched the Rational Rhapsody DiffMerge tool outside Rational Rhapsody) to display this information about your version of Rational Rhapsody (or the Rational Rhapsody DiffMerge tool):

- ◆ Product release number
- ◆ Build number
- ◆ Serial number
- ◆ Contact information
- ◆ Copyright information

License Details

When you have the About Rhapsody window open, you can click the **License** button to open the License Details window. This box lists the information you might need when calling technical support or upgrading your software:

- ◆ Host details for your machine including the Host Name and Host ID
- ◆ Software license information

Notice that you can resize the width of the License Details window.

Reporting Rational Rhapsody Problems from the Software

When Rational Rhapsody is running, you might want to use the problem reporting facility from the Rational Rhapsody Help menu.

To send an automated problem report:

1. In Rational Rhapsody, choose **Help > Generate Support Request** to open the Generate Support Request window.
2. If requested, enter your **ICN** (IBM customer number), select your **Geographic Region** and click **OK** to access the Generate Support Request form.
3. Review the **Rhapsody Information** and **System Information** areas to verify the accuracy of the automatically entered data.
4. From the **Impact** list, select the severity of the problem.
5. In the **Summary** box, summarize the problem.
6. In the **Problem** box, type a detailed description of the problem.
7. If possible, take a snapshot of the problem and attach it to the problem report. Click the **Rhapsody Window Snapshot** button or **Screen Snapshot** button, whichever is applicable, and select the snapshot file from wherever you have it on your machine.
8. If possible, add the model, active component, files, and/or a video capture by using the buttons in the **Attachment Information** area.
9. Include an item description for each item in the **Attachment Information** area, if needed.
10. Click **Preview and Send** to submit the report.

The problem report is recorded in the Rational Rhapsody case tracking system and put into a queue to be assigned to a support representative. This representative works with you to be certain that your problem is solved.

Note

If your Rational Rhapsody system crashes, it displays a message asking if you want to send a problem report to technical support about this crash. If you select to send the report, the system displays the same online form that is available from **Help > Generate Support Request**. However, this form contains information about the crash condition in addition the information that is usually filled in describing your system. Add any more information that you can to help the support staff identify the problem and then click **Preview and Send** to submit the report.

Rational Rhapsody glossary

abstract class

A class that cannot be directly instantiated, but whose descendants can have instances. Contrast with [concrete class](#).

abstract operation

An operation defined, but not implemented, by an abstract class. The operation must be implemented by all concrete descendant classes.

abstraction

Selecting the essential or common characteristics of an entity that distinguish it from all other kinds of entities. An abstraction defines a boundary relative to the perspective of the viewer.

action

The specification of an executable statement that forms an abstraction of a computational procedure. An action typically results in a change in the state of the system, and can be realized by sending a message to an object or modifying a link or value of an attribute.

action sequence

An expression that resolves to a sequence of actions.

action state

A state that represents the execution of an atomic action, typically the invocation of an operation.

activation

The execution of an action.

active class

A class whose instances are active objects.

active concurrency

The system runs in a distributed environment with many threads. Each active object runs on its own thread. Active objects are also known as tasks.

active object

An instance of an active class. An active object owns its own thread and can initiate control activity.

In Rational Rhapsody, active objects are graphically portrayed with thicker borders.

activity

An operation in dynamic modeling that takes time to complete. Activities are associated with states and represent real-world accomplishments.

activity diagram

A special case of a state machine used to model processes involving one or more classifiers that involve behavior that is not event-driven. Compare to [statechart](#).

activity final

Signals an exit from the process specified by the [activity diagram](#) or [flow chart](#).

actor

A coherent set of roles that users of use cases play when interacting with these use cases. An actor has one role for each use case with which it communicates.

An actor is an external object that interacts with a use case of the system. In use case diagrams (UCDs), actors are drawn as stick figures and can populate both use case diagrams and object model diagrams (OMDs).

actual parameter

A synonym for argument.

aggregate

A class that represents the “whole” in an aggregation relationship.

aggregation

A special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part.

The [Unified Modeling Language \(UML\)](#) symbol for an aggregation relationship is a line with a hollow diamond at the end attached to the aggregate class.

analysis

The software development activity for studying and formulating a model of a problem domain. Analysis focuses on what is to be done; design focuses on how to do it.

analysis time

Refers to something that occurs during an analysis phase of the software development process. See also [design time](#) and [modeling time](#).

ancestor class

A class that is a direct or indirect superclass of a given class.

and state

An orthogonal state.

animation

The act of executing an animated model. During an animation session, Rational Rhapsody highlights the current states of execution using animated diagrams and views.

Animation is not the same as simulation. The Rational Rhapsody animator actually runs the real application on the host machine or, if wanted, on the target machine. This gives you a better idea of the system's actual behavior than merely simulating it.

API

The Rational Rhapsody Application Program Interface (API) allows you to write applications that access and manipulate Rational Rhapsody model elements. It facilitates reading, changing, adding to, and deleting from all model elements that are available in the Rational Rhapsody [browser](#).

architecture

The organizational structure and associated behavior of a system. An architecture can be recursively decomposed into parts that interact through interfaces, relationships that connect parts, and constraints for assembling parts. Parts that interact through interfaces include classes, components, and subsystems.

architecture framework

An architecture framework is a specification of how to organize and present an enterprise architecture. It provides a means to present and analyze the enterprises problems. It does not generally tell you how to do something. Architecture frameworks tend to consist of a standard set of viewpoints that represent different aspects of an organization's business as it relates to a particular objective. In the context of MODAF, this implies a systems of systems approach as the analysis is complex and wide-ranging.

archive

A repository grouping that a configuration management (CM) tool creates to keep track of different versions of a project's configuration items. An archive can be a file or directory, depending on the requirements of the CM system.

argument

A binding for a parameter that resolves to a run-time instance. A synonym for argument is actual parameter. Compare to [parameter](#).

artifact

A piece of information used or produced by a software development process. An artifact can be a model, a description, or software. Synonym: *product*.

association

Defines a semantic relationship between two or more classifiers that specify connections among their instances. It represents a set of connections between the objects (or users).

An association has at least two ends, each of which is connected to an object. The same object can be connected to both ends of an association.

An association can be bidirectional, in which the connected objects know about each other, or directed, in which only one of the objects knows of the other.

An association defines a relationship among instances of two or more classes describing a group of links with common structure and common semantics.

association class

A model element that has both association and class properties. An association class can be seen as an association that also has class properties, or as a class that also has association properties.

association end

The endpoint of an association, which connects the association to a classifier.

attribute

In Rational Rhapsody, an attribute is a feature within a classifier that describes a range of values that instances of the classifier can hold. An attribute consists of three parts:

- ◆ A data member
- ◆ An accessor (get) operation for the data member
- ◆ A mutator (set) operation for the data member

Attributes are displayed in the object box using the following format:

```
<attribute_name>:<type>
```

Contrast with [part](#).

In ReporterPLUS, attributes are the items listed in the [attribute view](#). Attributes represent the pieces of data that can be extracted from a model. To add model text or diagrams to your document, you add attributes to the Text tab. ReporterPLUS adds some attributes automatically when you drag elements to the template view.

Attribute breakpoint condition

Interrupts a running application when any of the object's member attributes changes value. Copies of all attribute values are stored as a reference when you set the breakpoint. When any value changes with respect to the reference, a break occurs. After the break, the latest values are again stored as a new reference.

attribute view

The upper, right pane in the ReporterPLUS window. Compare with [model view](#). See also [attribute](#), [template node view](#), and [template view](#).

automatic transition

An unlabeled transition in dynamic modeling that automatically fires when the activity associated with the source state is completed.

AUTOSAR

The AUTOSAR (AUTomotive Open System ARchitecture) standard provides development and design guidelines and diagrams to streamline and standardize component modeling in the automotive industry. The Rational Rhapsody AUTOSAR profiles define a new project to use these AUTOSAR standard-compliant diagrams:

- ◆ ECU diagram
- ◆ Internal Behavior diagram
- ◆ SW Component diagram
- ◆ System diagram
- ◆ Topology diagram

base-aware comparison

The Rational Rhapsody [DiffMerge tool](#) compares two versions of a unit with a baseline (common ancestor) version of the unit.

base class

A class from which other classes can inherit data and member functions.

batch transformation

A sequential input-to-output transformation in which inputs are supplied at the start and the goal is to compute an answer. There is no ongoing interaction with the outside world.

behavior

The observable effects of an operation or event, including its results.

behavioral feature

A dynamic feature of a model element, such as an operation or method.

behavioral inheritance

A mechanism by which more specific elements incorporate behavior of more general elements related by behavior.

behavioral model aspect

A model aspect that emphasizes the behavior of the instances in a system, including their methods, collaborations, and state histories.

binary association

An association between two classes. A special case of an [n-ary association](#).

binding

The creation of a model element from a template by supplying arguments for the parameters of the template.

black-box analysis

This type of system analysis defines the system structure and identifies the large-scale organizational pieces of the system. It can show the flow of information between system components and the interface definition through ports. In large systems, the components are often decomposed into functions or subsystems. Basically, it shows the system's interaction with the outside world. Using Rational Rhapsody, you can create activity diagrams, sequence diagrams, and statecharts to communicate this analysis. A [white-box analysis](#) shows a system's internal and external operations and relationships.

block

In Rational Rhapsody a block is represented as an [object](#). In [SysML](#), a block is the basic unit of structure and is represented as a [class](#). A systems design block shows the system hierarchy and system/component specifications.

boilerplate text

In ReporterPLUS, text that has been added to a template by typing in the **Text** tab. ReporterPLUS adds some boilerplate text automatically when you drag elements to the template view. Boilerplate text does not come from the model.

Boolean

An enumeration whose values are true and false.

Boolean expression

An expression that evaluates to a Boolean value.

breakpoint

A useful debugging tool. During animation, breakpoints enable you to inspect data values and the states of various objects in the system at the time of the break.

You can set a breakpoint on object's instance or on the object itself. If you set the breakpoint on an object, it sets the breakpoint on all the object's instances.

browser

Provides an overview of your entire model using an expandable tree structure.

call

An action state that launches an operation on a classifier.

call operation node

Represents a call to an operation of a classifier.

canceled timeout

Timeouts are set to wait for something to happen. If the something happens, the timeout is canceled. If it does not happen, the object resumes its operation, perhaps with an error recovery process.

For example, a telephone emits a dial tone while waiting for you to dial. If you dial, the dial tone is canceled. If you do not dial, the dial tone changes to a repeating beep. Canceled timeouts are labeled $\text{CanTm}(n)$, where n is the length of the time during which the timeout can be canceled.

cardinality

The number of elements in a set. Contrast with [multiplicity](#).

categories mode

Specifies that metatype nodes are displayed in the browser for all metatypes, such as classes, packages, and operations, in hierarchical arrangement by ownership.

For example, under the Objects metatype for a package, all objects belonging to that package are listed. Each object or object_type, in turn, can be expanded into categories for Attributes, Operations, and Relations belonging to that object or object type.

child

In a generalization relationship, the specialization of another element, the parent. See also [subclass](#), [subtype](#). Contrast with [parent](#).

class

In object-oriented languages such as C++ and Java, a class is a template for the creation of instances (objects) that share the same attributes, operations, methods, relationships, and semantics. A class can use a set of interfaces to specify collections of operations it provides to its environment. See also [interface](#).

In the Rational Rhapsody Developer for C implementation, classes are replaced by object types, which are generated into structures. Like classes, object types function as templates in the creation of objects of explicit type. Although object types differ fundamentally from classes, from the perspective of the Rational Rhapsody GUI, they are handled almost identically. Thus, for documentation purposes, any GUI functionality ascribed to classes applies almost equally as well to object types.

In addition to object types, Rational Rhapsody Developer for C supports objects of implicit type. All objects in C, whether objects of implicit or explicit type, are known as objects. Objects are similar to instances in the Rational Rhapsody C++ implementation, with the exception that they have a separate identity independent of object types.

For example, objects are listed in their own category under a package in the browser. Instances, by contrast, appear in the browser under the class that defines it, and only during execution of the model's application. Because an object is a permanent instance with class-like behavior, in many cases, the GUI functionality ascribed to C++ classes also applies to C objects.

class attribute

An attribute whose value is common to a class of objects, rather than a value peculiar to each instance.

class diagram

A diagram that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships.

class interface element

Operations, events, and event receptions.

class name

Identifies the class. If you do not explicitly assign a class to a package, Rational Rhapsody assigns the class to the default package of the diagram. To assign the class to a different package, use the format <package>: :<class> for the name. If this box is also an instance, use the format <instance>:<class> for the class name.

class template

Specifies individual classes constructed using parameterized types. When class templates are instantiated into template classes, types used in the class are provided as arguments.

The following example declares the class template `MyTemplateClass`:

```
template<class T> class MyTemplateClass {  
    T* data  
    public:  
        T& func()  
};
```

The class template `MyTemplateClass` can then be used to instantiate a template class and an object as shown by the following example:

```
MyTemplateClass<int> int_object
```

In this example, `MyTemplateClass<int>` is a template class in which the type `int` replaces `T` in the class template definition. It is then used to instantiate the object `int_object`.

Rational Rhapsody allows you to create or change an existing class into a class template or instantiate it into a template class in the Class window.

See also [template class](#).

classification

The assignment of an object to a classifier. See also [dynamic classification](#), [multiple classification](#), and [static classification](#).

classifier

A mechanism that describes behavioral and structural features. Classifiers include interfaces, classes, data types, and components.

cleanup operation

Cleans up an instance that is no longer needed. It replaces the concept of a destructor operation. Instead of cleaning up class instances, a cleanup operation cleans up C objects.

For example, an object can call a cleanup operation to free a dynamically allocated pointer. In sequence diagrams, a cleanup operation is represented as a dotted red arrow from the destroyer object to the object being destroyed. Cleanup lines can either be horizontal or point back to the originating object. Cleanup lines are not labeled.

client

A classifier that requests a service from another classifier. Contrast with [supplier](#).

CM

Acronym for [configuration management](#).

code-centric development

Software development environment that focuses on writing application code rather than using the [Model-driven Development \(MDD\)](#) method.

code frame

Refers to code generated from OMDs only, without the behavioral input of statecharts and activity diagrams.

collaboration

The specification of how an operation or classifier, such as a use case, is realized by a set of classifiers and associations playing specific roles used in a specific way. The term collaboration defines this interaction. See also [interaction](#).

collaboration diagram

A diagram that shows interactions organized around the structure of a model, using either classifiers and associations or instances and links. Unlike a sequence diagram, a collaboration diagram shows the relationships among the instances. Sequence diagrams and collaboration diagrams express similar information, but show it in different ways. See also [sequence diagram](#).

comment

An annotation attached to an element or a collection of elements. A note has no semantics. Contrast with [constraint](#).

compile time

Refers to something that occurs during the compilation of a software module. See also [modeling time](#), [run time](#).

component

A physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical piece of implementation of a system, including software code (source, binary, or executable) or equivalents such as scripts or command files.

The role of the component is important in the modeling of large systems that are comprised of several libraries and executables. For example, the Rational Rhapsody application itself is comprised of several dozen components including the graphic editors, browser, code generator, and animator, all provided in the form of a library.

component diagram

A diagram that shows the organizations and dependencies among components.

component file

Contains elements for a [component](#) and can be saved as a [unit](#). In Rational Rhapsody 7.2 or greater, this type of file is now called a [SourceArtifact](#).

composite aggregation

A synonym for composition.

composite class

A class related to one or more classes by a composition relationship.

composite object

An object that contains one or more other objects, typically by storing references to those objects in its instance variables.

composite state

A state that consists of either concurrent (orthogonal) substates or sequential (disjoint) substates. See also [subpackage](#).

composition

A form of aggregation association with strong ownership and coincident lifetime as part of the whole. Put another way, composition is a strong form of aggregation in which the lifetime of the whole determines that of its parts. Parts with non-fixed multiplicity can be created after the composite itself, but once created, they live and die with it (they share lifetimes). Such parts can also be explicitly removed before the death of the composite. Composition can be recursive. A synonym for composition is composite aggregation.

The UML symbol for a composition relationship is a line with a filled diamond at the end attached to the composite class. In Rational Rhapsody, you draw classes as boxes inside the composite class, rather than using relation lines.

If you want the component class to come into being and die with the composite class as a whole, use composition. If, however, you want the parts to have lifetimes of their own separate from that of the whole, use aggregation.

Note the following information:

- ◆ By definition, an object that contains another object is a reactive object.
- ◆ Composition in Rational Rhapsody Developer for C is a form of containment of one object by another. An object that contains another is said to be a *composite object*.

compound state

A state that contains other nested states.

concrete class

A class that can be directly instantiated. Contrast with [abstract class](#).

concurrency

The occurrence of two or more activities during the same time interval. The activities are said to be concurrent. Concurrency can be achieved by interleaving or simultaneously executing two or more threads.

concurrent

Two or more tasks, activities, or events are said to be concurrent when their executions overlap in time.

concurrent substate

A substate that can be held simultaneously with other substates contained in the same composite state. Contrast with [disjoint substate](#).

condition

In Rational Rhapsody, a condition is a Boolean function in dynamic modeling of object values valid over an interval of time.

In ReporterPLUS, a condition is a statement that limits the elements ReporterPLUS extracts from a model for an iteration.

condition mark

A hexagon located on an instance line in a sequence diagram. A condition mark indicates that the object is in a certain condition or state. The name of the condition often corresponds to a state name in the object's statechart.

configuration

Defines the construction of a built component. For example, the configuration determines the target environment of the component and whether it is instrumented. It specifies which checks Rational Rhapsody should perform before generating code, which link and compiler switches to use, any additional libraries, sources, and headers to include in the compilation, and which initial instances to create in the main program loop. The configuration also allows you to add custom initialization code to the `main()` function.

configuration item

A unit of collaboration that developers can exchange among themselves. See also [unit](#).

configuration management

The process of managing a set of classes and other resources selected for compilation including version control. This is usually managed by software system that requires “check in” and “check out” procedures for changes to any files managed in the system. Rational Clearcase and MKS Integrity are both configuration management (CM) systems.

constraint

A semantic condition or restriction. Certain constraints are predefined in the UML, whereas others are user-defined. Constraints are one of three extensibility mechanisms in the UML. See also [tagged value](#), [stereotype](#).

In general, UML constraints add semantic information to model elements, which represent requirements, invariants, and so on. Constraints can be specified in a natural language, constraint language (such as OCL), or programming language.

A constraint is a model element owned by some other model element. In Rational Rhapsody, only model elements that have specification windows can own constraints. These include packages, classifiers, operations, attributes, states, and transitions

Note that the object owning the constraint is not necessarily the object to which the constraint applies. By default, if there is no “applies to relation” (via a dependency), the constraint applies to the owner object. This is an optimization, because this would be the common, default case in which there is no need to apply a dependency relation.

A constraint applies to one or more model elements. A model element can have more than a single constraint applied to it.

construct

Previously created items, such as files and classes, that can be added to a model or design.

constructor

Called when an object is instantiated. An object can use a constructor to explicitly initialize object members or dynamically allocate space for member pointers.

In sequence diagrams, a constructor is represented as a dotted green arrow from the creator object or system border to the object being created. Constructor lines are horizontal.

Convert and copy constructors can have arguments.

In Rational Rhapsody Developer for C, the tasks of constructors are performed by initializers for the class-like objects and object_types.

container

Can be either of the following types:

- ◆ An instance that exists to contain other instances, and provides operations to access or iterate over its contents. (for example, arrays, lists, and sets).
- ◆ A component that exists to contain other components.

containment hierarchy

A namespace hierarchy consisting of model elements and the containment relationships that exist between them. A containment hierarchy forms a graph.

context

A view of a set of related modeling elements for a particular purpose, such as specifying an operation.

continuous transformation

A system in which the output actively depends on changing inputs and must be updated periodically.

contract

A contract is a specification of a set of interfaces between elements, which might be either offered (provided), required, or both. The contract includes the set of operations and events that constitute those interfaces and features. The important features of the interfaces are the parameters and their types, return values, and conditions. A [port](#) is a named connection point for a class and a typed interface (contract).

control

The aspect of a system that describes the sequences of operations that occur in response to stimuli.

control flow

A Boolean value that affects whether a process is executed.

controlled files

Files produced in other programs, such as Word or Excel, that are added to a project for reference purposes and then controlled through Rational Rhapsody.

data type

A descriptor of a set of values that lack identity and whose operations do not have side effects. Data types include primitive, predefined types and user-defined types. Predefined types include numbers, string, and time. User-defined types include enumerations.

decision node

A way of visually representing an if-then-else condition. It splits a single transition in a statechart or activity diagram into several branch transitions with guards enclosed in square brackets labeling the branches. Whichever guard is true determines to which element the object will branch. The following information applies to decision nodes and branches:

- ◆ A decision node can have only one entering transition.
- ◆ It can branch to any number of "if" conditions, but only one else condition.
- ◆ Branching segments can be nested. That is, an activity flow exiting a decision node can enter another decision node.
- ◆ Branches entering a decision node can contain triggers. Labels for transitions into a condition use the following standard format:
`trigger[guard]/action`
- ◆ Branches exiting a decision node cannot contain triggers. Labels for transitions into a condition use the following standard format:
`[guard]/action`

deep transition

A transition from a parent statechart into a nested statechart.

delegation

The ability of an object to issue a message to another object in response to a message. Delegation can be used as an alternative to inheritance. Contrast with [inheritance](#).

dependency

A relationship between modeling elements in which a change to one modeling element (the independent element) affects the other modeling element (the dependent element).

In a dependency between elements, the implementation or functioning of one element requires the presence of another element. Thus, dependency is a directed relationship. For example, an object might require the definition of another object to compile, similar to when a client element requires the presence and knowledge of a server element. Stereotypes are used to denote different types of dependency.

dependent unit

Configuration items that reference another configuration item.

deployment diagram

A diagram that shows the configuration of run-time processing nodes and the components, processes, and objects that live on them. Components represent run-time manifestations of code units. See also [component diagram](#).

derived association

An association defined in terms of other associations.

derived attribute

An attribute computed from other attributes.

derived class

A class that inherits data and member functions from a base class (subclass).

derived element

A model element that can be computed from another element, but that is shown for clarity or included for design purposes even though it adds no semantic information.

derived scope

Rational Rhapsody decides which instances to include in the configuration based on the objects you select in the **Initial Instances** box. Use the **Derived Scope** option if you are not sure which objects to include in the compilation scope.

descendant class

A class that is a direct or indirect subclass of a given class.

design

The part of the software development process whose primary purpose is to decide how the system will be implemented. It is the software development activity during which strategic and tactical solution decisions necessary for meeting client functionality and quality requirements are made. Analysis focuses on what is to be done; design focuses on how to do it.

design time

Refers to something that occurs during a design phase of the software development process. See [modeling time](#). Contrast with [analysis time](#).

destructor

An operation that cleans up an existing instance of a class (an object) that is no longer needed.

For example, an object can call a destructor to free a dynamically allocated pointer. In sequence diagrams, a destructor is represented as a dotted red arrow from the destroyer object to the object being destroyed. Destructor lines can either be horizontal or point back to the originating object. Destructor lines are not labeled.

In Rational Rhapsody Developer for C, the tasks of destructors are performed by cleanups for the class-like objects and object_types.

development process

A set of partially ordered steps performed for a given purpose during software development, such as constructing models or implementing models.

diagram

A graphical presentation of a collection of model elements, most often rendered as a connected graph of arcs (relationships) and vertices (other model elements).

The UML supports the following types of diagrams:

- ◆ Activity diagram
- ◆ Collaboration diagram
- ◆ Component diagram
- ◆ Deployment diagram
- ◆ Object model diagram
- ◆ Sequence diagram
- ◆ Statechart
- ◆ Use case diagram

diagram connector

Joins two segments of a statechart. It allows you to separate different segments of the chart onto different pages to make it easier to read. Matching labels on the source and target diagram connectors define the jump from one diagram to the next. A statechart can have several source diagram connectors with the same label, but only one destination connector of each label. Diagram connectors should have either input transitions or a single outgoing transition.

DiffMerge tool

The Rational Rhapsody DiffMerge tool supports team collaboration by showing how a design has changed between revisions and then merging units as needed. It performs a full comparison including graphical elements, text, and code differences.

This tool can be operated inside and/or outside your CM software to access the units in an archive. The [unit](#) only need to be stored as separate files in directories and accessible from the PC running the DiffMerge tool.

direct instance

An object that is an instance of a class, but not an instance of any subclass of the class.

directed association

A unidirectional relationship between objects (and users). Only the source of the directed association (client user) knows about the target (the server). The target of the relation can receive messages without knowing their source.

directional relation

A relation that exists in one direction (for example, from source to target), but not in the other.

disjoint substate

A substate that cannot be held simultaneously with other substates contained in the same composite state. Contrast with [concurrent substate](#).

distribution unit

A set of objects or components allocated to a process or a processor as a group. A distribution unit can be represented by a run-time composite or an aggregate.

DoDAF

U.S. Department of Defense Architectural Framework (DoDAF) provides an industry standard for diagrams and notations used for developing DoDAF-compliant architecture models.

domain

An area of knowledge or activity characterized by a set of concepts and terminology used by practitioners in that specific area of knowledge or activity.

dynamic classification

A semantic variation of generalization in which an object might change its classifier. Contrast with [static classification](#).

dynamic model

A description of aspects of a system concerned with control, including time, sequencing of operations, and interaction of objects.

dynamic simulation

A system that models or tracks objects in the real world.

element

An atomic constituent of a model. See also [primary model elements](#).

element name

Specifies the name of the element. As a benefit of hierarchical arrangement of model elements, primary model elements can be entered/identified by a path name in the following format:

`<ns1>::<ns2>::...::<nsn>::<name>`

In this format, `<ns>` can be the name of a package or an object. For example, object `X` in package `P` can be entered for a search or specified in an OMD as `P::X`.

enterprise architecture

Enterprise architecture is the practice of applying a comprehensive and rigorous method for describing a current and/or future structure and behavior for an organization's processes, information systems, personnel, and organization sub-units, so that they align with the organization's core goals and strategic direction. It is effectively a structured approach to describing how a business works or is intended to work so that it can reach its primary objectives. Enterprise architecture is used typically by the military for capability procurement, by governments, and by large businesses.

entry action

An action executed upon entering a state in a state machine, regardless of the transition taken to reach that state.

enumeration

A list of named values used as the range of a particular attribute type. For example, `RGBColor = {red, green, blue}`. `Boolean` is a predefined enumeration with values from the set `{false, true}`.

event

The specification of a significant occurrence that has a location in time and space.

Statecharts use events to describe the behavior of objects. In that context, an event is an instantaneous occurrence that can trigger a state transition in a class.

The use of events facilitates asynchronous collaborations. In other words, objects that generate events do not need to wait for a response before continuing with their next task.

Because events do not exist as in any programming language, they can be implemented in a variety of ways.

event attribute

Because an event is a type of object, it has its own data elements called event attributes.

Event Received breakpoint condition

Interrupts a running application when the object receives an event.

event reception

Represents an object's ability to react to an event. In other words, the event can be a trigger in that object's statechart.

Event Sent breakpoint condition

Interrupts a running application when the object sends an event to another object.

executable

A file, application, or program that can perform operations when launched. In Rational Rhapsody, executable components have a file extension defined in the `<lang>_CG::<Environment>::ExeExtension` property. A common executable extension is ".exe."

exit action

An action executed upon exiting a state in a state machine, regardless of the transition taken to exit that state.

explicit scope

In **Configuration Settings**, explicit scope means that you explicitly select which packages and objects are used for compiling an application.

export

In the context of packages, to export is to make an element visible outside its enclosing namespace. See also [visibility](#). Contrast with [import](#).

expression

In Rational Rhapsody, an expression is a string that evaluates to a value of a particular type. For example, the expression `(7 + 5 * 3)` evaluates to a value of type number.

In [Q Language](#) in ReporterPLUS, expressions examine and gather information about the model. Basic expressions are the fundamental building blocks of all Q language expressions. They are similar to numbers in arithmetic expressions. Composite expressions are the means by which larger expressions are constructed from smaller expressions. Hence, they are similar to arithmetic operators.

Another parallel between arithmetic expressions and expressions in Q is in evaluation. Just as in arithmetic expressions, the evaluation of an expression proceeds recursively, with each composite expression evaluating its subexpressions.

extend relationship

A relationship from an extension use case to a base use case that specifies how the behavior defined for the extension use case augments (subject to conditions specified in the extension) the behavior defined for the base use case. The behavior is inserted at the location defined by the extension point in the base use case. The base use case does not depend on performing the behavior of the extension use case. See also [extension points](#).

extension points

Aspects of a use case that allow it to be extended in the future. To extend a use case means to inherit one use case from another. Use cases are extended by means of «Usage» or «Extends» stereotypes. In a «Usage» relationship, the sub-use case depends on the behaviors provided by the super-use case. In an «Extends» relationship, the subuse case adds its own behaviors to those of the super-use case.

facade

A stereotyped package containing only references to model elements owned by another package. It is used to provide a “public view” of some of the contents of a package.

feature

A property like an operation or attribute that is encapsulated within a classifier, such as an interface, a class, or a data type.

A feature is a modifiable item in a tabbed window. Related features that appear on the same tab in a window are internally stored in tab groups.

file

Placeholders for generated logical units (such as packages, classes, and so on) and verbatim code segments. Files exist only within the context of a component.

file diagram

A file diagram shows how files interact with one another. Typically, a file diagram shows how the `#include` structure is created. A file diagram provides a graphical representation of the system structure. The Rational Rhapsody code generator directly translates the elements and relationships modeled in a file diagram into C source code.

final state

A special kind of state signifying that the enclosing composite state or entire state machine is completed.

fire

Execute a state transition.

fixed-point support

Rational Rhapsody Developer for C supports variables (fixed points) that represent non-integral values. The user can define the word size and precision of the variable.

flat mode

Specifies that category nodes are not displayed in the browser for most metatypes. Instead, they are displayed for all first and second-level metatypes: root node, components, diagrams, and packages. Within these categories, however, all items are displayed alphabetically without being subdivided according to metatype.

flat statechart

Specifies that states are implemented as simple, enumeration-type variables.

When statecharts are inherited, the implementation is duplicated from the base class. This strategy is more effective with shallow statechart inheritance hierarchies.

floating-point support

Rational Rhapsody Developer for C supports constants, variables, or expressions that evaluate to an integer or floating-point number.

flow chart

A flow chart is a schematic representation of an algorithm or a process. In UML and Rational Rhapsody, you can think of a flow chart as a subset of an activity diagram that is defined on methods and functions. For more information, see [Flow charts](#).

flow ports

Flowports allow you to represent the flow of data between [blocks](#) in an [object model diagram \(OMD\)](#), without defining events and operations. Flowports can be added to blocks and classes in object model diagrams. They allow you to update an [attribute](#) in one object automatically when an attribute of the same type in another object changes its value.

focus of control

A symbol on a sequence diagram that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure.

folder

File subdirectories that appear in the browser under the Files category for a component. They are set to contain files that are compiled together to build the component. When you assign (map) generated model elements to a folder, you are telling Rational Rhapsody to place the generated files for the specified elements in that subdirectory of the project. Elements that are not assigned to any folder are generated in the configuration directory.

formal parameter

A synonym for *parameter*.

framework

There are two definitions:

- ◆ A stereotyped package consisting mainly of patterns
- ◆ An architectural pattern that provides an extensible template for applications within a specific domain

function template

Specifies individual functions constructed using parameterized types that are supplied as input parameters when the function is instantiated into a template function.

In the following C++ example, a function template is defined for the global function swap:

```
template <class T> void swap (T& x, T&y) {  
    T temp = x;  
    x=y;  
    y = temp;  
}
```

Function templates are called the same way ordinary functions are called. This action generates a template function such as the following code:

```
int i=22, j=66;  
swap(i, j);
```

Rational Rhapsody allows you to define global function templates through the mechanism for defining global functions.

generalizable element

A model element that can participate in a generalization relationship.

generalization

A taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. An instance of the more specific element can be used where the more general element is allowed. See also [inheritance](#).

generic element

In ReporterPLUS, an element that represents a type of element that might be found in any Rational Rhapsody model. See also [model element](#).

global object

In C, all top-level objects are global across the project. Nested objects only have local scope relative to the object in which they are nested.

Got Control breakpoint condition

Interrupts a running application when the object gets control. An object gets control when it starts executing a member operation, responds to an event, or when an operation that the object has called on another object finishes and the object resumes its own execution.

Note

Do not enter any information in the **Data** box of the Define Breakpoint window for this condition.

gravity distance

The minimum space, in points, that must always remain between adjacent graphic elements.

guard condition

A condition that must be satisfied in order to enable an associated transition to fire. It is a Boolean expression in dynamic modeling that must be true for a transition to occur.

Harmony process

Systems engineering steps, a standard, iterative workflow, and SysML diagrams used to simplify the communication among all of the participating groups in a design project. See the [Harmony Process](#) diagram for a high-level view of this process.

helper applications

Custom programs that you attach to Rational Rhapsody to extend its functionality. They can be either external programs (executables) or Visual Basic for Applications (VBA) macros that typically use the Rational Rhapsody COM API. They connect to a Rational Rhapsody object via the `GetObject()` COM service.

history connector

Recalls the most recent active configuration of a state and its substates. Each state can have only one history connector. A history connector transitively restores all the subconfigurations that originated in the state.

A transition originating in a history connector denotes the history default. The history initial connector is taken if no history existed prior to entry into the history connector.

host machine

For a node-locked license, the host machine is the machine on which Rational Rhapsody is running. For a floating license, the host machine is the machine on which the license server is running.

identity

A distinguishing characteristic of an object that denotes a separate existence of the object, even though the object might have the same data values as another object.

implementation

A definition of how something is constructed or computed. For example, a class is an implementation of a type; a method is an implementation of an operation.

implementation inheritance

The inheritance of the implementation of a more specific element. It includes inheritance of the interface. Contrast with [interface inheritance](#).

implementation method

A style that implements specific computations on fully specified arguments, but does not make context-sensitive decisions.

import

In the context of packages, import is a dependency that shows the packages whose classes can be referenced within a given package (including packages recursively embedded within it). Contrast with [export](#).

include relationship

A relationship from a base use case to an inclusion use case, specifying how the behavior for the base use case contains the behavior of the inclusion use case. The behavior is included at the location defined in the base use case. The base use case depends on performing the behavior of the inclusion use case, but not on its structure (that is, attributes or operations). See also [extend relationship](#).

inherent concurrency

Two objects that can receive events at the same time without interacting have inherent concurrency.

inheritance

The derivation of one class from one or more other classes. The derived class inherits the same data members and behaviors present in the parent class. It is the mechanism by which more specific elements incorporate structure and behavior of more general elements related by behavior. See also [generalization](#).

initial connector

A transition to the default state of the object. An initial connector can have neither a trigger nor a guard. It can, however, have an action and lead to a decision node, after which there might be guards.

initial instance

An instance of a class that typically bootstraps the system. To configure a system, this class must be instantiated first.

initializer

In Rational Rhapsody Developer for C, a non-object-oriented language, an initializer is called when an object is instantiated much like a constructor is called when a class is instantiated into an object. A Rational Rhapsody Developer for C object or object_type can specify an initializer to explicitly initialize object members or dynamically allocate space for member pointers.

In sequence diagrams, an initializer is represented as a dotted, green arrow from the creator object or system border to the object being created. Constructor lines are horizontal.

instance

An entity to which a set of operations can be applied and that has a state that stores the effects of the operations.

An object described by a class.

Instance Created breakpoint condition

Interrupts a running application when the specified class or a subclass of it is instantiated.

For these breakpoints, specify the instance name without an instance number. For example, you can use the instance name Heater, but not the instance name Heater[0].

Instance Deleted breakpoint condition

Interrupts a running application when an object is deleted.

instance diagram

An object diagram that describes how a particular set of object instances relate to each other.

instance line

A vertical timeline in a sequence diagram that shows the sequence of messages that an object processes and states that it enters over its lifetime.

instantiation

For classes, this process denotes of creation of objects (instances) from classes. For templates, this process denotes the creation of a template class from a class template or a template function from a function template.

interaction

A specification of how stimuli are sent between instances to perform a specific task. The interaction is defined in the context of a collaboration. See also [collaboration](#).

interaction diagram

A generic term that applies to several types of diagrams that emphasize object interactions, including collaboration and sequence diagrams.

interaction occurrence

An interaction occurrence (or reference sequence diagram) enables you to refer to another sequence diagram from within a sequence diagram. This allows you to break down complex scenarios into smaller scenarios that can be reused. Each such scenario is an *interaction*.

interface

A named set of operations that characterize the behavior of an object.

interface inheritance

The inheritance of the interface of a more specific element. It does not include inheritance of the implementation. Contrast with [implementation inheritance](#).

internal transition

A transition signifying a response to an event without changing the state of its object.

iteration node

In ReporterPLUS, a template node formed by dragging a generic or model element to the template view. (An iteration subnode is created at the same time.) The iteration node specifies what class the iteration pertains to and what element to extract from that class. Iteration nodes can also specify conditions applied to the iteration, how elements are sorted, what happens when the iteration does not yield elements from the model, and can contain attributes and boilerplate text.

iteration subnode

In ReporterPLUS, a template node formed by dragging a generic or model element to the template view. The iteration subnode specifies the information included in the generated document for each element extracted by the iteration. Subnodes can contain attributes and boilerplate text, and might be the parent node to further iterations.

keywords

In ReporterPLUS, reserved words used in [Q Language](#) commands that have preset functions.

layer

The organization of classifiers or packages at the same level of abstraction. A layer represents a horizontal slice through an architecture, whereas a partition represents a vertical slice.

leaf state

A state without And state components and descendants.

library

A library component has a `.lib` extension.

link

A semantic connection among multiple objects. It is an instance of an association.

link attribute

A named data value held by each link in an association.

link end

An instance of an association end.

Lost Control breakpoint condition

Interrupts a running application when the object loses control. An object loses control when it finishes executing an operation, finishes responding to an event, or calls an operation on another object.

Note: Do not enter any information in the **Data** box of the Define Breakpoint window for this condition.

merge node

Joins more than one transition into a single, outgoing transition. This enables you to combine several segments into a single, graphical description or use a common transition suffix.

message

A specification of the conveyance of information from one instance to another, with the expectation that activity will ensue. A message can raise a signal or call an operation.

message diagram

Message diagrams, available in the FunctionalC profile, show how the files functionality might interact through messaging (through synchronous function calls or asynchronous communication). Message diagrams can be used at different levels of abstraction. At higher levels of abstractions, message diagrams show the interactions between actors, use cases, and objects. At lower levels of abstraction and for implementation, message diagrams show the communication between classes and objects.

Message diagrams have an executable aspect and are a key animation tool. When you animate a model, Rational Rhapsody dynamically builds message diagrams that record the object-to-object messaging.

Rational Rhapsody message diagrams are based on [Sequence diagrams](#). For more information about the FunctionalC profile, see [Profiles](#).

message passing

The means by which objects communicate with one another to provide information, send information, and launch actions. Sending messages is how work gets done in an object-oriented system.

message-to-self

An arrow that bends back onto the originating instance line. The arrow can be on either side of the instance line. If the message is a primitive operation, it immediately folds back (operations are synchronous). If it is an event, it can fold back sometime later (events are asynchronous).

metaclass

A class whose instances are classes, meaning a class of classes. In Rational Rhapsody, metaclasses are used to define properties for whole classes of objects. For example, under the subject of code generation (CG), there are metaclasses for Class, Component, and Configuration.

metamodel

A model that defines the language for expressing a model.

meta-metamodel

A model that defines the language for expressing a metamodel. The relationship between a meta-metamodel and a metamodel is analogous to the relationship between a metamodel and a model.

metaobject

A generic term for all metaentities in a metamodeling language. For example, metatypes, metaclasses, metaattributes, and metaassociations.

method

The implementation of an operation. It specifies the algorithm or procedure associated with an operation.

methodology

A process for the organized production of software using a collection of predefined and notational conventions.

MODAF

United Kingdom's Ministry of Defence Architectural Framework (MODAF) provides an industry standard for diagrams and notations used for developing MODAF-compliant architecture models. This standard builds on the U.S. [DoDAF](#) standard.

mode

The access permission of a unit of collaboration. You can change a read/write (RW) unit, but you cannot change a read-only (RO) unit. An item is "locked" if it is RW for you and RO for others.

model

An abstraction of something for the purpose of understanding it before building it. The model is the overall database of information about your project. Different projects can use the same model. You perform requirements modeling using UCDs, static object modeling using OMDs, and dynamic modeling using sequence diagrams and statecharts.

model aspect

A dimension of modeling that emphasizes particular qualities of the metamodel. For example, the structural model aspect emphasizes the structural qualities of the metamodel.

Model-driven Development (MDD)

This development environment allows designers to visualize a domain, system, product, or process and create that design in diagrams within a model. Then the modeling tool generates implementation artifacts for the design for a selected target, such C++ code.

model elaboration

The process of generating a repository type from a published model. It includes the generation of interfaces and implementations that allows repositories to be instantiated and populated based on, and in compliance with, the model elaborated.

model element

In ReporterPLUS, an element from a specific model. See also [generic element](#).

model view

The upper, left pane in the ReporterPLUS window. When a model is open, the model view displays generic and model elements. When no model is open, it displays only generic elements. Compare with [attribute view](#).

model-view controller (MVC)

An architecture that separates the model and its views by allowing certain model elements to be viewed in different views or not at all.

modeling time

Refers to something that occurs during a modeling phase of the software development process. It includes analysis time and design time. When discussing object systems, it is often important to distinguish between modeling-time and run-time concerns.

module

A software unit of storage and manipulation. Modules include source code modules, binary code modules, and executable code modules. See also [component](#).

multiple classification

A semantic variation of generalization in which an object can belong directly to more than one classifier. See [static classification](#) and [dynamic classification](#).

multiple inheritance

A semantic variation of generalization in which a type can have more than one supertype. Contrast with [single inheritance](#).

multiplicity

Refers to the number of instances of one class that can relate to a single instance of an associated class.

n-ary association

An association among three or more classes. Each instance of the association is an *n*-tuple of values from the respective classes. Contrast with [binary association](#).

namespace

A part of the model in which the names can be defined and used. Within a namespace, each name has a unique meaning.

nested unit

An element that is inside an element of the same [unit](#) type. For example, a package might contain another package (or a nested unit).

node

In Rational Rhapsody, a classifier that represents a run-time computational resource, which generally has at least a memory, and often processing, capability. Run-time objects and components can reside on nodes.

In ReporterPLUS, this is basically a section in a report that is specified at a heading level in the report.

node label

In ReporterPLUS, the text next to the template node icon in the template view. The node label describes what the node does and what part of the model it is from. ReporterPLUS adds a default label automatically; you can change it in the **User-defined label** box on the **Properties** tab.

non-public inheritance

The subclass inherits only the public attributes and operations of the superclass.

null transition

A transition with a guard and an action, but no trigger. Null transitions can be useful when you want to allocate a resource that might not be available. In this case, you might branch based on some entry action or join transition.

object

An entity with a well-defined boundary and identity that encapsulates state and behavior. *State* is represented by attributes and relationships, whereas *behavior* is represented by operations, methods, and state machines.

An object is an instance of a class.

In Rational Rhapsody Developer for C, an object consists of a C struct that is mapped, via naming conventions, to operations (functions). This unifies attributes (data) and operations (functions) under one element, a C object, which displays as a primary element in both the browser and OMD. A Rational Rhapsody Developer for C object can be defined as an object of implicit type or an object of explicit type.

object box

In an OMD, a rectangle with three compartments (unless it is a simple object box). The first compartment contains the object name, the second contains the attributes, and the third contains operations.

The symbol for a simple object is simply a rectangle without compartments. An object and simple object are semantically equivalent.

object design

A stage of the development cycle during which the implementation of each class, association, attribute, and operation is determined.

object diagram

A diagram that encompasses objects and their relationships at a point in time. An object diagram can be considered a special case of a class diagram or a collaboration diagram. See [class diagram](#) and [collaboration diagram](#).

object execution framework (OXF)

Rational Rhapsody has one central run-time library, the OXF, that provides all the services required by the generated code in the running application. All the other libraries are related to animation or tracing in one respect or another. See also [framework](#).

object flow state

A state in an activity graph that represents the passing of an object from the output of actions in one state to the input of actions in another state.

object group

Consists of several objects that are examined together as a unit for the purpose of comparing two sequence diagrams. This enables you to compare, for example, a black box diagram showing messages to and from the system as a whole to a gray or white box diagram showing messages to and from the individual parts. For example, given a model of an oven consisting of `Oven`, `Timer`, and `Display` objects, you can create both a black box sequence diagram showing an `Oven` object and a white-box sequence diagram showing objects of all three.

Without object groups, you could compare only the `Oven` objects in the two diagrams, omitting from the comparison any messages to or from the `Timer` and `Display`. Using an object group, you can compare messages to and from the `Oven` in the black box diagram to those to and from the `Oven`, `Timer`, and `Display` in the white box diagram because they are considered to be a single unit. This results in fewer discrepancies and simplifies the comparison.

object lifeline

A line in a sequence diagram that represents the existence of an object over a period of time. See also [sequence diagram](#).

object model

A description of the structure of the objects in a system including their identity, relationships to other objects, attributes, and operations.

object model diagram (OMD)

Show the logical views of a system. OMDs are static, structural diagrams that show the parts of a software system and the relationships that exist between them. These parts can include classes (in C, objects and object types), packages, and actors.

OMDs include objects that appear in related scenarios, which are described using sequence diagrams. OMDs show all relationships, including inheritance, dependencies, compositions, and associations between collaborating objects.

OMDs are constructive in that Rational Rhapsody generates code based on what you draw in them.

object modeling technique (OMT)

An object-oriented development methodology that uses object, dynamic, and functional models throughout the development life cycle.

object of explicit type

In Rational Rhapsody Developer for C, an object of explicit type is defined by a single reference to an object type that defines the object. If an object A is based on an object type B, this is expressed as “object A is of type B.” The name of the object A displays in an OMD as A:B.

Objects of both implicit and explicit types show all or part of their object type-related properties as part of their object appearance. However, objects of explicit type cannot alter or add directly to their object type-related properties. In other words, an object of explicit type is explicitly and completely defined by its object type.

Objects are set as implicit or explicit type through the **Is Of Type** check box in the Object window. If, for a particular object, this box is checked and an object type is specified in the adjacent box, the object is an explicit object. Otherwise, the object is an implicit object.

You can convert objects of explicit type to objects of implicit type through the **Create Type-less Object** feature. This option copies all properties (functions, statechart, and so on) of the object’s object type to a new object of implicit type.

object of implicit type

In Rational Rhapsody Developer for C, objects of implicit type are defined only by the attributes, operations, objects, and object types that they contain. You can display these elements in the box representation of the implicit object base.

Both implicit and explicit objects show all or part of their object type-related properties as part of their object appearance.

You can convert an implicit object to an explicit object and object type through the **Expose Object Type** feature.

object-oriented

A software development strategy that organizes software as a collection of objects that contain both data structure and behavior.

object type

Rational Rhapsody Developer for C uses object types as templates for defining objects. For example, if A is an object type, object B can be defined as an object of type A, also referred to as B:A.

operation

A service that can be requested from an object to affect behavior. An operation has a signature, which might restrict the actual parameters that are possible.

Operation breakpoint condition

Interrupts a running application when the object starts executing a member operation.

Operation Returned breakpoint condition

Interrupts a running application when a member operation of the object returns.

origin class

The top-most class that defines an attribute or operation.

orthogonal state

Two or more independent states that an object can be in at the same time. This is also known as an And state. For example, a clock radio can be counting the time and playing music at the same time, states that can be completely independent of each other. Dotted lines separate the compartments of orthogonal states in an And state.

output type

In ReporterPLUS, the format of the generated document. ReporterPLUS can produce documents in five output types: Microsoft Word, Microsoft PowerPoint, HTML, Rich Text Format (RTF), and text. You select the output type in the Generate Document window.

override

Replace an inherited method for the same operation, or replace the default value of a property with a new value.

package

A general-purpose mechanism for organizing elements into groups. You can think of a system as a single, high-level package, with everything else in the system contained in it. A package is a collection of packages, classes (in C, objects and object types), events, diagrams, globals, types, use cases, and actors.

Because packages can be nested with other packages, they enable you to partition a system into smaller subsystems. Thus, package nesting provides a way to organize large projects into package hierarchies.

You can import packages into your project from other projects using **File > Add to Model**.

panel diagram

A panel diagram provides you with a number of graphical control elements that you can use as a graphical user interface (GUI) to monitor and regulate an application. Each control element can be bound to a model element (variable/attribute/event/state). During animation, you can use the animated panel diagram to monitor (read) and regulate (change values/send events) your user application.

Panel diagrams are intended only for simulating and prototyping, and not for use as a production interface for the user application. In addition, panel diagrams can only be “used” on the host and can be “used” only from within Rational Rhapsody

parameter

The specification of a variable that can be changed, passed, or returned. A parameter can include a name, type, and direction. Parameters are used for operations, messages, and events. See also [formal parameter](#). Contrast with [argument](#).

parameterized element

The descriptor for a class with one or more unbound parameters. Also referred to as a [template](#).

parent

In a generalization relationship, the generalization of another element (the child). See also [subclass](#), [subtype](#). Contrast with [child](#).

part

A part is a role that an instance of that class plays in a context. Meaning it is a stand-in for an object.

Note that attributes, when used by value, will be generated like a part. However, if its type is reactive, then the attribute would be missing the reactive code (startbehavior call, and so on).

Use a part (rather than an attribute) if it is to be used as an stand-in for an object. The advantage of using a part is that it can be drawn on a diagram, you can connect parts using links, and also view ports, attributes, operations, and so on. See also [class](#), [package](#), and [component](#). Contrast with [attribute](#).

participation

The connection of a model element to a relationship. For example, a class participates in an association, whereas an actor participates in a use case.

pattern

A template collaboration.

partition

There are two possible definitions:

- ♦ A portion of an activity graph that organizes the responsibilities for actions. See also [swimlane](#).
- ♦ A set of related classifiers or packages at the same level of abstraction or across layers in a layered architecture. A partition represents a vertical slice through an architecture, whereas a layer represents a horizontal slice.

partition line

A red, horizontal line dividing a sequence diagram into vertical segments indicating different phases of the sequence. The text note that accompanies a partition line is initially positioned at the left end of the line. You can move the note anywhere in the diagram, but it always remains associated with the same partition line.

periodic actions

To model periodic actions that repeat as long as a state is active, you can use a transition that loops back to the same state with a timer as a trigger. For example, to define an action that should repeat once every second while an object is in some state, set the trigger on the loop transition to `tm(1000)` and set the action on entry to the statements to be executed at that interval.

persistent object

An object that exists after the process or thread that created it has ceased to exist.

physical system

Either of the following items:

- ◆ The subject of a model.
- ◆ A collection of connected physical units, which can include software, hardware and people, that are organized to accomplish a specific purpose. A physical system can be described by one or more models, possibly from different viewpoints. Contrast with [system](#).

pinned window

In “pinned” mode, the information displayed in the “pinned” Features window remains displayed and accessible from all of the window tabs even when a different element is selected. This allows multiple windows to be displayed and the contents reviewed together. The pin icon in the upper right corner of the window controls this feature.

plug-in

A Rational Rhapsody plug-in is a user Java application that Rational Rhapsody loads into its process. While a plug-in is loaded, Rational Rhapsody supports it with an interface to the hosting Rational Rhapsody application.

Use a plug-in to extend the Rational Rhapsody features with customized functionality by adding menus to the Rational Rhapsody IDE or reacting to Rational Rhapsody events, such as code generation and model check.

polymorphism

A way of simplifying communication between objects by hiding different implementations behind a common interface. For example, defining multiple print methods behind one print command.

port

As in the standard [Unified Modeling Language \(UML\)](#), a port is a named connection point for a [class](#), [object](#), or a [block](#). It is a distinct interaction point between a class and its environment or between (the behavior of) a class and its internal parts. A port allows you to specify classes that are independent of the environment in which they are embedded. The internal parts of the class can be completely isolated from the environment and vice versa. A port can have either of these interfaces: required or provided.

postcondition

A constraint that must be true at the completion of an operation.

precondition

A constraint that must be true when an operation is launched.

predefined types

The Rational Rhapsody predefined types include `int`, `char`, `double`, `void`, `OMBoolean`, `long`, `OMString`, and `void*`. They do not belong to the `Default` or any other user-defined package. Rather, they belong to the `PredefinedTypes` package, which is part of every model, albeit hidden (in the `PredefinedTypes.sbs` file in the `Share\Properties` directory). Thus, clashes are prevented between user-defined and predefined types with the same name.

primary model elements

Primary model elements within the browser are packages, classes, OMDs, associations, dependencies, operations, variables, events, event receptions, triggered operations, constructors, destructors, and types. Primary model elements in OMDs are packages, classes, associations (links), dependencies, and actors.

Rational Rhapsody Developer for C and C++ classes and their instances are replaced by C equivalent object types and objects, respectively. Similarly, class constructors and destructors are replaced by initializers and cleanup operations.

primitive operation

An operation whose body you write yourself. Rational Rhapsody automatically generates bodies for all other types of operations.

primitive type

A predefined, basic data type without any substructure, such as an integer or a string.

private

When referring to an attribute or operation of a class, private means accessible only by methods of the current class.

process

Has the following meanings:

- ♦ A heavyweight unit of concurrency and execution in an operating system. Contrast this with thread, which includes heavyweight and lightweight processes. If necessary, an implementation distinction can be made using stereotypes.
- ♦ A software development process; the steps and guidelines by which to develop a system.
- ♦ To execute an algorithm or otherwise handle something dynamically.

profile

A special kind of package that is distinguished from other packages in the browser. You specify a profile at the top level of the model. Therefore, a profile is owned by the project and affects the entire model. By default, all profiles apply to all packages available in the workspace, so their tags and stereotypes are available everywhere. A profile is similar to any other package; however, profiles cannot be nested. You might select a specific profile when you create a project and add profiles to existing projects, and Rational Rhapsody automatically adds profiles required for add-on products. See [Profiles](#) for more information.

project

Overall set of artifacts describing the system being modeled across all developers. The terms project and [model](#) are interchangeable in Rational Rhapsody.

projection

A mapping from a set to a subset of it.

property

A named value denoting a characteristic of an element. A property has semantic impact. The UML predefines some properties; others are user-defined. The definitions of Rational Rhapsody properties are displayed in the **Properties** tab of the Features window.

See also [tagged value](#).

protected object

In a system with several threads of control, there can be contentions where different threads apply messages to the same passive object. A protected object serves only one message at a time. In other words, an object applying a message to a protected object will be blocked until the protected object processes the message.

provided event

An event to which a class responds. It is defined with the class or its superclass.

pseudo-state

A vertex in a state machine that has the form of a state, but does not behave as a state. Pseudo-states include initial and history vertices.

public

Accessible by methods of any class.

public inheritance

The subclass inherits all attributes and operations of the superclass.

Q Language

In ReporterPLUS, the language used to write advanced conditions and create advanced statements about attributes.

qualifier

An association attribute or tuple of attributes whose values partition the set of objects related to an object across an association.

qualified association

An association that relates two classes and a qualifier; a binary association in which the first part is a composite comprising a class and a qualifier, and the second part is a class.

reactive object

A class (in C, an object or object type) is reactive if:

- ♦ It has a statechart.
- ♦ It is a composite.
- ♦ It has an event reception.

Most reactive objects and object types have statecharts to define their behavior. The presence of a statechart for an object or object type is indicated by the appearance of a small overlaid mini-statechart icon in the browser, OMD, and sequence diagrams.

read-only mode

You can view an item, but cannot modify it.

real-time model

Runs the instrumented application in quasi-real time in which timeouts and delays are computed based on the system clock.

read/write mode

You can modify an item.

receive a message

The handling of a stimulus passed from a sender instance. See also [sender object](#), [receiver object](#).

receiver object

The object handling a stimulus passed from a sender object. Contrast with [sender object](#).

reception

A declaration that a classifier is prepared to react to the receipt of a signal.

refactoring

Search and replace mechanism to rename an element in the user code when the engineer has renamed it in the Rational Rhapsody project.

reference

Has the following definitions:

- ♦ A denotation of a model element
- ♦ A named slot within a classifier that facilitates navigation to other classifiers

Also called a pointer.

reference class

Classes that are included in the model by reference only, without specifying any of their internal details, such as attributes. For example, you can include the MFC classes for reference, merely to show that they are acting as superclasses or peer classes to other classes in your model, without including any specific information about the MFC classes themselves.

refinement

A relationship that represents a fuller specification of something that has already been specified at a certain level of detail. For example, a design class is a refinement of an analysis class.

relation

Elements can be related to each other in different ways. The relationships that can exist between software entities are modeled on relationships that exist in the real world, as follows:

- ♦ Association
- ♦ Directed association
- ♦ Dependency

Relation breakpoint condition

Interrupts a running application when the object modifies a relation in any way. In other words, the application breaks if the object connects to, disconnects from, or clears a relation.

Relation Cleared breakpoint condition

Interrupts a running application when the object clears a relation. To clear a relation means to disconnect from all instances on the relation and clear the relation itself.

Relation Connected breakpoint condition

Interrupts a running application when the object connects to a relation.

Relation Disconnected breakpoint condition

Interrupts a running application when the object disconnects from a relation. To disconnect from a relation means to disconnect from one individual instance on the relation. Connections to other instances on the same relation remain intact.

relationship

A semantic connection among model elements. Examples of relationships include associations and generalizations.

ReporterPLUS template

A set of instructions that tells ReporterPLUS what data to extract from a model and how that data should be formatted. You can build your own ReporterPLUS templates or use the ones provided with ReporterPLUS.

repository

A facility for storing object models, interfaces, and implementations. Repository files are in ASCII format for easy storage and differentiation by a [CM](#) system.

The repository can also be defined as a subdirectory inside the project directory containing all configuration items in the project, such as diagram, class, and package files. The directory name consists of the project name followed by `_rpy`.

required event

An event that a class knows how to generate. It is not defined with the class, but with the target of the event.

requirement

A wanted feature, property, or behavior of a system.

respect mode

In C++ projects, Rational Rhapsody preserves (respects) the changes to the model and other information changes through subsequent code generations.

responsibility

A contract or obligation of a classifier. It defines what the system expects from an object.

reusable statechart

With reusable statechart implementation, states are implemented as class types derived from the abstract state classes defined in the implementation framework. When statecharts are inherited, states are reused by instantiating the same state object from a base class. This strategy is more effective with deep statechart inheritance hierarchies.

reuse

The use of a pre-existing artifact.

role

The named specific behavior of an entity participating in a particular context. A role can be static (for example, an association end) or dynamic (for example, a collaboration role).

In the case of an association, a role specifies how each object relates to the object at the other end of the association.

root state

The default state of a statechart.

roundtrip

The action taken to update a Rational Rhapsody model based on changes made to code previously generated by Rational Rhapsody.

run time

The period of time during which a computer program executes. Contrast with [modeling time](#).

SCC

Abbreviation for the Microsoft Common Source Code Control.

scenario

A specific sequence of actions that illustrates behaviors. A scenario can be used to illustrate an interaction or the execution of a use case instance. See also [interaction](#).

semantic variation point

A point of variation in the semantics of a metamodel. It provides an intentional degree of freedom for the interpretation of the metamodel semantics.

send a message

The passing of a stimulus from a sender instance to a receiver instance. See also [sender object](#), [receiver object](#).

sender object

The object passing a stimulus to a receiver object.

sequence diagram

A diagram that shows object interactions arranged in time sequence. In particular, it shows the objects participating in the interaction and the sequence of messages exchanged between them. Vertical lines represent the objects and arrows are drawn to represent the messages.

Unlike a collaboration diagram, a sequence diagram includes time sequences but does not include object relationships. A sequence diagram can exist in a generic form (describing all possible scenarios) and in an instance form (describing one actual scenario). Sequence diagrams and collaboration diagrams express similar information, but show it in different ways.

In addition to drawing sequence diagrams, Rational Rhapsody also creates animated sequence diagrams from your running application so you can see that your code is really working the way you want it to.

sequential concurrency

The system runs on a single thread and all operations are executed in sequential order. A sequential object either exists within a composite, or can be a global instance.

Service Oriented Architecture (SOA)

SOA projects are business-centric and facilitate an effective IT infrastructure to help streamline and improve processes across the enterprise. See [Domain-specific projects and the NetCentric profile](#) for more information.

signal

The specification of an asynchronous stimulus communicated between instances. Signals can have parameters.

signature

The name and parameters of a behavioral feature. A signature can include an optional, returned parameter.

simulated time model

A virtual timer orders timeouts and delays, which are posted when the system completes a computation.

single inheritance

A semantic variation of generalization in which a type can have only one supertype. Contrast with [multiple inheritance](#).

singleton

An object type that, by definition, has exactly one instance. It is designated by a small “1” in the upper, right corner of the object type box in an OMD.

SourceArtifact

A Rational Rhapsody element that contains code respect information for reverse engineering and roundtripping.

Note that previous to Rational Rhapsody version 7.2, a SourceArtifact was referred to as a component file. While component files still exist, they now refer to elements under the **Components** category in Rational Rhapsody. When component files are located under packages or classes, and so on, they are referred to as SourceArtifacts.

specification

A declarative description of what something is or does. Contrast with [implementation](#).

spontaneous transition

A transition in dynamic modeling that fires only if a guard condition is true.

state

An abstraction of the mode in which the object finds itself. It is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.

Messages, events, timeouts, and the satisfaction of conditions can cause an object to transition from one state to another. Statecharts define an object’s behavior by specifying messages, events, timeouts, and conditions that must occur to cause the object to transition from one state to another.

state awareness

Rational Rhapsody shows the configuration management “state” of elements in the [browser](#) with icons for the elements. The icons provide the following configuration management information about the individual items:

- ◆ Under source control
- ◆ Not under source control
- ◆ Checked out
- ◆ Checked in with changes
- ◆ Deleted from source control

State breakpoint condition

Interrupts a running application when the object changes state.

State Exited breakpoint condition

Interrupts a running application when the object exits a state.

state machine

A behavior that specifies the sequences of states that an object or an interaction goes through during its lifetime in response to events, together with its responses and actions. In Rational Rhapsody, the state machine is diagrammed in a [statechart](#).

statechart

Defines the behavior of reactive objects by specifying how they react to events or operations. The reaction can be to perform a transition between states and possibly to execute some actions. When running in animation mode, Rational Rhapsody highlights the transition taken and the entered state to show transitions between states.

statechart diagram

A diagram that shows a state machine.

static classification

A semantic variation of generalization in which an object cannot change classifier. Contrast with [dynamic classification](#).

static structure

Used to describe an OMD. An OMD shows the relationship between instances at any given time during a system's operation. Therefore, the structure displayed by an OMD is timeless, or static.

stereotype

A type of modeling element that extends the semantics of the metamodel. Stereotypes must be based on certain existing types or classes in the metamodel. Stereotypes can extend the semantics, but not the structure of pre-existing types and classes. Certain stereotypes are predefined in the UML, whereas others are user-defined. Stereotypes are one of three extensibility mechanisms in UML. See also [constraint](#) and [tagged value](#).

Stereotypes allow extension of the UML metamodel by “typing” different UML entities. Entities that allow stereotypes in Rational Rhapsody are use cases, packages, classes, (objects and object types in Rational Rhapsody Developer for C) and dependencies. See [Stereotypes](#) for more information.

stimulus

The passing of information from one instance to another, such as raising a signal or invoking an operation. The receipt of a signal is normally considered an event. See also [message](#).

string

A sequence of text characters. The details of string representation depend on implementation, and can include character sets that support international characters and graphics.

structural feature

A static feature of a model element, such as an attribute.

structural model aspect

The model aspect that emphasizes the structure of the objects in a system, including their types, classes, relationships, attributes, and operations.

subactivity state

A state in an activity graph that represents the execution of a non-atomic sequence of steps that has some duration.

subclass

In a generalization relationship, the specialization of another class; the superclass. See also [generalization](#). Contrast with [superclass](#).

submachine

A statechart that is a decomposition of a containing state referred to as a submachine state. Viewing the submachine state in submachine view displays the contents of the submachine state, referred to as a submachine.

submachine state

A state in a state machine that is equivalent to a composite state, but whose contents are described by another state machine.

A submachine state is a composite state that is decomposed to a submachine. The submachine state is in the parent statechart.

subpackage

A package that is contained in another package.

substate

A state that is part of a composite state. See also [concurrent substate](#) and [disjoint substate](#).

subsystem

A grouping of model elements that represents a behavioral unit in a physical system. A subsystem offers interfaces and has operations. In addition, the model elements of a subsystem can be partitioned into specification and realization elements. See also [package](#) and [physical system](#).

subtype

In a generalization relationship, the specialization of another type; the supertype.

subunit

An element that is nested inside another element. For example, a package B that is nested inside a package A is considered a subunit of A.

superclass

In a generalization relationship, the generalization of another class; the subclass. See also [generalization](#). Contrast with [subclass](#).

A superclass is a base class from which another class derives. A superclass is created when you draw an inheritance relation from one class to another in an OMD.

supertype

In a generalization relationship, the generalization of another type, the [subtype](#).

supplier

A classifier that provides services that can be started by others. Contrast with [client](#).

swimlane

A partition on an activity diagram for organizing the responsibilities for actions. Swimlanes typically correspond to organizational units in a business model. See also [partition](#).

symmetric relation

Both entities know about and can send messages to each other. Also known as a bidirectional association.

synch state

A vertex in a state machine used for synchronizing the concurrent regions of a state machine.

SysML

Systems Modeling Language (SysML) is a domain specific modeling language for systems engineering applications. It supports the specification, analysis, design, verification and validation of a broad range of systems and systems-of-systems. These systems might include hardware, software, information, processes, personnel, and facilities. SysML's goal is to allow systems engineers to define their designs with precision, conciseness, consistency and understandability and to be able to test the designs for correctness. See [Systems engineering with Rational Rhapsody](#) for more information.

system

A top-level subsystem in a model. Contrast with [physical system](#).

System Architect (SA)

IBM® Rational® System Architect® software provides the tools for you to build a Business and Enterprise Architecture, a fully integrated collection of models and documents across five keys domains: Strategy, Business, Information, Systems, and Technology. System Architect software creates a shared workspace for all team members to understand how to improve the company's architecture and overall business. DoDAF data can be imported from SA into Rational Rhapsody as SysML. See [Importing DoDAF diagrams from Rational System Architect](#) for more information.

system border

Represents the environment outside the system under design. In a sequence diagram, the system border is indicated by a column of short diagonal lines. Messages that originate outside the system or subsystem shown in the sequence come from the system border.

table node

In ReporterPLUS, an iteration node that produces a table rather than paragraphs of text. The information on the table node is formatted into column headings in the generated document, whereas the information on the iteration subnodes beneath the table node is formatted into table rows.

tagged value

The explicit definition of a property as a name-value pair. In a tagged value, the name is referred to as the tag. Certain tags are predefined in the UML; others are user-defined. Tagged values are one of three extensibility mechanisms in UML. See also [constraint](#) and [stereotype](#).

target environment

Rational Rhapsody is used to develop real-time application software to run in different operating system environments that rely on different software compilers.

Rational Rhapsody distinguishes these environments based on a combination of compiler and operating system (some compilers can compile for several operating systems). See the *IBM Rational Rhapsody ReadMe* (release notes) for information on the supported target environments.

template

A parameterized element. Rational Rhapsody provides two types of templates: class and function. See also [template instantiation](#) and [template parameters](#).

For ReporterPLUS, see [ReporterPLUS template](#).

template class

The C++ language enables you to instantiate a class template into a template class using types passed to the class template as arguments. Rational Rhapsody allows you to create or change an existing class into a class template or instantiate it into a template class in the class window.

See also [class template](#).

template function

Generates a template function based on the data types passed to it as arguments.

Consider the following function template:

```
template <class T> void swap (T& x, T&y) {  
    T temp = x;  
    x=y;  
    y = temp;  
}
```

When the function template is called (the same way ordinary functions are called), a template function is generated, as follows:

```
int i=22, j=66;  
swap(i, j);
```

template instantiation

Rational Rhapsody provides C++ template instantiation in the following two cases:

- ♦ Class templates can be instantiated into template classes using the **Instantiation** radio button.
- ♦ Function templates are instantiated into template functions, using the **Is Template** check box on the global function window.

In Rational Rhapsody, the following features and limitations apply to class template instantiation:

- ♦ A new class is declared as a class template or instantiated template in the browser.
- ♦ An instantiation of a template must be full, all template parameters must have values.

Template instantiation is a named instantiation interpreted as a “typedef” of a real instantiation. The following examples demonstrate the Rational Rhapsody generated code for an instantiated class template:

```
typedef vector<int> vec_int_inst;  
typedef MyTemplateClass<int> MySimpleIntInstance;
```

In the examples, the class `vec_int_inst` is equivalent to the class `vector<CString>` and the class `MySimpleIntInstance` is equivalent to the class `MyTemplateClass<int>`.

- ♦ Operations and attributes cannot be added to an instantiation.
- ♦ A model element can be a template (by having template parameters), an instantiation (by having instantiation parameters), or neither one.
- ♦ See also [template parameters](#) and [template specialization](#).

template node (or section)

In ReporterPLUS, refers to any node in the template view. Template nodes form the structure of the generated document, and hold the attributes, boilerplate text, format commands, and other information that ReporterPLUS needs to generate a document from a model.

template node view

In ReporterPLUS, the lower, right pane in the ReporterPLUS window. The template node view has six tabs, which show information about the nodes in the template view. Compare with [template view](#). See also [attribute view](#) and [model view](#).

template parameters

When templates are instantiated into template classes or template functions, the types used in the defined body are provided as parameters. In Rational Rhapsody, the following features apply to template parameters and their scope:

- ◆ A template can define some of its parameters with default values.
- ◆ Template parameters can be class or metaclass, as shown by the following examples:

```
template<int> class x;  
template<class T> class x;
```
- ◆ Inside a template object, metaclass template parameters are treated as regular types/classes. Consider the following class template definition:

```
template<class T> class x;
```
- ◆ In this template, x can have an attribute of type T.
- ◆ Inside a template object, template parameters are treated as expressions for the purpose of instantiation. For example:

```
template<class T> class x {  
    vector<T> a;  
}
```
- ◆ In this example, T can be a metaclass or type parameter.
- ◆ The name of a metaclass can be used as a class of another parameter. For example:

```
template <class T, T t> class C {};
```
- ◆ This can be done only in the browser, not in the OMD.
- ◆ When adding a superclass, metaclass template parameters are available. For example:

```
template <class T> class C : public T {...};
```
- ◆ This can be done only in the browser, not in the OMD.

See also [template instantiation](#) and [template specialization](#).

template specialization

Allows specific instantiations to override the content of the general template.

For member operations, set the value of the `Specialization` property for the operation to the text of the instance (for example, `vector<CString>...`). For functions, the only difference

between the original template function and the specialized function is that the return type, arguments, and so on. of the specialization are instantiations with the type chosen for specialization.

Consider the template global function, `f`:

1. Define `f<T>` and in it write the general body.
2. In the same scope, define the specialized function (for example, `f<int>{ . . . }`) and in it write the specialized body.

See also [template instantiation](#) and [template parameters](#).

template view

The lower, left pane in the ReporterPLUS window. The template view displays the currently open ReporterPLUS template. This view is empty when ReporterPLUS first starts. Compare with [template node view](#). See also [attribute view](#) and [model view](#).

termination breakpoint condition

Interrupts a running application when an object reaches a termination connector in its statechart. The application does not break if the object is deleted in any other way than by entering a termination connector.

termination connector

“Suicide” or “self-destruct” connectors used in statecharts. A transition to a termination connector causes an object to delete itself.

Although termination connectors have the same appearance as termination states in activity diagrams, final activities do not destroy the object they are in.

A termination connector cannot have any outgoing transition.

text node

In ReporterPLUS, a [template node \(or section\)](#) that holds attributes and boilerplate text. Text nodes can stand on their own, serve as subnodes under iteration subnodes, or serve as parent nodes for iteration nodes. They cannot hold iterations.

thread of control

A single path of execution through a program, dynamic model, or some other representation of control flow.

In addition, a stereotype for the implementation of an active object as a lightweight process. See also [process](#).

time event

An event that denotes the time elapsed since the current state was entered. See also [event](#).

time expression

An expression that resolves to an absolute or relative value of time.

time interval

A vertical annotation that shows how much (real) time should pass between two points on an instance line in a sequence diagram. The label is free text and is not limited to a number or unit of any kind.

timeout

Used in sequence diagrams when an object should wait a set amount of time before continuing its operation. A timeout is shown as a message-to-self on an instance line with a small box at the start end of the message line. Timeouts are labeled $Tm(n)$, where n is the length of the timeout. Normal timeouts such as these cannot be canceled and must run to completion.

timing mark

A denotation for the time at which an event or message occurs. It is used in sequence diagrams to show how much real time passes between two points in a scenario.

tooltips

Text displayed when the cursor is moved over an interface element. For example, the full path name of an element is displayed in a tooltip when the cursor is over that element.

trace

A dependency that indicates a historical or process relationship between two elements that represent the same concept, without specific rules for deriving one from the other.

tracing

Displays trace messages during program execution.

transient object

An object that exists only during the execution of the process or thread that created it.

transition

A relationship between two states indicating that an object in the first state will perform certain specified actions and enter the second state when a specified event occurs and specified

conditions are satisfied. On such a change of state (first state to second state), the transition is said to *fire*.

A transition can have a trigger, a guard, and an action, which is formatted in the transition label as follows:

`trigger[guard]/action`

Transitions usually consist of at least a trigger and an action. A trigger can be either an event or a triggered operation. A transition can be qualified by a guard condition, meaning that the guard must be true for the transition to be taken.

transition context

The scope in which message data (parameters) are visible. Any guard and action can inherit the context of a transition determining the parameters that can be referenced within it.

trigger[guard]/action

An expression attached to a transition that determines the following following actions::

- ◆ What event will fire the transition (trigger)?
- ◆ Under what condition the transition will be allowed (guard)?
- ◆ What action code will be executed when the transition takes place (action)?

triggered operation

A synchronous communication between objects (between a client object and a server object). It has a body of operation defined by the statechart actions that it triggers, and can return a value.

A triggered operation is a cross between an operation and an event. It is started by another object to trigger a state transition and its body is executed in response to the transition taken. Because it is a synchronous event, the sending object waits for the execution of the triggered operation.

The body of a triggered operation is set in the statechart of the receiving object by the action language associated with a transition. Thus, the body of the same triggered operation can be different based on the state of the object when the triggered operation is launched.

Because its activation is synchronous, an operation can return a value to the client object. To return a value from a triggered operation, use the `REPLY(VARIABLE)` macro inside the body of the triggered operation.

A statechart consumes one event at a time. Until an event is consumed, no subsequent event can be consumed. Because triggered operations inject events (and “wait” until they are consumed), two triggered operations must run sequentially.

type

A stereotype of class used to specify a domain of instances (objects) together with the operations applicable to the objects. A type cannot contain any methods. See also [class](#) and [instance](#). Contrast with [interface](#).

type expression

An expression that evaluates to a reference to one or more types.

Unified Modeling Language (UML)

The UML is a third-generation language for describing complex systems using models. This language allows system architects and engineers to work at a high level of abstraction and to communicate design intent effectively. Using UML models allows system simulation so that errors to be found and corrected early in the development process.

uninterpreted

A placeholder for types whose implementation is not specified by the UML. Every uninterpreted value has a corresponding string representation.

unit

A composite model element stored in its own file that you can check in and out of a CM system. A model element can be made into a unit as long as it can be saved as a separate file. Some elements that can be saved as units are the entire model, packages, classes (in C, objects and object types), any type of Rational Rhapsody diagram, and components. The project, represented by the root node displayed in the browser, is always a unit.

The primary purpose of units is to support collaboration with other developers. You have explicit control over unit files and modification rights (RO versus RW), and you can check unit files in and out of a CM system. You can also compare units using the Rational Rhapsody [DiffMerge tool](#).

unresolved reference

A reference between two configuration items that have become disconnected, possibly by one having been moved to a different workspace.

usage

A dependency in which one element (the *client*) requires the presence of another element (the *supplier*) for its correct functioning or implementation.

use case

The specification of a sequence of actions, including variants, that a system (or other entity) can perform by interacting with actors of the system.

A use case is one way in which a user, or external actor, might interact with a system. For example, the user of a railcar system might want to call a car. This main use of the system can be represented in a use case named “Call Car.” Each use case belongs to a package, as reflected in the browser.

A use case can be referred to from other packages, but can belong to only one package at a time. The same holds true for types.

use case diagram (UCD)

A diagram that shows the relationships among actors and use cases within a system.

UCDs describe a high-level functional analysis of the system. They enable users to specify major system requirements. For example, user requirements for a VCR system might be to set up the system, play back and record video cassettes, set the time, and store channels into memory. These main uses of the system can be represented by the use cases Setup, Playback, Record, SetTime, and AutoProgram. Use cases represent the externally visible behaviors, or functional aspects, of the system.

Actors are added to the UCD to show the external entities interacting with the system.

use case instance

The performance of a sequence of actions being specified in a use case. An instance of a use case.

use case model

A model that describes a system's functional requirements in terms of use cases.

user-defined type

A type defined by a user, not a predefined type. Types allow for meaningful interpretation of fixed-length bit sequences. Common predefined types include integer (int), char (characters), and float (floating-point).

utility

A stereotype that groups global variables and procedures in the form of a class declaration. The utility attributes and operations become global variables and global procedures, respectively. A utility is not a fundamental modeling construct, but a programming convenience.

value

An element of a type domain.

variable

A storage place within an object for a data element. The data element can be a data type such as a date or number, or a reference to another object.

VBA project

A VBA project is a file container for other files and components that you use in Visual Basic to build an application. After all the components have been assembled in a project and code written for it, you can compile the project into an executable file.

Each Rational Rhapsody project is associated with a single VBA project that contains all the VBA artifacts (scripts, forms, and so on) you created within a Rational Rhapsody project. This project file has the name <project name>.vba, located in the same directory with the Rational Rhapsody project file (<project>.rpy). If present, this binary file is loaded with the Rational Rhapsody project and saved when you press the **Save** button in Rational Rhapsody or the VBA IDE.

vertex

A source or a target for a transition in a state machine. A vertex can be either a state or a pseudo-state. See [pseudo-state](#) and [state](#).

view

A projection of a model seen from a given perspective or vantage point that omits entities that are not relevant to this perspective. This could be a picture of existing model information consisting of a diagram or [statechart](#).

Different views display different sets of information. A class comprises the sum of its appearances in all views. The browser shows a complete view of a class, as does the generated code.

view element

A textual or graphical projection of a collection of model elements.

view projection

A projection of model elements onto view elements. A view projection provides a location and a style for each view element.

virtual base class

An indirect descendant inherits only one set of members from a virtual base class through multiple, intermediate inheritances.

virtual instance

An instance group named like a single instance, but actually containing several instances.

visibility

An enumeration whose value (public (+), protected (#), or private (-)) denotes how the model element it refers to can be seen outside its enclosing namespace.

Visual Basic for Applications

Visual Basic for Applications (VBA) is an OEM version of Microsoft Visual Basic integrated as an automation engine into the Microsoft Office family and ultimately wanted for all Microsoft tools.

white-box analysis

This analysis decomposes the system's functions into subsystem components or the internal "logical subsystems." It describes how they relate to each other and to the outside world. A white-box analysis use case is different from a [black-box analysis](#) use case in that the black-box is used only to show the interactions with the outside world. The white box shows interactions both internal and external.

wizard

A small program or macro designed to perform repetitive or simple tasks automatically. See the Rational Rhapsody installation instructions and see [Systems engineering with Rational Rhapsody](#) for examples of Rational Rhapsody wizards.

workspace

A project in Rational Rhapsody that contains a subset of the project artifacts on which a single user works.

Index

Symbols

"And" line 758
#define 999
#endif directive 949
#if...#ifdef...#else...#endif 1000
#ifdef directive 949
#include structure 1489
#pragma directive 949
\$archive keyword 163
\$archiveddirectory keyword 163
\$Arguments keyword 163
\$attribute keyword 164
\$base keyword 164
\$CheckOut keyword 164
\$class keyword 164
\$ClassClean keyword 164
\$cname keyword 164
\$coclass keyword 164
\$CodeGeneratedDate keyword 164
\$component keyword 164
\$ComponentName 164
\$ConfigurationName keyword 164
\$datamem keyword 164
\$DeclarationModifier keyword 164
\$Description keyword 164
\$executable keyword 164
\$FILENAME keyword 165
\$FullCodeGeneratedFileName keyword 165
\$FULLFILENAME keyword 165
\$FullModelElementName keyword 165
\$FullName keyword 165
\$id keyword 165
\$IDInterface keyword 165
\$ImplName keyword 967
\$index keyword 165
\$item keyword 166
\$iterator keyword 166
\$keyname keyword 166
\$label keyword 166
\$log keyword 166
\$Login keyword 166
\$LogPart keyword 166
\$makefile keyword 166
\$maketarget keyword 166
\$member keyword 166
\$mePtr keyword 166
\$mode keyword 166
\$ModePart keyword 167
\$Name keyword 167
\$OMAdditionalObjs keyword 167
\$OMAllDependencyRule 167
\$OMBuildSet keyword 167
\$OMCleanOBJS keyword 167
\$OMConfigurationCPPCompileSwitches keyword 167
\$OMConfigurationLinkSwitches keyword 167
\$OMContextDependencies keyword 167
\$OMContextMacros keyword 168
\$OMCPPCompileCommandSet keyword 168
\$OMCPPCompileDebug keyword 168
\$OMCPPCompileRelease keyword 168
\$OMDEFExtension macro 169
\$OMDllExtension macro 169
\$OMExeExt keyword 169
\$OMFileCPPCompileSwitches keyword 169
\$OMFileDependencies keyword 169
\$OMFileImpPath keyword 169
\$OMFileObjPath keyword 170, 171
\$OMFileSpecPath keyword 170
\$OMImplIncludeInElements keyword 170
\$OMImplExt keyword 170
\$OMInstrumentation keyword 170
\$OMLibExt keyword 170
\$OMLibs keyword 170
\$OMLinkCommandSet keyword 171
\$OMLinkDebug keyword 171
\$OMMakefileName keyword 171
\$OMModelLibs keyword 171
\$OMObjectsDir keyword 171
\$OMObjExt keyword 171
\$OMObjs keyword 171
\$OMROOT keyword 167
\$OMSourceFileList keyword 172
\$OMSpecExt keyword 172
\$OMSpecIncludeInElements keyword 172
\$OMTargetMain object 172
\$OMTargetName keyword 172
\$OMTargetType keyword 172
\$OMTimeModel keyword 172
\$opname keyword 172
\$opRetType keyword 172
\$package keyword 172
\$PackageLib keyword 172

- \$ProgID keyword 172
- \$projectname keyword 172
- \$Property keyword 172
- \$RegTlb keyword 173
- \$RhapsodyVersion keyword 173
- \$rhpdirectory keyword 173
- \$Signature keyword 173
- \$Target keyword 173
- \$target keyword 173
- \$targetDir keyword 173
- \$ThreadModel keyword 173
- \$tlbPath keyword 173
- \$Type keyword 174
- \$type keyword 174
- \$typeName keyword 174
- \$unit keyword 174
- \$VersionIndepProgID keyword 174
- \$VtblName keyword 174
- %s character 108
- .clb files 264
- .cls files 264
- .cmp files 264
- .csv files 236
- .ctd files 264
- .dpd files 264
- .ehl file 265
- .hep files 476
- .msc files 264
- .omd files 264
- .rpw file 265
- .rpy file 264
- .sbs files 264
- .sdo file 711
- .ucd files 264
- .vba file 265
- _auto.rpy file 264
- _bak1.rpy file 264
- _bak2.rpy file 264
- _DEBUG, compiling with 949

Numerics

64-bit targets 916

A

- Accelerator keys 1453, 1455, 1457, 1458
 - Ctrl-R for relations 213
- Accelerators
 - application 1455
 - for code editing 1458
 - in diagrams 1457
- AcceptChanges property 1054, 1057, 1061, 1212, 1214
- Access privileges 255
- Accessor implementation file 939
- Accessors 67, 69
- Acquisition viewpoint (MODAF) 1346

- Action blocks 628, 660
 - creating 628, 660
 - creating subactivities from 629
- Action field 737, 1279
- Action flows
 - completion 663
 - default for flow chart 664
 - join for flow charts 665
 - loop for flow chart 664
- Action language 1277, 1279
 - arithmetic operations 1279
 - assignments 1279
 - basic syntax 1277
 - C code generation 974
 - checking 1280
 - example 1279
 - reserved words 1278
- Action on entry field 729
- Action on exit field 729
- Action pins 648, 650
 - adding to diagram 648
 - displayed in search 650
 - graphical characteristics 649
 - modifying features 649
- Action states 1260
 - defining 1279
- Actions 621, 622, 623, 625, 656, 658, 659
 - block 621, 628, 656, 660
 - blocks 623, 1265
 - creating 625, 659
 - display options 626, 660
 - drawing 622, 658
 - expression 749
 - flow charts 658
 - showing attributes 626, 660
- Active
 - class 770
 - component 860
 - configuration 861
 - project 11, 244, 245
 - state configuration 761
 - thread 1100
- Active Code View 33
- Active code view 33
 - code generation 915
 - edit code in 610
 - line numbers 922
 - show/hide 39
 - window 17, 33
- ActiveStackSize property 1100
- ActiveThreadName property 1100
- ActiveThreadPriority property 1100
- Activities 1226
 - initial flow 1261
- Activity diagrams 4, 5, 157, 621, 1257
 - accept event action 623
 - accept time event 623

- action block 628
- action blocks 623
- action pins 623, 1259
- action states 1260
- action types 1257
- actions 622, 623
- activity flow 624
- activity flows 634
- advanced features 622
- algorithm 621
- browser icon 301
- call behavior 623
- call behaviors 634, 646
- call operation 623, 1259
- code for operations 652
- code generation 652
- code generation limitations 654
- code generation scope 653
- code generation values returned 653
- connectors 636
- create new 577
- creating 44, 623
- creating in SysML 1258
- creating new 409
- decision node 624
- decision nodes 637
- decision points 621
- defining an action 1279
- dependency 624, 1259
- diagram connectors 624
- drawing tools 623
- final activities 631
- flow of control 1257
- fork node 624, 1259
- generating sequence diagram from 1263
- initial flow 624, 635
- IntelliVisor information 466
- join node 624, 1263
- limitations 654
- link wizard 1240
- main behavior 621
- merge node 624
- mode 651, 732
- modifying called behaviors 647
- object node 632
- object nodes 624
- properties for SysML 1258
- Send Action 756
- send activity 624
- send activity state 1260
- setting activity parameters 623, 1259
- setting initial flow 1261
- similarity to flow charts 656
- states 622
- subactivities 624
- subactivity 630, 1261
- swimlanes 624, 1260, 1262
- synchronization bars 638
- SysML behavior interconnections 1257
- system engineering options 1234
- systems engineering drawing icons 1259
- transition labels 623, 636
- transitions 634, 1261
- Activity Final 623, 662
- Activity flows 634, 663
 - creating flow chart 663
- Activity frames 626
- Activity parameters 648, 650
 - graphical characteristics 649
 - modifying 649
- Activity view 1232, 1234
 - consistency checks 1235
 - copy 1234
- ActivityReferenceToAttributes property 653
- Actors 524, 577, 841, 1252
 - attributes 525
 - browser icon 299
 - collaboration diagram 841
 - concurrency 524
 - creating in OMD 577
 - creating UCD 524
 - drag and drop 331
 - drawing 524
 - elements 321
 - external interfaces 1264
 - generate code for 863
 - generating code for 863, 926
 - generating code for UCDs 526
 - limitations on characteristics 927
 - line creating 695
 - owner 525
 - use cases 524, 1254
- Actual Call window 534, 833
- Ada language 2
 - code generation profile 207
 - code generator symbols 1052
 - composite types 106
 - conditions 110
 - roundtripping 86
 - SPARK profile 208
 - static models 113
 - variant record keys 110
- Add New menu 501, 504, 505
 - bottom (groups/categories) portion 502
 - common list portion 501
 - middle portion 501
- Add On products
 - Rational Gateway 1242
 - Rational Rhapsody Gateway 1242, 1243
 - System Architect 1293
- Add To common list option 182
- Adding
 - properties to the common view 182
- Additional keywords (reverse engineering) 1001

- Additional Sources
 - configuration option 864
 - field 851
- AdditionalBaseClasses 965
- AdditionalHelpersFiles property 512
- AdditionalKeywords property 1001
- AdditionalNumberOfInstances property 960
- AddNewMenuStructure property 501, 503, 504, 505
- Address spaces, sending events across 752
- Advanced button 866
- Aggregates property 1376
- Aggregation 559, 560
- Aggregation Kind field 557
- Algorithms 962
 - activity diagram 621
 - flow chart 655
 - sequence comparison 707
- All Include Files option 1004
- All view properties filter 180
- All Views view (DoDAF) 1313
- All Views viewpoint (MODAF) 1345
- Allocations
 - block definition diagram 1267, 1273
 - in SysML 1226
- AlternativeDrawingTool property 495
- Analysis 7
 - black-box 7, 715, 1265
 - field 410
 - mode 672
 - object 8
 - recursive mode 1004
 - requirements 7
 - trade in Harmony 1237
 - white-box 9, 715
- AnalyzeGlobalFunctions property 1017
- AnalyzeGlobalTypes property 1017
- AnalyzeGlobalVariable property 1017
- AnalyzeIncludeFiles property 1004
- Anchored Elements box 369
- Anchors 370, 371
- And state, code generation 733
- Animated browser 1115
- Animated sequence diagrams 1115, 1119
- Animated statecharts 1119, 1124
- AnimateSDLBlockBehavior property 293
- Animation 146, 198, 1079, 1080
 - automatic for sequence diagrams 703
 - automatic scripts 1128
 - black-box 157, 1130
 - breakpoints 1102, 1103, 1104
 - calling operations 692, 867, 1108
 - command bar 1095
 - configurations 1083
 - creating initial instances 1094
 - debugging applications 148
 - ending 1092
 - exception handling 1136
 - highlighting 1124
 - hints 1136
 - in Eclipse 131
 - injecting events 1095
 - instance lines on sequence diagrams 1117
 - instrumentation mode 1084
 - keyboard shortcuts 1455
 - lifeline 1135
 - limitations 1110
 - local host 1086
 - locating ports automatically 1088
 - mainThread 1099
 - messages 1121
 - MicroC target monitoring 21
 - modes 972, 1112
 - overriding 1137
 - panel diagrams 791, 794
 - partial 1089, 1090, 1110
 - port number 196
 - preparing for 147, 1080
 - properties 1130
 - remote targets 1087
 - return values 691
 - running 147, 1080, 1086
 - running without Rational Rhapsody 1141
 - scripts 1126
 - SDLBlock 293
 - selective instrumentation 866
 - sequence diagrams 1116
 - serialization functions 1138
 - settings 1137
 - stand-alone text version 1143
 - statecharts 1124
 - suppress animated sequence diagram messages 1123
 - synchronization with application 1136
 - testing library 1089
 - threads 1098
 - toolbar 40, 1093
 - ViewCallStack variable 198
 - ViewEventQueue variable 198
 - viewing 1113
- Animation tab 32
- AnimationPortNumber 1088
- AnimationPortRange 1088
- AnimSerializeOperation property 1138
- AnimUnserializeOperation property 1140
- Annotations 368, 1052, 1215
 - creating 366
 - Java 1215
 - modeled versus graphical 366
 - symbols 87
 - types 365
- Anonymous instance 954
- API
 - annotations supported by COM 374
 - exporting COM interfaces 1425
 - ports 101

- sending events across address spaces 753
- Applications 148, 299
 - animating steps 1093
 - development cycle 1
 - external 102
 - helper 477
 - monitor and control 1143
 - OSEK21 adaptor 1381
- AR macro 174
- Architect for Software edition 2, 240
 - creating a new project 240
 - primary implementation language 240
 - samples 240
 - using NetCentric profile 275
- Architect for System Engineers edition 239
 - creating a new project 239
- Architect for Systems Engineers edition 2
 - command line switch 1445
 - no development language used 239
 - project profiles 239
 - using NetCentric profile 275
- Architectural design
 - UML 8
- Architectural Design Wizard 1238
- Architecture
 - AUTOSAR 1377
 - block definition diagrams 1265
 - high-level diagram 1265
 - multi-threaded 1
 - service oriented (SOA) 272
 - static 959
- Archive 149, 221
- ARFLAGS macro 174
- Arguments 76
 - comparing 713
 - creating 76
 - displaying 682
 - field 684
 - roundtripping 1054
 - variable length list 935
- Arrange toolbar 448
- Arrows
 - creating 693
 - dependency 573
 - destroying 693
 - drawing 419
 - naming 421
- Artifacts 1322, 1355
- Asian languages 327
- Associate
 - image file with element 336
 - stereotype with element 385
- Association classes 1424
- Association Ends field 554
- Associations 527, 563
 - aggregation 560
 - bi-directional 552
 - block definition diagram 1266
 - block definition diagrams 1266
 - changing underlying 844
 - Complete Relations 569
 - composite 561
 - composition 561
 - consist of 554
 - container 563
 - creating 552
 - directed 558, 559
 - display options 563
 - displaying in the browser 562
 - features 527
 - icon in browser 300
 - implementing 563
 - in collaboration diagram 841
 - modifying features 553
 - navigability 557
 - qualified in OMDs 557
 - roundtripping 1054
 - selecting in OMDs 564
 - UCD 527
- Asynchronous events 681
- ATG 265
- Attribute types 69
- Attributes 67, 68, 69
 - accessor 939
 - action 626, 660
 - actor 525
 - adding to OMD 530
 - classes 67
 - display options 89
 - editing 333
 - initializing 78
 - listed in table views 224
 - mutator 939
 - private 67
 - protected 67
 - protected icon 67
 - reversed 94
 - roundtripping 1054
 - static 71, 934
 - use case 521
 - value 537
- Auto Enlarge 434
- AutoCopied property 378
- Auto-creating instance lines on sequence diagrams 1117
- Auto-indent text 398
- AutoLaunchAnimation property 703
- Automatic
 - animation of sequence diagrams 703
 - ANSI-compliant code generation 1
 - code generation 15, 914
 - diagram population 43
 - instance lines on sequence diagrams 1117
 - refresh rate 1194
 - requirement sequences comparison 9

- save 206, 218
- Automatically show this window check box 616
- Automotive development 1377
 - AUTOSAR profiles 207, 1377
 - C profile 207
 - OSEK21 adaptor 1380
- AutomotiveC profile 207, 1377, 1380
 - code generation 1303
 - stereotypes 1382
- AutoReferences property 379
- AUTOSAR 2, 1377
 - C language only 207
 - import/export 1379
 - profiles 207
- Autosave 218
 - directory 264
 - file 264
- AutoSaveInterval property 218
- AvailableMetaClasses property 490

B

- Back button 39
- Backup 220
 - BackUps property 220
 - directory 264
 - file 264
 - loading 221
 - project 220
- Backward compatibility 927, 976
 - code generation in C 976
 - for pre-3.0 Rational Rhapsody models 930
 - issues 1044
 - profiles 377
- Bar layout
 - Bar variable 199
 - BrowserFloating variable 199
 - BrowserVisible variable 199
 - FeaturesFloating variable 199
 - FeaturesVisible variable 199
 - ScreenCX variable 199
 - ScreenCY variable 199
- BaseNumberOfInstances property 960
- Batch mode 1047
- Behavioral port 103
- Behaviors
 - activity diagrams 621
 - attribute 94
 - classes 85
 - dynamic views 5
 - ports 80
 - time-based 1
- Bidirectional option 1049
- Binding 808
 - connectors 1266, 1273
 - embedded objects 1190
 - layout and view 234
 - SysML value 1226
- Bitmaps 364
 - associating with stereotypes 387
 - transparent background 388
- Black-box
 - analysis 7, 1265
 - animation 1130
 - testing 1134
- Black-box animation 157
- Block definition diagrams 1265, 1268
 - aggregations 1266
 - allocation 1267
 - association 1266
 - binding connector 1266
 - block 1266
 - connector 1266
 - constraint block 1266
 - constraints 1266
 - create package 1266
 - dependency 1266
 - dimension 1267
 - directed associations 1266
 - directed composition 1266
 - drawing tools 1266
 - executing actions 1265
 - flow 1266
 - flow port 1266
 - flow specification 1266
 - graphics in 1268
 - inheritance 1266
 - NetCentric 273
 - part 1266, 1273
 - perform trade analysis in Harmony 1237
 - problem satisfaction 1267
 - rationale 1267
 - reference properties 1264
 - standard port 1266
 - starting point 1225
 - structural properties 1264
 - unit 1267
 - value properties 1264
 - value type 1267
- BlockIsSavedUnit property 258
- Blocks 1226, 1264, 1265
 - action 628
 - behavior 1265
 - constraint 1226, 1266, 1272, 1273, 1274
 - create test bench from 1235
 - Java initialization 969
 - properties 1264
 - Rational Statemate 1385
 - SDL 208, 292
- Bookmark 407
- Border, system 676
- Boundary box 520, 1252
- Bounded relation 961
- Box

- boundary 520
 - drawing 418
 - naming 421
 - selection handles 431
 - Break 1095
 - break command 1148
 - Breakpoints 289, 1101
 - creating 1102
 - deleting 1105
 - deleting tracer 1148
 - enabling 1104
 - reasons for 1103
 - Browse From Here browser 304
 - BrowserIcon property 493
 - Browsers 22, 130, 295
 - active component icon in 299
 - activity diagram icon in 301
 - actor icons 299
 - animated 295
 - Asian language support 327
 - association icon in 300
 - basic icons 299
 - Browse From Here browser 304
 - class icons 299
 - collaboration diagram icon in 301
 - comments icon 301
 - component diagram icon in 301
 - component icons 299
 - constraint icon in 300
 - controlled file icon in 300
 - copy elements in 332
 - copy multiple elements 22
 - create objects in 318
 - create package in 317
 - creating classes 65
 - delete multiple elements 22
 - deleting categories 333
 - deleting items 306
 - dependency icon in 300
 - deployment diagrams icon in 301
 - display modes 296
 - display stereotype or model element 335
 - displayed in Eclipse 130
 - drag and drop elements in 331
 - editing code 331
 - event icon in 300
 - executables icon 299
 - file icon 299
 - filter 21
 - filtered views 302
 - filtering by views 34
 - filtering views 302
 - flow port icon in 300
 - folder icon 299
 - HTML 1428
 - hyperlinks folder icon 299
 - initial connector icon 300
 - keyboard shortcuts 1455
 - link to editor 144
 - locate diagram element in 45
 - menu commands for Harmony 1234
 - multiple projects in 244
 - object model diagrams icon in 301
 - opening 296
 - opening multiple instances 22
 - part icon in 299
 - profile icon in 300
 - profiles 215
 - profiles displayed in 375
 - Rational Rhapsody 295
 - removing elements 333
 - requirement icon in 300
 - requirements diagram icon in 301
 - sequence diagrams icon in 301
 - Settings folder 314
 - state icon in 300
 - statechart icon in 301
 - stereotype icon in 300
 - structure diagram icon in 301
 - superclass icon 300
 - tag icon in 300
 - type icon in 300
 - unit icon 299
 - use case diagram icon in 301
 - use case icon in 300
 - view overridden properties 302
 - viewing swimlanes 644
 - Bubble Knob control 796
 - Build
 - diagram 44
 - failed error 1443
 - Set field 853
 - tab 29
 - target component 916
 - Button Array control 805
 - Button Array tool 795
- ## C
- C language 2
 - adding external file 830
 - animation 1137
 - automotive development 1377
 - automotive profile 207
 - automotiveC profile 1380
 - AUTOSAR 207
 - backward compatibility 976
 - code generation customization 977
 - code generator symbols 1052
 - component file type 852
 - create file model elements icon 530
 - creating a hierarchy of packages 607
 - enumeration types property 112
 - FunctionalC profile 44, 207

- FunctionalC profile diagrams 14
- interfaces 974
- interrupt-driven framework 217
- MicroC profile 21, 208
- optimization for ports 975
- OXF 217
- packages 205
- panel diagrams 792
- partial animation 1089
- ports 974
- preserving comments 1039
- projects in Eclipse 127, 280
- Rational Statemate blocks 1385
- Rational Statemate code generation 1385
- RespectProfile 208
- reverse engineering 985
- reverse engineering legacy code 601
- roundtripping 86
- Send Action code generation 757
- serialization properties 970
- simplifying code generation 979
- Simulink profile 1433
- SimulinkInC profile 208
- statechart serialization 970
- static models 113
- strings in Web-managed devices 1173
- target monitoring 21
- undefined symbols in reverse engineering 1000
- Visual Studio 288
- C++ language 2
 - call stack 1114
 - calling operation calls during animation 1108
 - class implementation 78
 - code generator symbols 1052
 - CodeCentric profile 207
 - component file type 852
 - composite types 106
 - constant 110
 - constructs in reverse engineering 1045
 - dialects in reverse engineering 1002
 - enumeration types property 112
 - for action language 1277
 - functor-based code generation 652
 - inheritance 970
 - library for reverse engineering 1023
 - packages 205
 - panel diagrams 792
 - partial animation 1089
 - preserving comments 1039
 - projects in Eclipse 127, 280
 - RespectProfile 208
 - reverse engineering 985
 - roundtripping 86
 - Send Action code generation 757
 - serialization properties 970
 - Simulink profile 208, 1433
 - SPT profile 277
 - statechart serialization 970
 - static models 113
 - template limitation in Web-managed devices 1173
 - undefined symbols in reverse engineering 1000
 - variables 319
 - Visual Studio 288
- Call behaviors 634
 - in activity diagram 646
- CALL command 1151
- Call Graph diagram 45
- CALL macro 692
- Call operations 623, 1108, 1259
 - in animation 867
 - nodes 622
- Call stack variable 198
- Call stack view 1114
- CALL_INST macro 692
- CALL_SER macro 692
- Called behaviors
 - displaying features 647
 - limitations 647
 - modification 647
- Calls
 - in activity diagrams 623
- Cancel
 - changes in Features window 49
 - timeout 694
- Categories
 - deleting 333
 - mode 296, 1474
- Change
 - applying 48
 - elements 433
 - hyperlink 58
 - line shape 420
 - order of types 115
 - property values 177
- Change to Package option 378
- Change to Profile option 378
- Check
 - for consistency in activity diagrams 1235
 - for static memory allocation 963
- Check Data 1404
- Check model 140
- Check Model tab 26
- Check Model tool 882, 885
 - list of checks 889
 - sample check projects 888
 - user-defined checks 886
- Checks 881, 886
 - activity view 1235
 - swimlane consistency 1234
- Checks tab 882
- Class
 - \$class keyword 164
- Class Type field
 - actor 525

- ClassCentricMode property 672
- ClassCodeEditor 924
- Classes 65, 531, 545, 680, 1235
 - active without statecharts 770
 - aggregated associations 560
 - as containers 326
 - attributes 67, 68
 - base for rapid ports 98
 - behavior 85
 - bi-directional association 552
 - browser 320
 - browser icons 299
 - code structure 932
 - collaboration diagram for multiple objects 840
 - composite 530, 546, 830
 - composite associations 561
 - constructor arguments 76
 - constructors 76, 77
 - converting to objects 535
 - create test bench from 1235
 - creating 65, 545
 - creating in OMD 545
 - default name 65
 - defining features 65
 - deleting 90
 - dependencies between 572
 - derivation 84
 - destructors 79
 - directed association 558
 - display name 88
 - display options 87
 - drag and drop 331
 - editing code 86
 - editing multiple 333
 - files 264
 - functor 652
 - Generic Class 122
 - header file 932
 - importing as a type 1023
 - in collaboration diagram 840
 - inheriting from external 549
 - instances 1125
 - mapping to types and packages 1008
 - modeling for reverse engineering 1023
 - naming guidelines 251
 - nested 67, 126
 - nested roundtripping 1054
 - object node association 633
 - opening the main diagram 87
 - operations 71
 - ports 80
 - primitive operations 72
 - properties 83
 - reactive 773
 - reactive and refining the hierarchy 778
 - receptions 73
 - reference 1018
 - regular 66
 - relations 80
 - relations, show all 82
 - removing 90
 - roundtripping 1050, 1054
 - roundtripping supported modifications 1053
 - solutions 1237
 - structured 457
 - superclass 300, 341, 545, 548
 - swimlane association 642
 - tags 83
 - Template Class 122
 - template instantiation 66
 - templates 66, 122
 - triggered operations 75
- Classifier role 840
- Classifier role names 679
- Classifier roles 677
- ClassIsSavedUnit property 258
- CLASSPATH 997
- Cleaning
 - delete redundant code files 926
 - delete sequence diagram messages property 672
 - old objects 917
 - redundant source files 926
- CleanupRealized property 672
- Clipboard, consistency check results to 1235
- coclass
 - keyword 164
- Code 6
 - active view 922
 - associated with element 144
 - clean 917, 921
 - compilation errors 920
 - displaying 33
 - DMCA 143
 - documentation system 944
 - documentation systems 944
 - editing 921
 - editing from a diagram 143
 - editing from browser 331
 - editing in Eclipse 143
 - editing with an external editor 925
 - editor, Eclipse 143
 - editor, internal 395
 - elaborative generation 910
 - exporting to Eclipse 135
 - external 994
 - for actors 526
 - for classes 86
 - generate for actors 863
 - generated 921
 - generating 141, 142
 - generating for files 593
 - generating for relations 611
 - generation 85
 - generation for actors 926

- generation of individual elements 919
- generation results 920
- inlining 786
- keyboard shortcuts 1455
- legacy 9, 994
- line numbers 922
- renaming in user 219
- return from command-line 1443
- reverse engineering in Eclipse 136
- roundtripping 1048
- roundtripping class code 86
- source import 136
- structure of generated 932
- toolbar 37
- translative generation 910
- viewing 921
- wrapping with `#ifdef #endif` 949
- writer 980
- Code centric 206
- Code check box 346
- Code editor, Eclipse 143
- Code editor, external 395, 925
- Code generation 15, 141, 142, 909
 - activity diagrams 652
 - ANSI-compliant 1
 - automatic 914
 - automotiveC 1303
 - backward compatibility 976
 - change order of operations/functions 942
 - component diagrams 928
 - customizing C 977
 - customizing using properties 977
 - customizing using rules 977
 - dynamic model -code associativity 914
 - ExternalGenerator variable in .ini file 198
 - flow charts 666
 - for links 570
 - for Send Action 756, 757
 - for templates 126
 - forcing complete 913
 - guidelines 914
 - incremental 912
 - JavaAnnotations 1220
 - limitations 976
 - macros 1043
 - ModelCodeAssociativityMode variable 198
 - option 1049
 - options 142
 - restrictions in activity diagrams 654
 - restrictions in flow charts 666
 - simplified models 979
 - stereotype-based 968
 - stop 915
 - transformation phase 977
 - writing phase 977
- Code respect 1063
 - activating 1064
 - reverse engineering 909, 1037, 1044
 - roundtripping 909, 925, 1047, 1058, 1063
 - SourceArtifact 1065
 - where code respect information is defined 1065
- Code writer 980, 981
- Co-debugging 290
- CodeCentric
 - profile 207
- Code-centric mode 1069
- CodeGeneratorTool property 979, 1039
- CodeTEST 865
- Collaboration diagrams 4, 5, 837
 - actors in 841
 - browser icon 301
 - changing underlying association 844
 - classifier role 840
 - creating 44, 410
 - creating links in 842
 - creating messages in 845
 - define for core cases 9
 - editor 13
 - files 264
 - IntelliVisor information 464
 - link 841
 - message numbering 837
 - messages 844
 - multiple objects 840
 - specifying behavior 7
 - tools 839
- CollectMode property 1042
- Color 395
 - coding in editor 395
 - default settings 396
 - in sequence diagram comparisons 709
 - set diagram fill 417
- Command line 1441, 1442, 1444, 1446
- Command Prompt tool 1095
- Command-line interface 1439
 - commands 1441, 1446
 - exiting 1442
 - interactive mode 1440
 - interactive switch 1440
 - methods of operation 1439
 - order of commands 1442
 - path names 1441
 - quotation marks in commands 1441
 - return codes 1443
 - scripts using commands 1442
 - socket mode 1440
 - switches 1441, 1444
 - syntax 1441
- Commands 1441, 1446
 - examples for running Rational Rhapsody 1443
 - order of 1442
 - tracer 1145, 1148
- Comment specification search option 346
- Comments 365, 1039

- added to components 859
- adding to properties files 189
- anchoring 370
- browser icon 301
- floating 1039
- in tracer commands 1145
- limitations 1040
- preserving (reverse engineering and roundtripping) 1039
- reverse engineering 1039
- specification 346
- Common drawing tools 41
- Common view
 - changing 182
- CommonList property 505
- Communicate with ports 102
- Compare
 - features 51
 - Names and Values option 714
 - Names Only option 714
 - sequence diagram excluding a message 713
 - sequence diagram instance groups 715
 - sequence diagram's algorithm 707
 - sequence diagrams 711
- Compartments 530
 - display stereotype for elements in list 454
 - show label 88
- Compatibility
 - automatic settings 377
 - profiles for backward 377
- Compiler
 - messages 916
 - specific keywords 938
- Compiler Switches field 853
- Compilers 31
 - adding keywords 949
 - MATLAB MEX 1436
- Complete Relations
 - associations 569
- Complete Relations option 461
- Complex parameters
 - overriding 1137
- ComplexityForInlining property 787
- Component diagrams 4, 5, 848
 - browser icon 301
 - code generation 928
 - components 850
 - creating 44, 410
 - dependency 857
 - drawing tools 849
 - editor 14
 - elements 850
 - files 852
 - files extension 264
 - folder 855
 - IntelliVisor information 464
- Component instances
 - deployment diagrams 875
 - modify features 877
- Component interface 857
 - creating 857
- Component Type field
 - component instance 877
- Component View 1409
- ComponentIsSavedUnit property 258
- Components 205, 315, 850
 - \$ComponentName keyword 164
 - active 299, 860
 - adding file to 852
 - browser icons 299
 - building 916
 - comments added to 859
 - configuration 315
 - controlled files added to 859
 - creating 850
 - creating interface 857
 - deleting composite 955
 - directory 265
 - file 264
 - files 315
 - icon 850
 - importing from Simulink 1432
 - interface 847
 - options 859
 - view 302
- Components (subsystems) 1081, 1286
 - creating 1081, 1286
 - creating configuration 1083
 - features 1287
 - setting features 1082
- Components-based development in C 973
- Composite association 561
- Composite class 457, 530, 546, 830
- composite stereotype 1249
- Composite type 106
 - properties 111
- Composition association 561
- Composition option 557
- Compound transition 738
- Concurrency field
 - actor 524
 - block 534, 832
- Concurrent Versions System (CVS) 132
- Condition mark 693
- Configuration 1083, 1174
 - \$ConfigurationName keyword 164
 - time model setting 172
- Configuration for Simulink 1436
- Configuration management 355
 - not for project list files 250
- Configuration management (CM) 149
 - DiffMerge tool 153
 - performing operations in Eclipse 152
 - preferences 152

- Rational Rhapsody and Eclipse 150
- Configuration Management tab 32
- Configurations 147, 315, 860, 1287
 - active 136, 861
 - active Eclipse 141
 - active state 761
 - automotiveC stereotypes 1382
 - checks 865
 - clean 921
 - component 859
 - convert to Eclipse 281
 - creating animation 1083
 - default 1287
 - Eclipse 280
 - features 862
 - generate main option 861
 - generating 861
 - initialization search option 346
 - menu 861
 - partial animation 1090
 - regenerating 913
 - scope 863
 - Set as Active option 861
 - settings 864
 - Web components property 1174
 - Web-enable 1288
- Conflict transition 768
- Conform 1226, 1228
- Connect
 - objects using ports 102
 - ports 97
 - to filtered views 1177
 - to model from the Web 1176
 - to model from the Web, troubleshooting 1177
- Connectors
 - activity diagram 636, 638
 - binding 1266, 1273
 - block definition diagram 1266
 - condition 637, 665
 - default 750
 - diagram 624
 - flow charts 664
 - history 727
 - initial in statecharts 750
 - junction for statecharts 727
 - statechart 728
 - termination 727
- Consists of field 554
- Constant modifier 70
- ConstantVariableAsDefine property 999
- Constraint specification search option 346
- Constraints 320, 365, 1228
 - anchoring 370
 - binding 1276
 - block 1266, 1273
 - block definition diagram 1266
 - blocks 1272, 1274
 - browser icon 300
 - editing text 366
 - finding references 370
 - parameters 1273, 1276
 - properties 1274, 1275
 - property 1273
 - specification supports Asian languages 346
 - SysML 1226
- Constructor Arguments window 76
- Constructors 76
 - Features window 77
 - implementation file 939
 - roundtripping 1054
- Constructs added to Model option 1035
- Constructs Analyzed option 1035
- Container 563
- Containment by value 962
- Content window 23
- Contract tab 94
- Contracts 80
 - specifying 94
- Control
 - model from the Web 1183
 - point 434
- Control Properties window 809
- Controlled files 349
 - browse to 351
 - configuration management 355
 - creating 350
 - features 352
 - limitations 357, 359
 - tags 354
 - troubleshooting 356
- Convert
 - class to object 535
 - external elements 608
 - file 592
 - note to comment 370
 - object types 535
 - package to profile 378
 - profile to package 378
- Conveyed information 582
- Copy
 - actor menu option 577
- Copy with Model 447
- Copying 246
 - between Features windows 51
 - element in browser 332
 - elements 445
 - elements to other projects 247
 - format from one element to another 440
 - instance line menu option 433
- CORBA 10
 - check 889
 - inheritance 613
 - limitation 115
- CP macro 174

- CPP_EXT macro 174
- CPPCompileCommand property 917
- CPU 873, 874
- Create Reference Sequence Diagram option 699
- CreateDependencies property 1004
- CreateReferenceClasses property 1017
- Custom help file 379
- CustomHelpMapFile property 381, 382
- CustomHelpURL property 380, 382
- CustomizableTableAndMatrixLayoutsPkg 1362
- CustomizableTableAndMatrixViewsPkg 1366
- CustomizedStereotypesPkg 1363
- Customizing
 - Add New menu 501, 504
 - C code generation 977
 - C rules with RulesComposer 981
 - code generation rules with RulesComposer 981
 - keyboard mappings 400
 - linking to helper applications 477
 - navigation to your Web GUI 1182
 - new diagrams 493
 - profile 486
 - Rational Rhapsody Web server 1193
 - Tools > Diagrams menu 503
 - Web interface 1184
 - with properties 155
- Cut option
 - actor menu 577
 - instance line menu 433
- CVS 132

D

- Data
 - checking 1277
 - export to Excel 236
 - flow 674
 - manage from table or matrix 236
 - query in table 226
 - vendor-neutral sharing 1429
- Dataflows 696, 1123
- DataTypes property 1023
- Debugger 290
- Debugging 146, 1080
 - animated applications 148
 - perspective 146
- Decision node 727
- Decision nodes 637, 665
- Decision points 621, 656
- DeclarationModifier keyword 164
- Decomposed field
 - instance line 678
- Decomposition 700
 - limitations 700
- Default flows
 - flow chart 664
- DefaultDirectoryScheme property 258
- DefaultProvidedInterfaceName property 98
- DefaultReactivePortBase property 98
- DefaultReactivePortIncludeFile property 98
- DefaultRequiredInterfaceName 98
- Define View page 1180
- Defined In field 369
 - actor 525
- Defined symbol 998
- DEGREES_PER_RADIAN variable 319
- Deleting
 - after search 212
 - anchors 371
 - anonymous instances 955
 - breakpoint 1105
 - breakpoint tracer 1148
 - categories 333
 - classes 90
 - composite components 955
 - elements from file 854
 - elements from model 451
 - from model 306
 - hyperlinks 59
 - instance groups 717
 - message groups 724
 - old objects 917
 - redundant code files 926
 - redundant source files 926
 - reference classes 1018
 - remarks 373
 - sequence diagrams 706
 - stereotypes 388
 - swimlane dividers 644
 - swimlane frames 645
 - tags 393
 - using the Edit menu 333
- Dependencies 320, 528, 572, 573
 - across packages 316
 - activity diagrams 624, 1259
 - adding stereotype 1256
 - arrow 573
 - between remarks 367
 - block definition diagrams 1266
 - browser icon 300
 - component diagrams 847, 857
 - creating 573
 - deployment diagrams 878
 - friend 976, 1058
 - from Includes option 1012
 - in UCD elements 528
 - Link wizard 1240
 - modifying features of 575
 - new 859
 - parametric diagrams 1273
 - relationships 1248
 - requirement diagrams 1247
 - statecharts 728
 - structure diagram 835

- system engineering 1264
 - system engineering requirements 1248
- Dependencies Linker (MODAF) 1372
- Dependency
 - on library component 171
- Depends On field 575
- Deployment diagrams 4, 871
 - assigning a package 880
 - component instances 875
 - creating 410
 - dependencies 878
 - drawing tools 872
 - editor 14
 - elements 873
 - files 264
 - icon in browser 301
 - IntelliVisor information 465
 - node owner 874
 - nodes 873
 - UML 5
- Derivation 84
- Derivation in requirement diagrams 1247
- derive stereotype 1249
- Derived scope 863
- Descendants
 - include in views 226, 230
- Description tab 56
- Descriptions
 - adding hyperlinks in 55
 - mass edit 333
 - search option 346
- DescriptionTemplate 945
- DescriptionTemplate property 1209, 1211
- Design
 - architectural 8
 - basic requirements 7
 - details of 9
 - mechanistic 9
 - mode 672
 - option 410
 - requirements for SysML 1249
 - SysML requirements in use cases 1255
- Designer for System Engineers edition
 - command line switch 1445
 - project profiles 237
- Designer for Systems Engineers edition 2, 237
 - samples 238
 - structure diagrams 238
- Destination of transition 735
- Destructors 79
 - implementation file 939
 - roundtripping 1054
- Developer edition 2, 3, 237
 - using NetCentric profile 275
- Development
 - analysis phase 7
 - design phase 8
 - environments 127
 - implementation phase 9
 - methodology 7
 - parallel 149
 - phases of 7
 - testing phase 9
- Devices
 - managing remotely 1171
 - setting name in Web pages 1193
 - Web-enabled 1171, 1177, 1178
 - Web-enabled, adding files to model 1184
 - Web-enabled, connecting to from the Web 1176
 - Web-enabled, controlling 1183
 - Web-enabled, customizing the GUI 1184
 - Web-enabled, Define View page 1180
 - Web-enabled, name/value pairs 1183
 - Web-enabled, Personalized Navigation page 1181
 - Web-enabled, setting as 1172
 - Web-enabled, using properties 1175
 - Web-enabled, viewing 1183
- Diagram connectors 638
- Diagram editor
 - grid 428
 - properties for 417
- DiagramIsSavedUnit property 254, 258
- Diagrams 6, 409
 - accelerators 1457
 - active resizing 456
 - activity 4, 5, 44, 621, 1257
 - adding 134, 322
 - adding elements 322
 - adding new customized 493
 - adding remarks to 365
 - automatically populating 412
 - AUTOSAR 1378
 - block definition 273, 1225, 1265
 - Build 44
 - Call Graph 45
 - close all 220
 - collaboration 4, 5, 44, 837
 - comparing 55
 - component 4, 5, 44, 847
 - connector 728
 - creating 43, 409
 - deployment 4, 5, 871
 - drawing area 130
 - editing 6
 - editing code from 143
 - exporting as images 364
 - external block 1264, 1265
 - File 45
 - flow chart 4, 45, 655, 658
 - for FunctionalC profile 44
 - fully constructive 6
 - high-level architecture 1265
 - in reports 153
 - internal block 1235, 1264, 1269

- locate element in browser 45
- locating element in IDE 279
- locating elements 323
- Message 45
- naming 43
- navigator 459
- object model 4, 5, 44, 529
- OpenDiagramsWithLastPlacement property 263
- opening 411
- output to UNISYS extensions format 1426, 1428
- OV-1 High Level Operational Graphic 1325
- OV-2 Operational Node Connectivity 1325
- OV-4 Organizational Relationships 1325
- OV-5 Operational Activity 1325
- OV-6a Operational Rules Model 1325
- OV-6c Operational Event-Trace Description 1325
- OV-6c Operational State Transition Description 1325
- panel 44, 791
- parametric 1272, 1275
- partially constructive 6
- populate 1245
- printing 360
- Project Overview 1325
- requirements 1245
- SA DoDAF import 1293
- saving 256
- scaling 457
- sequence 4, 5, 44, 669, 1263
- set fill color 417
- standard toolbar 41, 455
- statecharts 4, 5, 44, 725, 726, 1286
- structure 4, 5, 44, 238, 239, 829
- SV-10b System State Transition Description 1326
- SV-10c System Event-Trace Description 1326
- SV-11 Physical Schema 1326
- SV-2 System Communication Description 1326
- SV-4 System Functionality Description 1326
- SV-8 System Evolution Description 1326
- SysML from SA data 1296
- systems engineering 1223
- test context 1235
- UML 4, 205
- use case 4, 5, 44, 517, 1242, 1251
- view 302
- DiagramsToolbar property 494, 496
- Dialect 1002
- DiffMerge tool 149, 153, 266
 - annotations 374
 - hyperlinks 55
 - no project list support 250
 - supports action pins 650
 - supports activity parameters 650
 - templates 122
 - version conflicts 1302
- Digital Display control 801
- Digital Display tool 795
- Directed association 558, 559
 - creating 558
- Directed composition 1266
- Direction field
 - flow 581
- Directories
 - autosave 264
 - backup 264
 - component 265
 - containing reference classes 1019
 - DefaultDirectoryScheme property 258
 - flat structure 258
 - hierarchical structure 259
 - project 264
 - root for RE 1014
 - saving units in separate 258
 - structure in RE 1014
- Directory
 - \$rhpdirectory keyword 173
 - \$targetDir keyword 173
 - field 851, 864
 - structure 258
- Dismiss
 - IntelliVisor 462
- Display
 - associations 562
 - browser modes 296
 - code 33
 - messages-to-self 1130
 - port interfaces 95
 - stereotype in compartment lists 454
 - stereotype of element in browser 335
- Display options
 - actor menu 577
 - annotations 372
 - attributes 89
 - class name 88
 - default options 445
 - Enable Image View 1268
 - equations 1276
 - file 588
 - flow menu 583
 - general selections 88
 - operations 89
- DisplayMessagesToSelf 1130
- DisplayMode property 296
- Distributed team 149
- Dividers, swimlane 643
- DLL_CMD macro 174
- DLL_FLAGS macro 174
- DMCA 143, 1049
 - mode 1049
 - roundtripping 87
- Docking the Features window 50
- Documentation note, converting to comment 370
- Documentation system
 - sample 944
- DoDAF 2, 1311, 1322

- All Views 1313
- architectural model 1331
- artifacts 1322
- compared to MODAF 1341
- creating a project 1322
- helpers 1335
- importing SA diagrams 1293
- limitations 1333
- Operational view 1312
- OV-1 High Level Operational Graphic 1325
- OV-2 Operational Node Connectivity 1325
- OV-3 matrix 1329
- OV-4 Organizational Relationships 1325
- OV-5 Operational Activity 1325
- OV-6a Operational Rules Model 1325
- OV-6c Operational Event-Trace Description 1325
- OV-6c Operational State Transition Description 1325
- OV-7 Logical Data Model 1325
- profile 207, 1311
- Project Overview 1325
- reports 1321, 1331
- SA encyclopedia 1295
- setup packages 1320
- SV-1 System Interface Description 1325
- SV-10a Systems Rules Model 1326
- SV-10b System State Transition Description 1326
- SV-10c System Event-Trace Description 1326
- SV-11 Physical Schema 1326
- SV-2 System Communication Description 1326
- SV-4 System Functionality Description 1326
- SV-8 System Evolution Description 1326
- System Architect (SA) diagrams 1293
- Systems view 1313
- tags 1327
- Technical view 1313
- troubleshooting 1334, 1337
- utilities 1318
- verifying installation 1334
- views 1312
- Domain checks 883
- Domain Specific Language 1322
- Domain Specific Language (DSL) 2, 1322, 1355
- DOS 1141
- doStep() function 1433
- Doxygen
 - template-based comments 945
 - using 944
- Drag and drop in browser 331
- Drawing
 - arrows 419
 - boxes 418
 - elements 417
 - mode 417
 - state 728
- Drawing area 23
 - displayed in Eclipse 130
- Drawing tools 17

- DrawingShape property 495
- DrawingToolBar property 493, 496
- DrawingToolIcon property 493, 494
- DrawingToolTip property 494
- DSL 1322, 1355
- dxlapi.dll 1392
- Dynamic behavior views 5
- Dynamic model-code associativity 914, 1049
 - code generation 914
 - roundtripping 87
- Dynamic Model-Code Associativity (DMCA) 143
- Dynamic Object Oriented Requirements System (Rational DOORS) 1391

E

- Eclipse 127, 129, 280
 - animation 146
 - as IDE 279
 - build project 145
 - code editor 143
 - configurations 141
 - confirming Rational Rhapsody Platform Integration 128
 - content management operations with Rational Rhapsody 150
 - create IDE project 141
 - debug project 146
 - Developer edition 2
 - disassociating project from Rational Rhapsody 287
 - DMCA 143
 - Dynamic Model-Code Associativity (DMCA) 143
 - edit Rational Rhapsody project with 286
 - export Rational Rhapsody code 135
 - importing projects into Rational Rhapsody 281
 - importing Rational Rhapsody units 136
 - locating Rational Rhapsody code in 286
 - Model browser 150
 - open existing configuration 286
 - performing CM operations 152
 - perspectives 129, 141
 - Platform Integration 127
 - projects 133
 - properties 285
 - Rational Rhapsody Platform Integration 127
 - reports 153
 - repository 128
 - reverse engineering 136
 - reverse engineering source code 136
 - Rhapsody Debug perspective 130
 - Rhapsody Log 142
 - Rhapsody Modeling perspective 129
 - Rhapsody perspectives in 129
 - Select with Descendents in Unit View 150
 - sharing a Rational Rhapsody model 151
 - source code import 136
 - Unit View 150

- viewing code 144
- work area 133
- Workflow Integration 127
- Edit Code option 577
- Edit Configuration Main File option 861
- Edit Makefile option 861
- Edit menu
 - Complete Relations option 461
 - Format option 436
 - selecting elements 430
- Edit Type Order option 116
- Editable list 68
- Editing
 - code 921
 - constraint text 366
 - diagrams 6
 - elements 433
 - features 216
 - hyperlink 58
 - implementation code 616
 - multiple elements 333
 - Rational Rhapsody project 215
 - remark text 368
 - text 452
 - undo/redo 216
 - using mouse 405
- EditorCommandLine property 855, 925
- Editors
 - accelerators 1458
 - associating files with 924
 - collaboration diagram 13
 - component diagram 14
 - deployment diagram 14
 - drag and drop to 331
 - Eclipse 143
 - graphic 327
 - object model diagrams 13
 - opening 86
 - Rational Rhapsody internal 395
 - Rational Rhapsody internal code editor 395
 - selecting 855
 - sequence diagram 13
 - set scope 921
 - show in browser feature 144
 - statechart 13
 - use case diagram 13
 - XML for SA map 1293
- Elements
 - constraint 1228
- Elaborative code generation 910
- Element types 489
- Elements 205, 215, 313
 - adding 134, 215, 313
 - adding hyperlinks 55
 - adding points to 420
 - adding to file 853
 - arranging 448
 - associated with code 144
 - associating with image file 336
 - associating with stereotype 385
 - based on new term stereotypes 386
 - cell types in matrix views 228
 - changed to units 254
 - changing common operations 433
 - changing in Features window 48
 - changing the format 436
 - classes 320
 - code for external 610
 - code generation 919
 - comment 1228
 - component diagram 850
 - conform 1228
 - constraints 320
 - copy in the browser 332
 - copying 247, 445
 - create for customized diagram 494
 - defining tag for 391
 - deleting from model 451
 - deleting using the Edit menu 333
 - dependencies 320
 - dependency 1228
 - deployment diagram 873
 - diagram editing 433
 - diagrams 322
 - display stereotype in browser 335
 - drawing 417
 - editing in component diagram 854
 - editing multiple 333
 - events 320
 - exposing on the Web 1174
 - external 599, 608, 994
 - external and creating by modeling 605
 - external and generating code 611
 - external source path 609
 - files 321, 587
 - finding usage 342
 - functions 318
 - generating code for relations 611
 - graphical 336
 - identification 326
 - in profiles 207
 - labels 327
 - locate from code 923
 - locating in the browser from the code editor 144
 - locating on diagram 323
 - making a unit 255
 - mapped to the folder field 856
 - matrix view of 230
 - moving in browser 331
 - moving in graphic editors 435
 - new in block diagrams 1228
 - NewTerm stereotype 53
 - nodes 321
 - non-rectangular 431

- objects 318
- operations on a group 22
- package 205
- package types 316
- paths 326
- pinned features' display 50
- problem 1228
- project 205
- rationale 1226, 1228
- realization 1228
- realizing 617
- receptions 73
- receptions, browser icon 320
- references 342
- refinement 1226, 1228
- removing from view 451
- removing points from 420
- removing with browser 333
- renaming 332
- renaming new 314
- re-ordering in browser 334
- resizing 434
- saved as units 246
- search 348
- searching 137, 345
- searching for 210
- selecting 430
- selecting multiple 431
- selecting to import (Rational Rose) 1410
- selection handles 431
- Send Action 756
- set display default options 445
- set element size as default size 445
- set formatting 445
- setting as Web-manageable 1172
- show in browser from editor 144
- special characters in names 251
- stereotypes for SysML 1226
- SysML 1293
- table and matrix views 222
- types 320
- units 253
- usage 342
- use cases 321
- variables 319
- viewpoint 1228
- viewpoints 1226
- views 1226
- Elements box 853
- ElementsMap.xml 1298, 1299
- Else branch 760
- Embeddable object 1190
- Embedded Coder License (ERT) 1433
- EMF image format 364
- Enable Docking by Drag option 51
- Enable Operation Calls field 867
- EnableMultipleAnimation 1088
- End1 and End2 tabs 556
- EnterExit points 763
 - Updating 764
- EntryPoint property 290
- Enumerated type
 - creating 107
 - reverse engineering 1038
- Enums, Java 120
- Environment
 - field 853
 - Recent file list 200
 - variable used by Rational Rhapsody 196
 - variable using with reference units 260
- Environment settings
 - Rational Rhapsody Affinity 197
- Environment Settings group box
 - configuration 865
 - files 853
- Equations 1276
- Error
 - checking 920
 - parsing 1031
- ERT (Embedded Coder License) 1433
- Event Generator 1106
- Events 320, 687
 - accept event action in systems engineering 1259
 - accept event actions 623
 - accept time 623
 - adding operations to 744
 - allocated to subsystems 1238
 - browser icon 300
 - class hierarchy 742
 - destruction 675
 - Features window 329
 - generating in animation 1106
 - generating, gen method 743
 - generating, tracer 1153
 - generating, using friendship 743
 - History list 1107
 - history list file 265
 - in interrupt handlers 961
 - injecting 1095
 - internal 743
 - name in notation 757
 - naming conventions 252
 - pooling 960
 - private 743
 - queue 743
 - queue view 1114
 - receptions 73, 75
 - roundtripping 1055
 - semantics 743
 - Send Action 756
 - Sending across address spaces 752
 - statechart 751
 - triggers 742
 - usage 742

-
- ExcludeFilesMatching property 992
 - Executable
 - extension 169
 - files to include 167
 - language 1258
 - running 918
 - ExecutionModel property 1303
 - Exiting command-line interface 1442
 - Explicit
 - object 531
 - scope 863
 - Exporting 1425
 - diagrams as images 364
 - files 135
 - labels 1398
 - models 1426
 - projects 1400
 - Rational Rhapsody model files 1428
 - to Rational DOORS 1391, 1397
 - VBA macros 485
 - ExportPictures property 1398
 - extend stereotype 1249
 - Extensions
 - executables 169
 - libraries 170
 - object files 171
 - of implementation files 170
 - specification files 172
 - External
 - checks 881, 886
 - class, inheriting from 549
 - code editor 925
 - code writer 980
 - files 590
 - hyperlink 55
 - External block diagrams 1264, 1265
 - External code editor 925
 - External code writer 980
 - External elements 599
 - accessing the code 610
 - converting 608
 - creating by modeling 605
 - creating in pre-V5.2 models 604
 - generating code 611
 - limitations 612
 - relations, generating code 611
 - viewing the source path 609
 - visualization 994
 - External modeling 605
 - F**
 - Favorites 11, 306
 - Favorites browser 11, 306
 - creating list 308
 - folders 309
 - hierarchical structure 309
 - limitations 312
 - removing items 311
 - re-ordering items 310
 - showing/hiding 307
 - toolbar 40, 307
 - Features window 48, 130, 156, 177
 - action in activity diagram 625
 - action in flow chart 659
 - applying changes 48
 - Asian language text 327
 - attributes 68, 69
 - buttons 50
 - cancelling changes 49
 - changing property values 185
 - classes 65
 - comparing elements 51
 - component instance 877
 - configuration 862
 - constructors 77
 - copying text from 51
 - define Send Action 756
 - displaying properties 49, 177
 - docking 50
 - edit table and matrix data 222
 - editing 216
 - editing multiple elements 333
 - event 329, 687
 - file 590
 - Filter Properties 181
 - flow 581
 - Flowitem 584
 - Functions tab 591
 - General tab 49, 66, 93
 - hiding tabs 53
 - Instrumentation Mode 147
 - messages for associations 843
 - objects 534
 - opening 48
 - opening multiple instances 51
 - Operations tab 71
 - overridden properties 183
 - package 547
 - pinned mode 50
 - port 93
 - Ports tab 80
 - property search 178
 - receptions 75
 - Relations tab 80
 - tag 390
 - toolbar 50
 - undocking 51
 - Variables tab 590
 - Fields, search in 346
 - File diagrams 45, 1489
 - File extensions 134
 - File Type field 852
 - FileIsSavedUnit property 258
-

- FileName property 549
- Files 587, 852
 - \$FullCodeGeneratedFileName keyword 165
 - \$OMImplExt keyword 170
 - .hep 476
 - .rpy 264
 - .sbs 378
 - .sdo 711
 - .wsdl 276
 - adding 134
 - adding elements 853
 - adding text 854
 - adding to component 852
 - associating with an editor 924
 - autosave 264
 - backup 264
 - class 264
 - collaboration diagram 264
 - component 264
 - component diagram extension 264
 - connecting 592
 - controlled 859
 - controlled icon in browser 300
 - converting 592
 - creating 588, 852
 - delete redundant code 926
 - deleting elements 854
 - deployment diagram 264
 - display options 588
 - element 854
 - events history list 265
 - examining exported XMI 1428
 - Excel .csv 236
 - excluding from reverse engineering 992
 - export 135
 - extensions 264
 - external features 590
 - Features window 590
 - filesTable.dat 264
 - filtering out types 134
 - folders 855
 - FS and RES 1189
 - functions 591
 - generating code for 593
 - header 1005
 - header structure 932
 - implementation 321, 331, 938, 1054
 - imported shown in browser 299
 - in components 315
 - include for rapid ports 98
 - include in executable 167
 - include in reverse engineering 1003
 - list analyzed for reverse engineering 1007
 - load log 265
 - object 171
 - object file extensions 171
 - object model diagram 264
 - ownership 589
 - package 264
 - paths 590
 - project 264
 - project list 250
 - PRP including other 191
 - reverse engineering 1009
 - reverse engineering log 265
 - ReverseEngineering.log 197, 265
 - rhapsody.ini 196, 250, 1088
 - save log 265
 - SD comparison options 711
 - sequence diagram 264
 - server 1186
 - specification 321, 331, 1054
 - specification extensions 172
 - store.log 265
 - structure diagram 264
 - support for add-on tools 595
 - target for hyperlinks 56
 - types 853
 - types of controlled 352
 - unit 255
 - use case diagram 264
 - variables 590
 - VBA project 265
 - Windows 352
 - workspace 265
 - WSDL 208, 272
 - WSDLDiagrams
 - block definition 273
- Files property (for reverse engineering) 991
- filesTable.dat file 264
- Fill color set 417
- Filtering properties 180
- Filtering tab 1016
- Filters
 - browser views 34, 302
 - model view 1177
- Final activities 631
 - activity diagram 631
 - creating 631, 662
 - flow chart 662
- Final states 650
- Find element usage 342
- Fixed relation 961
- FixedPoint profile 207
- Fixed-point Variables 117
- Flat mode 258, 1490
 - browser display 296
- Flip Right/Left options 640
- Floating comments 1039
- Flow 578, 579
 - adding information element to 585
 - conveyed information 582
 - creating 579
 - displaying the keyword 580

- features 581
 - menu commands 583
 - sample 579
 - Flow charts 4, 655
 - action block 660
 - actions 658
 - activity flows 663
 - algorithm 655
 - code generation 666
 - code generation limitations 666
 - connectors 664
 - creating 45, 657, 658
 - decision points 656
 - drawing tools 657
 - final activity 662
 - limitations 666
 - Send Action 756
 - similarity to activity diagrams 656
 - Flow Ends field 581
 - Flowitem 583, 584
 - flowKeyword 580
 - Flowports 597
 - and Simulink 1435
 - atomic 597
 - attributes 598
 - block definition diagram 1266
 - dataflows 696, 1123
 - names 598
 - non-atomic 598
 - Rational Statemate 1388
 - StatemateBlock 1388
 - Flows 1264
 - activity 624, 663
 - block definition diagram 1266
 - data 674
 - embedded 586
 - in structure diagrams 835
 - in SysML 1226
 - initial 624, 1261
 - initial for activity diagram 635
 - initial in activity diagram 1261
 - limitations 586
 - Focus
 - thread in animation 1098
 - thread in tracer 1158
 - Folders 855
 - in browser 299
 - Fonts 395
 - Kanji characters 327
 - Footers 938
 - Force
 - complete code generation 913
 - roundtripping 1050
 - Fork 739
 - synchronization bar 638, 639
 - Fork node 1263
 - Form field 372
 - Formal modules 1397
 - Format option 436
 - Formats
 - copy from one element to another 440
 - default formatting 445
 - exported diagram images 364
 - reports 1195
 - toolbar 42, 439
 - Forward button 39
 - Frames
 - activity 626
 - Frames, swimlane 643
 - Framework
 - OMReactive class 743
 - support for 965
 - Free shapes 42, 422
 - Free text check box 346
 - Friend dependency 976, 1058
 - Friendship 743
 - FS file 1189
 - FullTypeDefinition property 111
 - Fully constructive diagrams 6
 - Functional C 2
 - FunctionalC profile 44, 207, 587
 - Functions 318
 - changing order in generated code 942
 - roundtripping 1055
 - tab 591
 - Functor class 652
- ## G
- Gateway 1394
 - Gauge control 797
 - Gauge tool 795
 - GEN macro
 - events with arguments 1096
 - standard operations 964
 - statechart 743
 - static memory allocation 961
 - tracer 1153
 - General tab 862
 - Features window 49, 66
 - port 93
 - Generalizations 527
 - Generate
 - class code 85
 - code for actors 926
 - code for actors in UCDs 526
 - code for component diagrams 928
 - code for links 570
 - code for objects 536
 - code incrementally 912
 - code with names 1252
 - complete code 913
 - event for statechart 743
 - event, tracer 1153

- formal reports 1195
- makefiles 914
- package 913
- report 1203
- Generate code
 - elements 919
- Generate option 577
- GeneratedCodeInBrowser 1058
- GeneratedCodeInBrowser property 925
- GenerateDirectory property 602, 607, 612, 1010
- GeneratePackageCode 913
- Generation
 - configuration 861
 - main configuration 861
- GeneratorRulesSet property 984
- Generic Class 122
- Global
 - functions 318
 - tags 391
 - variables 319
- Glossary 1467
- Go Event command 1154
- Go Idle command 1154
- Go Step command 1154
- Graphic editors 13, 62
 - collaboration diagram 13
 - component diagram 14
 - deployment diagram 14
 - drawing area 23
 - object model diagram 13
 - sequence diagram 13
 - statechart 13
 - use case diagram 13
 - using IntelliVisor 462
- Graphical annotations 366
- Graphics
 - block definition diagrams 1268
- Grid option 428
- Groups
 - instance 715
 - message 720, 722
- Guard 747
 - field 737
- Guidelines for naming model elements 251

H

- H_EXT macro 174
- Handles, selection 431
- Harmony 1234
 - Architectural Design Wizard 1238
 - auto-rename actions 1234
 - Copy MOEs from Base option 1235
 - Copy MOEs TO Children option 1235
 - generate N2 matrix 1235
 - Link Wizard 1240
 - measure of effectiveness (MOE) stereotype 1235

- measures of effectiveness (MOE) 1233
- measures of effectiveness (MOE) stereotype 1235
- profile 208, 240, 1232, 1245
- project type 1232
- special menu commands 1234
- test bench statechart 1235
- trade analysis 1235, 1237
- Harmony process 1230
- HasIDEInterface property 291
- Header file 932
- Header property 999
- Headers 938
- Heaps 959
- Help 1456
 - custom file 379
 - generate support request 1465
- Help applications
 - submenu structure 472
- help command 1155
- Helper applications 470
 - .hep files 476
 - adding links to VBA macros 479
 - deleting links to 470
 - examples of menu command links 474, 475
 - for MODAF 1342, 1372
 - icons on the Helpers window 470
 - linking to external programs 472
 - linking to Rational Rhapsody Tools menu 472
 - modifying 478
 - modifying links to 478
 - moving position of links on Rational Rhapsody Tools menus 470
 - samples 476
- Helper utilities 1318
- Helpers
 - arguments variable 197
 - command variable 197
 - initialDir variable 197
 - isMacro variable 197
 - isVisible variable 197
 - name setting 197
 - name variable 197
 - numberOfElements variable 197
- Helpers window 470
- HelpersFile property 478
- HideCellNames property 232
- HideEmptyRowsCols property 232
- Hierarchical
 - directory structure 259
 - relation type 412
 - requirements 367
- Hierarchical mode 258, 259
 - changing to flat 259
- Hierarchy
 - property inheritance 192
- Hierarchy of reactive classes 778
- History connector 727

- Home page
 - changing using the GUI 1174
 - changing using the webconfig.c file 1194
- Horizontal message 683
- HTML 1195, 1200
 - browser to examine exported models 1428
 - viewing reports in 1201
- Hyperlinks 55
 - changing tag value 59
 - creating in browser 57
 - creating in Description tab 56
 - deleting 59
 - editing 58
 - following 58
 - icons for targets 57
 - limitations 59
 - return to origin point 20
 - tag values 59
 - target files 56
- I**
- IBM
 - Passport Advantage 1461
 - technical support 1461
- ICN 1465
- IDE 279
 - locating element in 279
 - synchronize 279
 - Visual Studio 288
- IDE menu commands 290
- IDEConnectParameters property 291
- IDEInterfaceDLL property 291
- Identification
 - \$id keyword 165
- IDF 217
- IDFProfile 208
- Image View field 88
- Images
 - associate with element 336
 - formats 364
- ImpIncludes property 938, 1013
- Implement
 - base classes 613
 - relation 563
- Implement Base Classes option 614
- ImplementActivityDiagram property 652
- Implementation 78
 - code, editing 616
 - composite types property 111
 - file structure 938
 - phase 9
 - property 563
- Implementation files 1054
 - inline keyword 1055
- ImplementationEpilog property 949
- ImplementationExtension property 988
- ImplementationProlog property
 - #ifdef-#endif 949
- ImplementBaseClasses environment variable 616
- ImplementFlowchart property 666
- ImplementWithStaticArray property 563, 961
- Implicit
 - contracts 91
 - object 531
- Import as External 1009
- ImportDefineAsType property 999
- Imported macros
 - limitations 1043
- Importing 1425
 - Eclipse projects 281
 - from Rational Rose 1410, 1411, 1412
 - incrementally Rational Rose models 1414
 - language selection 1429
 - models 1429
 - Rational Rhapsody model into Teamcenter 1300
 - Rational Rose association classes 1424
 - requirements into Rational Rhapsody with Rational Rhapsody Gateway 1243
 - SA DoDAF 1293
 - Simulink components 1432
 - VBA macros 485
- ImportJavaAnnotation property 1220
- In property 111
- Include
 - other PRP files 191
- Include file
 - for rapid ports 98
- Include Path field 851
 - configuration 864
- Include statements
 - reverse engineering 1004
- IncludeScheme property 612
- Increment/decrement operators 1278
- Incremental code generation 912
- Information
 - conveyed by flow 582
 - element 585
- Inheritance 548
 - activity diagrams 621, 654
 - adding a level 778
 - block definition diagram 1266
 - from an external class 549
 - from external class 549
 - of properties 192
 - overriding for statechart 776
 - removing a level 780
 - rules for statechart 773
 - statechart 771
 - stereotypes 389
 - superclass 548
- Initial connector
 - statechart 750
- Initial flows

- activity diagram 635
- Initial Instances field 863
- Initial value search option 346
- Initial Values 70
- Initialization
 - block 969
 - field, for blocks 534, 832
 - tab 863
- Initialization code field 863
- InitializationBlockDeclaration 969
- Initialize
 - attribute 78
 - static attribute 934
 - static attributes 71
- Initializer box 78
- InitialLayoutForTables 234
- Inject event 1095
- Inlining 786
- InOut property 112
- input command 1155
- Inserting projects 245
- Installation
 - root directory 427
 - systems engineering 1223
 - verifying DoDAF 1334
 - verifying MODAF 1374
- Instance group 715
- Instance groups
 - adding instances to 719
 - creating 717
 - deleting 717
 - modifying 718
 - resetting 719
- Instance lines
 - creating 677, 1116
 - options 680
- Instances 674
 - adding to instance group 719
 - anonymous 954, 955
 - component 1125
 - component in deployment diagrams 875
 - creating 693
 - names of 1125
 - navigation expressions 1126
 - specifying the value of 537
 - specifying value 537
- Instantiation
 - class templates 66
- Instrumentation 170
 - field 864
 - header file 933
 - implementation file 940
 - mode 864, 1084
 - selective 866
- Instrumentation Mode 147
- Instrumentation Scope field 867
- Integrate
 - CodeTEST 865
- Integrity checks 883
- IntelliVisor 462
 - activating 462
 - dismissing 462
 - information, activity diagrams 466
 - information, collaboration diagrams 464
 - information, component diagrams 464
 - information, deployment diagrams 465
 - information, OMDs 463
 - information, sequence diagrams 465
 - information, statecharts 466
 - information, structure diagrams 468
 - information, UCDs 468
- Interaction occurrence 698
 - creating 698
 - menu 699
- Interaction Operators
 - creating 701
- Interactive mode
 - command-line interface 1440
- Interface 847
 - adding new 94
 - component 857
 - component, creating 857
 - provided 90
 - provided for rapid ports 98
 - required 90
 - required for rapid ports 98
- InterfaceGenerationSupport property 976
- Interfaces 973
 - automatic creation of 1236
 - C language 974
 - command-line 1439
 - in C 973
 - naming conventions 252
 - naming guidelines 251
 - realizing 973
 - service contract 275
 - virtual tables 973
- Internal
 - checks 881
 - code editor 395
 - event 743
 - hyperlink 55
 - text editor 392
- Internal block diagrams 1235, 1264, 1269
 - drawing tools 1270
- Internal code editor 395
 - auto indenting text 398
 - bookmarks 407
 - color-coding 395
 - keyboard shortcuts 399
 - printing 408
 - property 395
 - searching 406
 - split views 403

- using Undo/Redo 405
- view options 395
- window properties 395
- Interrupt handler
 - static memory 961
 - using 785
- Interrupt-driven framework 217
- Intertask communication
 - generating code without 912
- InvokeMake property 915
- IS_COMPLETED() macro 651
- IS_IN query 781
- IsCompletedForAllStates property
 - IS_COMPLETED macro 651
 - local termination code 733

J

- JAR files 1214
- Java
 - annotations 1215
 - generating JAR files 1214
 - reference model 1221
 - static blocks 1213
 - static import 1212
- Java 5 concepts 1215
- Java language 2, 1207
 - annotations 1215
 - call stack 1114
 - code generator symbols 1052
 - component file type 852
 - composite types 106
 - enums 120
 - inheriting from an external class 549
 - initialization blocks 969
 - interfaces 613
 - modeling constructs 115
 - no flow port support 597
 - packages 205
 - port code generation 105
 - projects in Eclipse 127, 280
 - reverse engineering 985
 - roundtripping 86
 - SA post processing plug-in 1297
 - Send Action code generation 757
 - specify include classpath 997
 - static models 113
 - template limitation 126
- JavaAnnotation property 1221
- JavaAnnotations 1215
 - adding 1216
 - code generation 1220
 - creating 1215
 - limitations 1221
 - reverse engineering 1220
 - sample code 1219
 - using 1217

- Javadoc comments 1207
- JavaDocProfile 1209
- Join synchronization bars 638, 639
- Join transitions
 - activity diagram 636
 - statecharts 747
- Joins 739
 - flow chart 665
 - MISRA rule 33 739
- JPEG 364
- Junction connector 727

K

- Keyboard
 - accelerator keys 1453, 1455, 1457
 - application accelerators 1455
 - code editor accelerators 1458
 - mnemonics 1454, 1460
 - modifiers 1454
 - shortcuts 68, 399, 400, 1459
 - shortcuts, standard Windows 1455
- Keywords
 - \$(Name) 967
 - \$Arguments 966
 - \$Attributes 966
 - \$Base 966
 - \$ClassClean 164
 - \$OMAllDependencyRule 167
 - \$OMBuildSet 167
 - \$OMCleanOBJS 167
 - \$OMConfigurationLinkSwitches 167
 - \$OMContextDependencies 167
 - \$OMContextMacros 168
 - \$OMCPPCompileCommandSet 168
 - \$OMFileDependencies 169
 - \$OMFileImpPath 169
 - \$OMFileObjPath 170
 - \$OMFileSpecPath 170
 - \$OMImpIncludeInElements 170
 - \$OMLibExt 170
 - \$OMLinkCommandSet 171
 - \$OMSourceFileList 172
 - \$OMSpecIncludeInElements 172
 - \$Relations 966
 - activity 1257
 - additional user-defined (reverse engineering) 1001
 - compiler specific 949
 - compiler-specific 938
 - custom 176
 - expanding properties 967
 - in standard operations 966
 - inline 1055
 - predefined 163
- Kind box 106
- Knob tool 795

L

- Labels 327
 - assigning 329
 - display options 297
 - in Asian languages 327
 - mode 330
 - removing 330
 - search option 346
 - show compartment 88
 - supported elements 327
 - transition 636
 - transition, statecharts 741
- Languages 127
 - Asian supported 327, 346
 - independent type 113
 - Japanese 327
 - more than one in projects 268
 - primary implementation for Architect for Software edition 240
 - selecting during import 1429
 - type 108, 109
 - WSDL 272
- Layout menu
 - Grid option 428
 - Replicate option 446
- Layouts
 - matrix views 228
 - table view 223
- Leaf state 726
- LED control 802
- LED tool 795
- Legacy code 994, 1030
 - reverse engineering 600
- Level Indicator control 799
- Level Indicator tool 795
- LIB_CMD macro 175
- LIB_EXT macro 175
- LIB_FLAGS macro 175
- LIB_NAME macro 175
- LIB_POSTFIX macro 175
- LIB_PREFIX macro 175
- Libraries 299, 1089
 - add 1023
 - for SysML models 1226
 - model 1267
 - units 1267
- Libraries box
 - in configuration Settings tab 864
 - in Features window 851
- Library
 - \$OMLibs keyword 170
 - \$OMModelLibs keyword 171
 - component dependency 171
 - extensions 170
 - linking additional 170
- License information 1464
- Licensing 128
 - SA Importer 1293
- Lifecycle 537
- Lifeline
 - animation 1135
- Limitations
 - activity diagrams 654
 - actors characteristics 927
 - animation 1110
 - Browse From Here browser 305
 - called behaviors 647
 - CM for project list files 250
 - code generation 976
 - code generation for activity diagrams 654
 - code generation for flow charts 666
 - comments 1040
 - controlled files 357
 - CORBA 115
 - customized diagrams with custom elements 493
 - decomposition 700
 - DiffMerge 250, 359
 - DoDAF 1333
 - Eclipse 127
 - Eclipse workflow integration 287
 - external elements 612
 - Favorites browser 312
 - flow diagrams 666
 - flows 586
 - hyperlink tags 59
 - imported macros 1043
 - importing diagrams (reverse engineering) 1034
 - JavaAnnotations 1221
 - joins and MISRA rule 33 739
 - locating elements on a diagram 325
 - no Java flow ports 597
 - no Linux forward and back 39
 - panel diagrams 828
 - project 249
 - properties for only active projects 250
 - Rational Rhapsody Gateway 1244
 - Rational Rose import views 1409
 - reverse engineering 985
 - roundtripping 1048
 - roundtripping for Send Action 757
 - roundtripping restricted mode 1060
 - search and replace 359
 - show transitions states on animated sequence diagrams 1120
 - Simulink 1437
 - static memory allocation 963
 - subactivities 1261
 - swimlanes 645
 - Teamcenter and Rational Rhapsody 1302
 - templates 126
 - Undo 216
 - Web-enabled devices 1173
- Line numbers in code 922

- Lines
 - anchor 370
 - changing shape 420
 - maintaining shape 435
 - selection handles 431
- Link
 - additional libraries 170
- Link modules 1401
 - Rational DOORS 1406
- Link Switches field 853
- Link with editor 144
- LINK_CMD macro 175
- LINK_FLAGS macro 175
- Linking data in Rational DOORS 1400
- LinkModuleName property 1401
- Links 565, 841, 842, 1271
 - creating in OMD 565
 - generating code for 570
 - in collaboration diagrams 842
 - messages 844
 - modify features 843
 - structure diagram 835
- Linux
 - no forward & back navigation 39
 - not supported on Eclipse 127
 - viewing reports 1201
- Lists, edit 68
- LmLicenseFile property 1392
- Load
 - backup 221
 - log 265
 - option settings 711
 - units 262
- load.log file 265
- Local
 - heaps 959
 - host 1086
 - tag 391
 - termination code 733
- Local termination rules 651
- Local termination semantics 650
- LocalizeRespectInformation property 1065
- LocalTerminationSemantics property
 - activity diagrams 631
 - flow charts 662
 - statecharts 732
- Locate element from code 923
- Locate On Diagram command 323
- Log tab 25, 1035
- LogCmd 1156
- Logical file type 853
- Logical files mode 1003
- Loop activity flows 664
- Loop transitions 635
- Lost constructs 1045

M

- MacroExpansion property 1043
- Macros 482, 1043
 - \$OMDEFExtension 169
 - \$OMDIIExtension 169
 - AR 174
 - ARFLAGS 174
 - CALL 692
 - CALL_INST 692
 - CALL_SER 692
 - CP 174
 - CPP_EXT 174
 - custom properties 176
 - DLL_CMD 174
 - DLL_FLAGS 174
 - GEN, standard operations 964
 - H_EXT 174
 - imported 1043
 - importing and exporting VBA 485
 - IS_COMPLETED() 651
 - LIB_CMD 175
 - LIB_EXT 175
 - LIB_FLAGS 175
 - LIB_NAME 175
 - LIB_POSTFIX 175
 - LIB_PREFIX 175
 - LINK_CMD 175
 - LINK_FLAGS 175
 - OBJ_EXT 175
 - OBJS 175
 - OM_RETURN 691
 - PDB_EXT 175
 - RM 175
 - RMDIR 175
 - saving 484
 - to examine models 1425
 - VBA 479
- main 172
- Main Diagram field
 - actor 524
 - block 832
- MaintainWindowContent property 23
- mainThread 1099
- MakeFileContent property 915
- Makefiles 913, 914
 - files to include in executable 167
 - library component dependency 171
 - linking additional libraries 170
 - object files 171
- Manage Web-enabled devices 1171
- Map
 - custom properties to macros 176
- Map to Package option 1010
- Mapping rules for Rational Rose 1419
- Marshallng 964
- MARTE

- profile 208
- Mathematical relationships 1272
- MATLAB 1436
- MATLAB MEX compiler 1436
- Matrices 222, 1329
- Matrix Display control 800
- Matrix Display tool 795
- Matrix views 222, 230
 - binding view and layout 234
 - cell element types 228
 - customize for MODAF 1360
 - export data 236
 - from and to values 228
 - layouts 228
 - manage data 236
 - toggle empty rows filter 232
- MaximumPendingEvents property 961
- Measures of effectiveness (MOE) 1233, 1235
- Mechanistic design 9
- Memory
 - allocation 962
 - allocation algorithm 962
 - fragmentation 959
- Memory allocation algorithm 962
- Memory pools 960
- Merge nodes
 - activity diagram 636
 - flow charts 665
- Merging existing packages 1033
- Message diagram 45
- Message diagrams 669, 1497
- Message groups 720
 - adding messages to 722
 - creating 722
 - deleting 724
 - modifying 724
 - removing messages from 722
- Message icon 681
- Message Type field 684
- Messages 845, 1121
 - arguments, displaying 682
 - arrival times 710
 - code output 920
 - code to pass 1277
 - collaboration diagram 844, 845
 - collaboration diagrams 837
 - copying 686
 - creating 681
 - cutting 686
 - data 783
 - exchanging, port 101
 - excluding from a comparison 713
 - formal parameters 783
 - found 675
 - horizontal 683
 - IDE 279
 - link 844
 - lost 675
 - menu 684
 - moving 686
 - names 682
 - numbering 837
 - pasting 686
 - reply 674
 - reverse link 844
 - selecting 685
 - sequence diagrams 674, 681
 - slanted 683
 - statechart 751
 - suppressing in animated sequence diagrams 1123
 - tab for associations 843
 - to self 683
 - to self, displaying 1130
 - tracer 1167
 - types 687
- Metaclasses 156, 159, 490
- Meter control 798
- Meter tool 795
- Methodology
 - development 7
- MicroC 207
- MicroC profile 21, 208, 1303
 - target monitoring 21
- Microsoft
 - Excel 1235, 1237
 - PowerPoint 1195, 1200
 - Word 331, 1195, 1200, 1205
- Ministry of Defence Architecture Framework 1341
- MISRA rule 33 739
- MISRA98
 - profile 208
- MISRA-C 1998 739, 909
- MKS Source Integrity 149
- Mnemonics 1454, 1460
- MODAF 2, 1341, 1342
 - Acquisition viewpoint 1346
 - All Views viewpoint 1345
 - architectural conformance 1374
 - artifacts 1355
 - checking your model 1374
 - creating a project 1355
 - CustomizableTableAndMatrixLayoutsPkg 1362
 - CustomizableTableAndMatrixViewsPkg 1366
 - CustomizedStereotypesPkg 1363
 - customizing table and matrix views 1360
 - Dependencies Linker 1372
 - Domain Specific Language 1355
 - Drawing toolbar 1374
 - general troubleshooting 1374
 - helper applications 1342, 1372
 - Java plug-ins 1374
 - ModafReport.tpl 1370
 - Network Enabled Capability (NEC) 1341
 - Operational viewpoint 1345

-
- products 1347
 - profile 208, 1342
 - quality of service requirements 1349, 1353
 - ReporterPLUS template 1368
 - Strategic viewpoint 1345
 - Systems viewpoint 1346
 - Technical viewpoint 1346
 - troubleshooting with Check Model 1374
 - troubleshooting, Dependencies Linker 1373
 - troubleshooting, general 1374
 - troubleshooting, ReporterPLUS 1371
 - UML 1342
 - verifying installation 1374
 - viewpoints 1343
 - views 1347
 - ModafReport.tpl 1370
 - Mode
 - activity diagram 651, 732
 - analysis 672
 - design 672
 - drawing 417
 - Flat 258
 - flat 258
 - Hierarchical 259
 - hierarchical 258
 - instrumentation 864
 - operation 410
 - pinned 50, 52
 - rapid 97
 - repetitive drawing 418
 - stamp 440
 - statechart 650, 725, 732
 - Workbar 20
 - Model
 - \$FullModelElementName keyword 165
 - Model as language types option 1020
 - Model browser 150
 - Model Update Failure option 1031
 - Model Updating tab 1033
 - ModelCodeAssociativityFineTune property 1049
 - ModelCodeAssociativityMode property 1049
 - Model-driven Development (MDD) 1041, 1377
 - Modeled annotations 366
 - Modeled operation 652
 - Modeling class type 1023
 - Modeling policy 1009
 - Modeling toolbar 40
 - Models 205
 - actor element 577
 - adding new elements 134
 - adding units to 136
 - animation 1113
 - animation (partial) 1089
 - checking 140, 884, 885
 - classes 1023
 - connecting to from the Web 1176
 - connecting to from the Web, troubleshooting 1177
 - creating actions 1277
 - data analysis 222
 - define environment 1252
 - deleting elements 451
 - deleting items 306
 - designing 7
 - elements 490
 - elements aggregation 559
 - elements associations 552
 - elements classes 545
 - elements inheritance 548
 - elements, composite classes 546
 - examining 1425
 - examining in HTML browser 1428
 - execution 1277
 - exporting to XMI 1426
 - exposing to the Web 1174
 - find elements 345
 - importing XMI 1429
 - in a Web browser 1291
 - library for SysML 1226
 - locating type of items in 213
 - management views 5
 - minimum requirement 7
 - naming guidelines 251
 - one-way association 558
 - packages 547
 - print to screen 1278
 - properties 220
 - search and replace 137, 209
 - search in 35
 - searching 342
 - searching for text 344
 - simplified 979
 - specifying with Rational Rhapsody 7
 - SysML stereotypes for elements 1226
 - validation 1080
 - warnings 881
 - Web-enable 1288
 - XMI 1425
 - Modes 1499
 - animation 972, 1112
 - batch 1047
 - categories 1474
 - DMCA 1049
 - flat 1490
 - logical files 1003
 - recursive analysis 1004
 - silent 1094, 1112
 - watch 1094, 1112
 - Modifiers 1454
 - attributes 70
 - Constant 70
 - declaration 164
 - destructors 80
 - primitive operations 73
 - Reference 70
-

- Static 70
 - Modifying
 - data types 70
 - instance groups 718
 - message groups 724
 - reference class 1019
 - ModuleNameFromProject property 1402
 - Monitor Web-enabled devices 1171
 - Monitoring
 - target 21
 - Mouse, select and editing with 405
 - Moving
 - control point 434
 - element in graphic editor 435
 - elements between projects 248
 - elements in browser 331
 - messages 686
 - synchronization bar 640
 - MSVC60 dialect 1002
 - Multiple projects 244
 - Multiple selection 22
 - Multiplicity
 - array index 756
 - attributes 70
 - ports 80, 93
 - Multiplicity field
 - association 556
 - block 534, 832
 - Multi-threaded architecture 1
 - Mutator
 - implementation file 939
 - MutatorGenerate property 111
 - Mutators 67, 69
 - Mutex 961
- ## N
- Name
 - \$ComponentName keyword 164
 - \$ConfigurationName keyword 164
 - \$FullCodeGenerationFileName keyword 165
 - \$opname keyword 172
 - \$projectname keyword 172
 - \$TypeName keyword 174
 - FullModelElementName keyword 165
 - target 172
 - Name search option 346
 - Name/value pairs
 - for Web GUI 1183
 - Names 251
 - boxes and arrows 421
 - class, default 65
 - guidelines for model elements 251
 - instance 1125
 - message 682
 - project 219
 - roles for classifiers 679
 - special characters in 251
 - thread in animation 1099
 - thread in tracing 1147
 - view field 1181
 - Namespace 326
 - Naming conventions
 - for Rational Rhapsody 251
 - for Rational Rose 1411
 - Navigable field 557
 - Navigate
 - from Rational DOORS to Rational Rhapsody 1396
 - hyperlinks 58
 - to reference SD 699
 - Navigation
 - customizing 1182
 - personalized 1194
 - Nested classes 67
 - Nested packages 67
 - exporting to Rational DOORS 1397
 - Nested types 67
 - NetCentric profile 208, 272
 - Architect for Software edition 240
 - creating projects using 275
 - generating WSDL file 275
 - generating WSDL output 276
 - service provider 272
 - WSDL output 276
 - Network Enabled Capability (NEC) 1341
 - New Attribute option 577
 - New element types 489
 - New Operation option 577
 - New Reception window 74
 - New Statechart option 577
 - New Term stereotype 385, 505
 - NO_OUTPUT_WINDOW 1036
 - NO_OUTPUT_WINDOW variable 197
 - Nodes 321, 873
 - call operation 622
 - decision 624
 - deployment diagrams 873
 - features 874
 - fork 624
 - join 624
 - merge 624
 - merge for activity diagram 636
 - merge for flow charts 665
 - owner 874
 - Notes 365
 - converting to comments 370
 - Rational Rose 374
 - NotifyOnInvalidatedModel property 1059
 - Null transitions 747
 - activity diagrams 650
 - statecharts 747
 - Null trigger 747

O

- OBJ_EXT macro 175
- Object
 - files 171
 - files to include in executable 167
- Object analysis 8
- Object Management Group 3
- Object Management Group (OMG) 1425
- Object model diagrams 4, 5
 - adding operations and attributes 530
 - automatically populating 412, 415
 - compared to structure diagram 829
 - creating 44, 410
 - drawing icons 530
 - editor 13
 - files 264
 - flowports in 597
 - icon in browser 301
 - importing by reverse engineering 987, 1034
 - IntelliVisor information 463
 - perform trade analysis in Harmony 1237
- Object nodes 632
 - associated with class 633
- ObjectIsSavedUnit property 258
- Objects 1, 318, 531
 - changing order of 833
 - changing the order of 537
 - concurrency 534, 832
 - converting types 535
 - creating in browser 318
 - creating in OMD 532
 - dependencies between 572
 - destroying 693
 - features 534
 - generating code for 536
 - in activity diagrams 624
 - in structure diagram or OMD 831
 - linking to ports 103
 - modifying 832
 - multiple 840
 - relations, show all 82
 - show status 1159
 - specifying the value of 537
 - StatemateBlock 1386
 - structure diagrams 831
 - types of 531
- OBJS macro 175
- OM_RETURN macro 691
- OMDOCROOT variable 196
- OMG 3
 - SysML profile 208
 - testing profile 209
- OMMemoryPoolsIsEmpty() operation 962
- OMReactive class 743
- OMROOT
 - setting 196
- OMSimulinkBlock 1433
- On/Off Switch control 803
- On/Off tool 795
- Only from file list option 1004
- Only header file with the same name option 1003
- Open
 - Active Code View 33
 - animated sequence diagram 1116
 - browser window 296
 - editor 86
 - existing diagrams 411
 - main diagram 87
 - multiple Features windows 51
 - multiple projects 244, 245
 - OpenDiagramsWithLastPlacement property 263
 - OpenWindowsWhenLoadingProject property 263
 - parent statechart 765
 - project 209
 - project with workspace information 263
 - subactivity diagram 630
 - submachine 765
 - workspace 263
- Open Main Diagram option
 - actor 577
- Open Reference Sequence Diagram option 680, 699
- OpenDiagramsWithLastPlacement property 263
- OpenWindowsWhenLoadingProject property 263
- Operation bodies search option 346
- Operation mode 410
- Operational view (DoDAF) 1312
- Operational viewpoint (MODAF) 1345
- Operations 688
 - adding to events 744
 - adding to OMD 530
 - adding to use case 522
 - call 623, 1259
 - calls 867
 - calls, during animation 692, 1108
 - changing order in generated code 942
 - code generation in activity diagram 652
 - contracts allocation 1238
 - display options 89
 - editing in multiple elements 333
 - generated 925
 - implementation file 939
 - modeled 652
 - naming conventions 252
 - primitive 72
 - receptions 73
 - roundtripping 1054
 - roundtripping triggered 1055
 - standard keywords in 966
 - statechart 751
 - that cannot be undone 216
 - triggered 75, 688, 745
- Operations tab 71
- Operator, overloading 949

- Or state 726
 - code generation 732
 - local termination code, flat 733
 - Order
 - of types, changing 115
 - of variables 319
 - Ordered to-many relations 957
 - Organize tree 296
 - Orthogonal
 - relation type 413
 - OSEK Adaptor 1380
 - hardware and software 1381
 - Out property 112
 - Output 209
 - command 1157
 - consistency checking results 1235
 - reports 153
 - search results 137
 - Output window 24, 130
 - Animation tab 32
 - Build tab 29
 - Check Model tab 26
 - Configuration Management tab 32
 - display search results 343
 - displayed in Eclipse 130
 - Log tab 25
 - Search Results tab 32
 - Output Window option 1036
 - OV-1 High Level Operational Graphic 1325
 - OV-2 Operational Node Connectivity 1325
 - OV-4 Organizational Relationships Diagram 1325
 - OV-5 Operational Activity Diagram 1325
 - OV-6a Operational Rules Model 1325
 - OV-6c Operational Event-Trace Description
 - diagram 1325
 - OV-6c Operational State Transition Description
 - diagram 1325
 - OV-7 Logical Data Model 1325
 - Overridden properties view 302
 - Override properties 183
 - Overwriting existing packages (during reverse engineering) 1033
 - Owners
 - actor 525
 - node 874
 - OXF 909
 - in a C project 217
 - libraries 1090
- P**
- PackageIsSavedUnit property 258
 - Packages 316, 526, 547
 - adding 134
 - as containers 326
 - assigning to a deployment diagram 880
 - converting to profile 378
 - creating hierarchy in C 607
 - creating in browser 317
 - creating in OMD 547
 - cross-package initialization 930
 - default systems engineering 1224
 - dependencies 572
 - drag and drop 331
 - editing multiple 333
 - elements 205
 - elements types 316
 - files 264
 - guidelines for creating 316
 - in requirements diagrams 1247
 - in UCD 526
 - naming guidelines 251
 - nested 67
 - Rational DOORS 1397
 - relations, show all 82
 - roundtripping supported modifications 1055
 - setup DoDAF 1320
 - stereotypes 385
 - storing in directories 258, 259
 - SysML profile 1226
 - units 253
 - views 1227
 - WSDL stereotyped 276
 - Pan 41, 455
 - Panel diagrams 791
 - animation 791, 794
 - attribute types 810
 - binding 808
 - binding (mapping) table 810
 - Bubble Knob control 796
 - Button Array control 805
 - Button Array tool 795
 - caption for Push Button control 811
 - changing control flow 811
 - changing display name options 827
 - changing graphical properties of a control
 - element 812
 - changing settings 811
 - color schemes for Matrix Display and Digital Display
 - controls 811
 - Control Properties window 809
 - creating 44, 410, 794, 795
 - Digital Display control 801
 - Digital Display tool 795
 - drawing icons 795
 - features 792
 - Gauge control 797
 - Gauge tool 795
 - graphical settings for Button Array control 827
 - Knob tool 795
 - LED control 802
 - LED tool 795
 - Level Indicator control 799
 - Level Indicator tool 795

- limitations 828
- Matrix Display control 800
- Matrix Display tool 795
- Meter control 798
- Meter tool 795
- minimum and maximum values for controls 811
- On/Off Switch control 803
- On/Off switch shape styles 811
- On/Off tool 795
- properties for Bubble Knob control 812
- properties for Digital Display control 822
- properties for Gauge control 814
- properties for LED control 823
- properties for Level Indicator control 820
- properties for Matrix Display control 822
- properties for Meter control 817
- properties for On/Off Switch control 824
- properties for Slider control 825
- Push Button control 804
- Push Button tool 795
- Select tool 795
- Slider control 807
- Slider tool 795
- Text Box control 806
- Text Box tool 795
- Parallel development 149
- Parameters
 - formal message 783
 - Webify 1174
- Parametric diagrams 1272
 - adding equations 1276
 - allocation 1273
 - binding connector 1273
 - block 1273
 - constraint binding 1276
 - constraint block 1273
 - constraint blocks for 1274
 - constraint parameters 1273
 - constraint properties 1274, 1275
 - constraint property 1273
 - create package 1273
 - dependency 1273
 - drawing tools 1273
 - problem satisfaction 1273
 - showing constraints 1275
- params
 - pseudo-variable 749
- params-> pseudo-variable
 - message parameters 783
- Parent statechart 765
- ParserErrors property 1059
- Parsing Errors option 1031
- Part 457, 531, 546, 1506
 - and composite class 546, 559, 561
 - decomposition 700
 - relation to whole 534, 832
- Partial animation 1089
 - per configuration 1090
- Partially constructive diagram 6
- Partition line 698
- Parts
 - browser icon 299
 - systems engineering 1264
- Passport Advantage 1461
- Path field 590, 856
 - file 852
 - folder 856
- Paths 326
 - names in commands 1441
 - relative 260
 - specifying physical 378
- Pause Animation command 1095
- pc_server.dxl 1392
- PDB_EXT macro 175
- Personalized bottom navigation 1194
- Personalized Navigation page 1181
- Perspectives 129, 141
 - Debug 131, 146
 - Modeling 130
- PI variable 319
- Platform Integration (Rational Rhapsody and Eclipse) 127
- Plug-ins
 - MTT variable 199
 - Rational Rhapsody 507
- Points (adding to arrow) 420
- Pooling events 960
- Populate Diagram 1245
- Populate Diagram feature 412, 415
- Ports 92
 - API for C++ 101
 - atomic flow 597
 - automatic creation of 1236
 - behavior attribute 94
 - C language 974
 - C language optimization 975
 - code generation in Java 105
 - code generation(C) 104
 - communicating with ports with multiplicity 102
 - connecting 97
 - connecting objects via ports 102
 - creating 92
 - creating programmatically 103
 - definition 90
 - display in the browser 97
 - display options 95
 - exchanging messages 101
 - features 93
 - field 89
 - flow 597
 - flow for StateBlock 1388
 - IDE 279
 - implicit contracts 91
 - in structure diagrams 835

- in SysML 1226
- linking objects via ports with multiplicity 103
- linking to owning instance 103
- links 1271
- listing 213, 1201
- multiplicity 93
- non-atomic flow 598
- properties 96
- rapid 91, 97, 974
- reversed attribute 94
- SDLBlock behavior 293
- service 973
- show all 1269
- show in views 233
- specifying contract 94
- standard 1266, 1271
- tab 80
- Ports tab
 - behaviors 80
 - contracts 80
 - multiplicity 80
 - reversed 80
- PowerPoint 1195, 1200
- PreCommentSensibility property 1039
- Predefined checks 881
- Predefined type 113
- PredefineIncludes property 1059
- PredefineMacros property 1059
- Preprocessing
 - reverse engineering 995
 - symbol 998
- Primary model elements 326
- Primary templates 123
- Primitive operation 688
- Primitive operations 72
- Print
 - diagrams 360
 - from internal code editor 408
- Print to screen 1278
- Priority
 - thread 1100
 - transition 768
- Private
 - attributes 67
 - event 743
- Problem (diagram element for SysML profile) 1226
- Problems
 - found by Check Data 1404, 1405
 - Simulink 1434
- Process tab 1031
- Product lines 241
- Profiles 2, 206, 207, 275, 375, 505
 - AdaCodeGeneration 207
 - adding to existing project 215
 - Architect for Software edition 240
 - as packages 205
 - as settings 314
 - AutomotiveC 207, 1377, 1380
 - AUTOSAR 207, 1377
 - backward compatibility 377
 - browser icon 300
 - CodeCentricCPP 207
 - compatibility settings 314
 - converting to package 378
 - creating your own 486
 - default 207
 - diagrams available for 43
 - DoDAF 207, 1311
 - enabling access to custom help file 379
 - FixedPoint 207
 - FunctionalC 14, 44, 207, 240, 587
 - Harmony 208, 240, 1232, 1238, 1245
 - IDF 208
 - JavaDocProfile 1209
 - MARTE 208
 - MicroC 21, 208, 1303
 - MISRA98 208
 - MODAF 208, 1342
 - NetCentric 208, 240, 272
 - new term 385
 - properties 378
 - Rational Rhapsody predefined 207
 - RespectProfile 208
 - re-using customized 487
 - RoseSkin 208
 - Schedulability, Performance, and Time (SPT) 208, 277
 - SDL 208
 - Simulink 208, 1433
 - SPARK 208
 - SPT 208, 277
 - StatemateBlock 208, 1386
 - stereotypes 385
 - SysML 208, 240, 1224, 1225, 1238, 1245, 1267
 - Testing 1235
 - TestingProfile 209
 - UPDM 209
 - when not needed 376
- Programs
 - command-line Rational Rhapsody 1439
 - external 472
 - files from Word or Excel 300
 - ReporterPLUS 1195
- Project
 - VBA file 481
- Project Overview diagrams 1325
- Projects 133, 205, 244
 - \$projectname keyword 172
 - active 11, 244, 245, 246
 - adding elements 215
 - adding new elements 134
 - adding profile to existing 215
 - adding project to list 249
 - Architect for System Engineers edition 239

- archiving 221
- AUTOSAR 1378
- autosave 218
- backing up 220
- build Eclipse 145
- closing 220
- closing all 248
- copying elements 247
- copying in multiple 246
- creating 206
- creating NetCentric 275
- creating new Rational Rhapsody 206
- creating units 255
- debug Eclipse 146
- directory structure 258
- disassociating Eclipse from Rational Rhapsody 287
- DoDAF 1322
- Eclipse 280
- Eclipse IDE 141
- editing Rational Rhapsody 215
- elements 205
- files and directories 264
- flat directory structure 258
- folder 244
- Harmony 1232
- incremental save 218
- insert another in open project 244, 245
- insert existing 244
- insert new 244
- inserting into other project 245
- keyboard shortcuts 1455
- large 327
- lifecycle instance values 537
- limitations 249
- loading backup 221
- managing lists 248
- migration 268
- MODAF 1355
- multi-language 268
- multiple 244
- NetCentric 272
- new 249
- opening 209
- opening project list 249
- organizing 295
- Rational Rhapsody profiles 207
- referencing in multiple 246
- removing from project list 249
- renaming 219
- restoring 221
- saving 217
- saving project in project list file 248
- settings 206
- SOA 272
- systems engineering 1223
- tool icons 35
- tools 33
- types 206, 275
- unit 253
- validation 1286
- Visual Studio 288
- without profiles 376
- workspace 250
- Prolog
 - header file 933
 - implementation file 938
- Properties 155, 220, 1398, 1401
 - AcceptChanges 1054, 1057, 1061, 1212, 1214
 - active project's displayed 250
 - ActiveThreadName 1100
 - ActiveThreadPriority 1100
 - activity diagram settings 1258
 - ActivityReferenceToAttributes 653
 - adding comments to files 189
 - adding customized 189
 - AdditionalBaseClasses 965
 - AdditionalKeywords 1001
 - AdditionalNumberOfInstances 960
 - AddNewMenuStructure 501, 503, 504, 505
 - AddressSpaceName 754
 - affecting diagram editors 417
 - Aggregates 1376
 - AlternativeDrawingTool 495
 - AnalyzeGlobalFunctions 1017
 - AnalyzeGlobalTypes 1017
 - AnalyzeGlobalVariables 1017
 - AnalyzeIncludeFiles 1004
 - AnimateSDLBlockBehavior 293
 - animation 1130
 - AutoCopied 378
 - AutoReferences 379
 - AutoSaveInterval 218
 - AvailableMetaClasses 490
 - backup 220
 - BackUps 220
 - BaseNumberOfInstances 960
 - BlockIsSavedUnit 258
 - blocks 1264
 - BrowserIcon 493
 - change directory scheme 258
 - changing 177
 - changing the common view 182
 - changing values 185
 - ClassCentricMode 672
 - ClassCodeEditor 86, 924
 - ClassIsSavedUnit 258
 - CleanupRealized 672
 - CodeGeneratorTool 979, 1039
 - CollectMode 1042
 - Common view 180
 - CommonList 505
 - ComplexityForInlining 787
 - ComponentIsSavedUnit 258
 - ConstantVariableAsDefine 999

- constraint in parametric diagrams 1274
- ContainerSet 563
- controls 182
- CPPCompileCommand 917
- CreateDependencies 1004
- CreateReferenceClasses 1017
- custom 176
- CustomHelpMapFile 381, 382
- CustomHelpURL 380, 382
- DataTypes 1023
- DefaultDirectoryScheme 258
- DefaultProvidedInterfaceName 98
- DefaultReactivePortBase 98
- DefaultReactivePortIncludeFile 98
- definitions 83, 96, 156, 178
- DescriptionTemplate 945, 1209, 1211
- DiagramIsSavedUnit 254, 258
- DiagramsToolbar 494, 496
- displaying 49, 177
- DisplayMessagesToSelf 1130
- DisplayMode 296
- DrawingShape 495
- DrawingToolbar 493, 496
- DrawingToolIcon 493, 494
- DrawingToolTip 494
- Eclipse Workbench 285
- EditorCommandLine 855, 925
- EventGenerationPattern 757
- EventToPortGenerationPattern 757
- ExcludeFilesMatching 992
- ExecutionModel 1303
- expanding with keywords 967
- FileIsSavedUnit 258
- FileName 549
- Files (for reverse engineering) 991
- filtered views 180
- flowKeyword 580
- for black-box testing 1134
- for composite types 111
- FullTypeDefinition 111
- GeneratedCodeInBrowser 925, 1058
- GenerateDirectory 602, 607, 612, 1010
- GeneratePackageCode 913
- GeneratorRulesSet 984
- grouping of 156
- Header (for reverse engineering) 999
- HelpersFile 478
- HideCellNames 232
- HideEmptyRowsCols 232
- ImpIncludes 1013
- ImplementActivityDiagram 652
- Implementation 563
- Implementation of composite types 111
- ImplementationExtension 988
- ImplementFlowchart 666
- ImplementWithStaticArray 563, 961
- import from Rational Rose model 1413
- ImportDefineAsType 999
- ImportJavaAnnotation 1220
- In 111
- IncludeScheme 612
- inheritance 192
- InitializationBlockDeclaration 969
- InitialLayoutForTables 234
- InOut 112
- InterfaceGenerationSupport 976
- internal code editor 395
- InvokeMake 915
- IsCompletedForAllStates, IS_COMPLETED
 macro 651
- IsCompletedForAllStates, local termination code 733
- JavaAnnotation 1221
- LocalizeRespectInformation 1065
- Locally Overridden view 180
- LocalTerminationSemantics 631, 662
- LocalTerminationSemantics, statecharts 732
- MacroExpansion 1043
- MakeFileContent 915
- MaximumPendingEvents 961
- ModelCodeAssociativityFineTune 1049
- ModelCodeAssociativityMode 1049
- ModuleNameFromProject 1402
- MutatorGenerate 111
- NotifyOnInvalidatedModel 1059
- ObjectIsSavedUnit 258
- OpenDiagramsWithLastPlacement 263
- OpenWindowsWhenLoadingProject 263
- Out 112
- Overridden view 180
- overridden view 302
- overriding 183
- PackageIsSavedUnit 258
- ParserErrors 1059
- ports 96
- PreCommentSensibility 1039
- PredefineIncludes 1059
- PredefineMacros 1059
- profile 378
- PropertiesXMLPath 1413
- ProtectStaticMemoryPool 960
- Rational Rose model 1413
- RealizeMessages 672
- ReferenceImplementationPattern 111
- ReflectDataMembers 1029
- RemoteHost 1086
- ReportChanges 1059
- RespectCodeLayout 1039, 1043, 1065
- RestrictedMode 1060
- ReturnType 112
- reverse engineering 1022
- RootDirectory 1014
- roundtripping 1059
- RoundtripScheme 1039, 1043, 1060
- RTFCharacterSet 1205

SDLSignalPrefix 292
 searching for 178
 serialization 970
 set for Asian languages 327
 setting a Web-enabled device 1175
 ShowAnimCancelTimeoutArrow 1123
 ShowAnimCreateArrow 1123
 ShowAnimDataFlowArrow 1123
 ShowAnimDestroyArrow 1123
 ShowAnimStateMark 1120
 ShowAnimTimeoutArrow 1123
 ShowArguments 682
 ShowAttributes 414
 ShowCGSimplifiedModelPackage 979
 ShowContainerElementForPorts 233
 ShowLabels 327
 ShowLogViewAfterBuild 31
 ShowOperations 414
 ShowPorts 96
 ShowPortsInterfaces 96
 simplification 979
 simplifying C code generation 979
 SpecificationEpilog 949
 SpecificationExtension 988
 SpecificationHeader 945
 SpecificationProlog 949
 SpecInclude 576
 SpecIncludes 1013
 StandardOperations 966
 StrictExternalElementsGeneration 604
 subjects 156, 157
 Submenu#List 505
 Submenu#Name 505
 Submenu1List 502
 Submenu1Name 502
 Submenu2List 502
 Submenu2Name 502
 Submenu3List 502
 Submenu3Name 502
 Submenu4List 502
 Submenu4Name 502
 SubmenuList 502, 505
 SupportExternalElementsInScope 604
 tab 96, 177, 866
 table display 179
 to customize Rational Rhapsody 155
 to set sequence numbers 689
 TriggerArgument 112
 UsageType 611
 UseAsExternal 549, 1016
 UseCalculatedRootDirectory 1015
 UseDescriptionTemplates 1209
 UseIncrementalSave 218
 UseRapidPorts 292
 UseRemoteHost 1086
 VariableInitializationFile 319
 VariableLengthArgumentList 935

viewing overridden 302
 WebComponents 1175
 WebManaged 1175
 PropertiesXMLPath property 1413
 Protected attributes 67
 ProtectStaticMemoryPool property 960
 Provided interface 90
 for rapid ports 98
 PRP file
 including other 191
 PurgeOnDelete property 1398
 Push Button control 804
 Push Button tool 795
 Pushpin note 372

Q

Qualified association
 in OMDs 557
 Qualified to-many relations 957
 Qualifier field 557
 Queries 222
 Queue 743
 Quick Add 390
 quit command 1157
 ending the tracer session 1169

R

Random access to-many relations 958
 Rapid external modeling 605
 Rapid ports 974
 include files 98
 reactive base class 98
 required interface 98
 Rational ClearCase 3, 132, 149
 Rational DOORS 1242, 1243, 1391
 action pins/activity parameters 650
 calling 1396
 Check Data 1404
 creating a project for Rational Rhapsody 1396
 dxlapi.dll 1392
 exiting 1407
 export options 1397
 formal modules 1397
 installation requirements 1392
 Interface window 1396
 link modules 1392, 1406
 linking data 1400
 navigating to Rational Rhapsody 1396
 nested packages 1397
 objects 834
 on Solaris systems 1392
 pc_server.dxl 1392
 requirements 1391
 shadows 1400
 stored information 1402

- using with Rational Rhapsody 1393
- Rational Rhapsody 1, 127
 - \$RhapsodyVersion keyword 173
 - .ini file 196
 - accessing Web services 1185
 - adding to Web design 1189
 - affinity variable 197
 - analysis phase 7
 - and Rational DOORS 1393, 1396
 - and Rational Rose 1410
 - and Tornado 289
 - animation 131
 - Animation toolbar 40, 1093
 - API, using with VBA 481
 - Architect for System Engineers edition 2
 - backing up 220
 - Browse From Here browser 304
 - browser 295
 - building the target 916
 - Check Model tool 882, 885, 1375
 - code generation, incremental 912
 - command-line interface 1439
 - common drawing tools 41
 - components-based development 973
 - convert configuration to Eclipse 281
 - creating model from existing Teamcenter project 1301
 - customized workspace 262
 - design phase 8
 - development methodology 7
 - diagram icons 43
 - diagrams 6
 - DiffMerge tool 266, 1302
 - directory structures 258, 259
 - drawing tools 17
 - dynamic model-code associativity 914
 - Eclipse Debug perspective 130
 - Eclipse Modeling perspective 129
 - editions 2, 237, 275
 - environment settings 197
 - environment variables 196
 - exporting 1425
 - exporting code 135
 - exposing elements on the Web 1174
 - Favorites browser 11, 306
 - Favorites toolbar 40, 307
 - features 2
 - file extensions 134
 - filtering out file types 134
 - formal reports 1195
 - generating code 141
 - generating reports 1203
 - glossary 1467
 - graphic editors 13, 62
 - help 1456
 - implementation phase 9
 - importing 1425
 - integrating with CodeTEST 865
 - IntelliVisor 462
 - keyboard shortcuts 1459
 - license for Platform Integration 128
 - license information 1464
 - linked elements 1400
 - list of Check Model tool checks 889
 - locating code in Eclipse 286
 - MODAF 1341, 1342, 1343
 - model checking 885
 - model, connecting to from the Web 1176
 - modeling class types 1023
 - Modeling toolbar 40
 - modifying elements shared with Teamcenter 1302
 - multiple projects 244
 - perspectives in Eclipse 129
 - Platform Integration (Rational Rhapsody and Eclipse) 127
 - plug-in for Eclipse 280
 - predefined keywords 163
 - project 205
 - project files and directories 264
 - project types 206, 275
 - properties for Rational DOORS 1398, 1401, 1402
 - properties for Tornado 291
 - Rational DOORS project 1396
 - Rational StateMate block in 1385
 - reports 1195
 - repository 128
 - roundtripping 1048
 - running animated application without Rational Rhapsody 1141
 - running the executable 918
 - search facility 1280
 - standard toolbar 35
 - stored information about Rational DOORS 1404
 - synchronizing with Rational StateMate 1388
 - systems engineering version 1223
 - Teamcenter 1298
 - testing phase 9
 - timeouts 746
 - units 136, 149
 - utilities 15
 - VBA toolbar 40
 - Web server 1193
 - windows 17, 39
 - with IDEs 279
 - Workflow Integration 127
 - XMI in development 1425
- Rational Rhapsody Gateway 1242, 1243
 - importing requirements into Rational Rhapsody 1243
 - limitations 1244
 - requirements and analysis 1243
 - use case diagrams 1242
- Rational Rhapsody handle 1391
- Rational Rhapsody Platform Integration (Rational Rhapsody and Eclipse)

- confirming 128
- Rational Rose 1409
 - import properties 1413
 - importing 208
 - importing association classes 1424
 - importing code from imported Rational Rose model 1417
 - importing from 1410
 - incrementally importing 1414
 - look-and-feel 1411
 - mapping rules 1419
 - merging imported code to imported Rational Rose model 1418
 - naming conventions 1411
 - notes in 374
 - properties 1413
 - re-importing a package 1415
 - rose_properties_import.xml 1413
 - rose_properties_import_java.xml 1413
 - selecting elements to import 1410
 - XML map file 1413
- Rational Rose Logical View 1409
- Rational Statemate 1385
 - block 1386
 - Block profile 208
 - flowports 1388
 - model requirements 1385
 - preparing for Rational Rhapsody 1385
 - synchronizing with Rational Rhapsody 1388
 - troubleshooting 1389
- Rational Synergy 3, 132
- Rationale 1226
 - block definition diagram 1267
 - requirements diagram 1247
- Reactive class 773
 - refining the hierarchy 778
- Reactivity 1
- Real time environment 1
- Realization field 684
 - instance line 678
- Realization Of option 844
- Realization relationship 550, 973
- Realization, implementing base classes 614
- Realize elements of base class 617
- RealizeMessages property 672
- Real-Time Workshop 1431, 1432, 1433
- Rearrange
 - elements 448
 - elements in file 854
- Receptions 73
 - browser icon 320
 - Features window 75
- Rectilinear line 420
- Recursive analysis mode 1004
- Recursive composite 326
- Redo 216
- Refactor 219
- ReferenceImplementationPattern property 111
- References 36, 211, 246
 - class 1018
 - diagram 698
 - finding 342
 - finding element 342
 - limitations 647
 - modifier 70
 - to constraints 370
 - to element 342
 - to locate elements 330
 - unit using environment variables 260
 - unresolved 261, 278
- References window 250, 342
- Referencing 246
- refine stereotype 1250
- Refinement 1226
- Reflect Data Members option 1021, 1028
- ReflectDataMembers property 1029
- Refresh 61, 226, 456
 - new terms 496
 - table and matrix view 222
 - view option 456
 - Web pages, using the GUI 1174
 - Web pages, using webconfig.c file 1194
- Regenerate
 - code 33
 - configuration files 913
- RegisterUpload function 1194
- RegisterUpload() call 1186
- Re-importing a Rational Rose package 1415
- Relations 80, 313, 956
 - accessor 939
 - adding 313
 - bounded 961
 - fixed 961
 - implementing 563
 - mutator 939
 - of instances 571
 - ordered to-many 957
 - qualified to-many 957
 - random access to-many 958
 - roundtripping 1055
 - show all 82
 - show in views 233
 - Show Relations in New Diagram menu command 82
 - tab 80
 - to external element 611
 - to whole field 534, 832
 - to-many 956
 - to-one 956
 - type for populating diagrams 412
- Relationships 527
- Remarks 368
 - anchors 370
 - converting to comment 370
 - creating 366

- deleting 373
- dependencies 367
- display options 372
- editing 368
- types of 365
- Remote
 - managed devices 1171
 - target animation 1087
- RemoteHost property 1086
- Remove
 - properties from the common view 182
- Remove from common list option 182
- Removing
 - classes 90
 - element from model 451
 - element from view 451
 - hyperlinks 59
 - label 330
 - level of inheritance 780
 - tags 393
 - user-defined points from lines 420
- Rename 1234
 - element 332
 - project 219
- Renaming 219
- Re-order model element in browser 334
- Repetitive Drawing Mode 418
- Replace 137, 209, 212
- Replicate option 446
- Report on imported items 990
- ReportChanges property 1059
- ReporterPLUS 1195
 - generate list 213
 - launching 1196
 - list of ports 1201
 - MODAF 1368
 - reports to examine models 1425
 - requirements diagrams 1202
 - viewing reports online 1201
- Reports 153, 1195
 - customize templates 1196
 - DoDAF 1321, 1331
 - for presentations 1195
 - formal 1195
 - generating from templates 1200
 - HTML 1195, 1200
 - internal output 1205
 - layout for systems engineering 1202
 - overridden properties 1204
 - PowerPoint 1195, 1200
 - prepackaged templates 1196
 - ReporterPLUS 153
 - requirements 1202
 - RTF 1195
 - RTF character set 1205
 - System Architect (SA) import 1297
 - system model templates 1201
 - templates 153, 1196
 - text format 1195
 - view overridden properties 1204
 - viewing online 1201
 - Word format 1195
- Reposition windows 18
- Repository 128, 153
- Repository file 264
- Represented field 584
- Represents field 644
- Required interface 90
 - for rapid ports 98
- Requirements 1, 7, 365, 1223, 1247
 - defining 8
 - defining in use cases 1255
 - dependencies 1248, 1255
 - derive 1248
 - diagrams 1245
 - hierarchical 367
 - icon in browser 300
 - identifying 7
 - importing 1243
 - in SysML 1227
 - listed in table views 224
 - Rational DOORS 1391, 1394
 - reports 1202
 - satisfaction of 1247
 - searching 1245
 - specialized types 1249
 - specification supports Asian languages 346
 - specifications option 346
 - synchronization 1395
 - systems engineering 1242
 - tabular view for SysML 1250
 - trace 1248
 - tracing in use cases 1255
 - verification of 1247
 - view 302
- Requirements diagrams
 - allocations in 1247
 - create package in 1247
 - dependencies 1247
 - derivations 1247
 - icon in browser 301
 - output in ReporterPLUS 1202
 - problem 1247
 - rationale 1247
 - use cases in 1247
- Re-route line 420
- RES file 1189
- Reset instance groups 719
- Resize element 434
- Respect 1063
- RespectCodeLayout property 1039, 1043, 1065
- RespectProfile 208
- RestrictedMode property 1060
- Return codes 1443

- Return type
 - \$opRetType keyword 172
- Return Value box 684
- Return values, animating 691, 1123
- Returns
 - primitive operations 73
- ReturnType property 112
- Reusable statechart
 - local termination code 732
- Reverse engineering 136, 985, 986
 - #define 999
 - #if...#ifdef...#else...#endif 1000
 - adding wild card expression 1023
 - additional user-defined keywords 1001
 - analyzing list of files 1007
 - analyzing same name header files 1005
 - code respect 1037, 1063
 - comments 1039
 - confirm settings 986
 - directing output 1036
 - enumerated types 1038
 - excluding particular files 992
 - external elements 994
 - files 1009
 - Filtering tab 1016
 - imported items report 990
 - importing object model diagrams 987, 1034
 - include files 1003
 - include statements 1004
 - include/CLASSPATH 997
 - initializing the Reverse Engineering window 991
 - Input tab 1003
 - Java 997
 - JavaAnnotations 1220
 - Javadoc comments 1211
 - legacy code 600, 1030
 - log file 265
 - Log tab 1035
 - lost constructs 1045
 - macros 1043
 - mapping classes 1008
 - mapping classes as types 1023
 - Mapping tab 1008
 - Merging existing packages 1033
 - Model Updating tab 1033
 - NO_OUTPUT_WINDOW variable 197
 - options 989
 - overwrite existing packages 1044
 - overwriting existing packages 1033
 - preprocessing 995
 - Preprocessing tab 995
 - preserving comments 1039
 - Process tab 1031
 - properties 1022
 - restrictions 985
 - results 1044
 - root directory 1014
 - static blocks 1212, 1214
 - static import statements 1212
 - templates in C++ 1037
 - to create external elements 600
 - tree view 987
 - unions 1038
- Reversed
 - attribute 94
 - messages 845
 - ports 80
- ReverseEngineering.log file 197, 265
- Rhapsody.exe 1439
- rhapsody.ini 196, 616
 - animation port, other than default 1444
 - control DMCA 1049
 - favorites list 307
 - General section 1416
 - hide Output window 1036
 - placement of GUI elements 250
 - plug-in information 980
- RhapsodyCL.exe 1439
- RM macro 175
- RMDIR macro 175
- Role names for classifiers 679
- Role of field 556
- Root directory for reverse engineering 1014
- RootDirectory property 1014
- ROPES process 7
 - analysis phase 7
 - design phase 8
 - implementation phase 9
 - testing phase 9
- rose_properties_import.xml 1413
- rose_properties_import_java.xml 1413
- Rotate on synchronization bar 640
- Roundtripping 86, 1047, 1049
 - annotations 1052
 - arguments 1054
 - associations 1054
 - attributes 1054
 - automatic 1050
 - classes 1054
 - classes, supported modifications 1053
 - code respect 925, 1047, 1063
 - constructor 1054
 - deletion of elements from the code 1057
 - destructors 1054
 - DMCA 1049
 - edited code 87
 - events 1055
 - forcing 1050
 - friend dependency 1058
 - functions 1055
 - limitations 1048
 - menu option 577
 - operations 1054
 - package supported modifications 1055

- preserving comments 1039
 - properties 1059
 - relations 1055
 - restricted mode 1060
 - static blocks 1212, 1214
 - static import statements 1212
 - supported elements 1048
 - templates in C++ 1058
 - text edits 87
 - triggered operations 1055
 - variables 1056
 - RoundtripScheme property 1039, 1043, 1060, 1064
 - rpyRetVal variable 653
 - RTF
 - reports 1195
 - storage format for Asian text 327
 - viewing reports in 1201
 - RTFCharacterSet property. 1205
 - Rules compiler 980, 981
 - RulesComposer 977, 980, 981
 - RulesPlayer 978, 980
 - Running
 - animation 1086
 - animation automatically 1128
 - animation scripts 1128
 - executable 918
 - Rational Rhapsody from command line 1439
- ## S
- Samples 656, 1431
 - Architect for Software edition 240
 - CustomCG 980
 - Designer for Systems Engineers edition 238
 - Extensibility 888
 - helper (.hep) applications 476
 - Simple Plug-in 512
 - System 1249
 - satisfy stereotype 1249
 - Save As command 217
 - Saving
 - diagrams 256
 - log files 265
 - option settings to a file 711
 - projects 217
 - units in separate directory 258
 - units individually 256
 - viewing preferences 262
 - window preferences 263
 - workspaces 262
 - Scale 457
 - Scenarios 1236, 1243
 - creating reusable 675
 - Schedulability, Performance, and Time (SPT)
 - profile 208, 277
 - Scope 297
 - code generation in activity diagram 653
 - code view editor 921
 - configuration 863
 - derived 863
 - explicit 863
 - Screen snapshot 1465
 - Scripts 1442
 - animation 1126, 1128
 - Java 1189, 1190
 - on Web server 1184
 - overwriting placeholder text 1189
 - tags 1191
 - uploading 1186
 - VBA 259
 - SDL 292
 - profile 208
 - SDLBlock 292
 - animation 293
 - behavior ports 293
 - SDLSignalPrefix property 292
 - Search 137, 138, 209
 - customize criteria 138
 - display results 139
 - working with results 139
 - Search and replace 35, 212
 - automatic 344
 - element name 343
 - elements 348
 - in model 342
 - limitations 359
 - text 344
 - using internal code editor 406
 - Search facility 1280
 - Search Results tab 32
 - Search window (Rational Rhapsody Platform Integration) 62
 - Searching
 - advanced capabilities 210
 - delete item 212
 - elements 345
 - references 211
 - results changes 211
 - within fields 346
 - Select Association option 563
 - Select Information Flow option 583
 - Select tool 795
 - Selecting
 - association in OMD 564
 - elements 430
 - messages 685
 - multiple elements 431
 - Selection handles 431
 - Selective instrumentation 866
 - Self transitions 635
 - Self-directed message 683
 - Send Action 624, 756, 1260
 - code generation 756, 757
 - display options 757

- event 756
- graphical behavior 757
- properties 757
- target 756
- Sequence
 - field 684
 - UCD 528
- Sequence diagrams 4, 5, 669
 - actor line 695
 - animated 1115
 - animated, partial 1091
 - animated, show transition states 1119
 - animating return values 691
 - auto-create instance lines 1117
 - automatic animation 703
 - cancelled timeout 674
 - comparison 707, 711
 - comparison excluding a message 713
 - comparison of instance groups 715
 - comparison of message arrival times 710
 - comparison of message groups 720
 - comparison, algorithm 707
 - comparison, color coding 709
 - comparison, saving options to a file 711
 - condition mark 675, 693
 - create arrow 674
 - created from activity diagrams 1234
 - creating 44, 674
 - creating message in 681
 - data flow 674
 - dataflows 696
 - deleting 706
 - destroy arrow 674, 693
 - destruction event 675
 - drawing tools 674
 - editor 13
 - execution occurrence 675
 - files 264
 - found message 675
 - generated from activity diagrams 1263
 - icon in browser 301
 - instance line 674, 677
 - IntelliVisor information 465
 - interaction occurrence 675, 698
 - interaction operator 675
 - interaction operator separator 675
 - layout 670
 - link wizard 1240
 - lost message 675
 - menu in browser 706
 - message 674, 681
 - message types 687
 - messages 1121
 - navigating to reference 699
 - part decomposition 700
 - partition line 674, 698
 - property for sequence numbers 689
 - reference diagrams 698
 - reply message 674
 - reusable scenarios 675
 - sequence numbers 689
 - shifting elements with mouse 705
 - system border 674, 676
 - time interval 674, 695
 - timeout 674
 - timeouts 694
- Serialization
 - generating methods 970
 - implementation methods 971
 - reactive instances 970
- Server
 - default Tornado target 199
- Service Oriented Architecture (SOA) 272
- Service ports 973, 1264
- SetDeviceName function 1193
- SetHomePageUrl function 1194
- SetPropPortNumber function 1194
- SetRefreshTimeout function 1194
- setSeparateSaveUnit 259
- SetSignaturePageUrl function 1194
- Settings 314, 377
 - backward compatibility profiles 377
 - WindowPos variable 198
- Settings tab 864
- Severity checks 883
- S-functions 1435
- Shadows 1400
- Shortcuts 1459
- Shortcuts, keyboard 399
- show command 1159
- Show Inherited option 538
- Show Labels 297
- Show Relations in New Diagram menu command 82, 323, 956, 1376
- Show Roles Labels option 563
- ShowAnimCancelTimeoutArrow property 1123
- ShowAnimCreateArrow property 1123
- ShowAnimDataFlowArrow property 1123
- ShowAnimDestroyArrow property 1123
- ShowAnimStateMark property 1120
- ShowAnimTimeoutArrow property 1123
- ShowArguments property 682
- ShowAttributes 414
- ShowCGSimplifiedModelPackage property 979
- ShowCleanImportData 1416
- ShowContainerElementForPorts property 233
- ShowLabels 327
- ShowLogViewAfterBuild property 31
- ShowOperations 414
- ShowPorts property 96
- ShowPortsInterfaces property 96
- Signature page
 - changing using the GUI 1174
 - changing using webconfig.c file 1194

- Silent animation 1112
- Silent mode 1094, 1112
- Simplification 979
- Simplified models 979
- Simulation 1286
 - setting scope 1287
- Simulink 1432
 - configuration 1436
 - flowports 1435
 - profiles 1433
- Simulink integration 1431
- SimulinkBlock 1431, 1433
- Slanted message 683
- Slider control 807
- Slider tool 795
- Smart generation 913
- Snapshot 1465
 - viewing system 537
- Socket mode
 - command-line interface 1440
- Software
 - configuration management 149
 - developers 127
 - development in Eclipse 280
 - prerequisites 128
 - product lines 241
- Solaris, Rational DOORS settings 1392
- Source of transition 735
- SourceArtifacts 1039, 1065
- Special characters 251
- Specialization parameters 123
- Specification field 369
- Specification files 1054, 1055
- SpecificationEpilog property 949
- SpecificationExtension property 988
- SpecificationHeader 945
- SpecificationProlog property 949
- Specifications
 - behavior 1286
 - comment 346
 - requirements option 346
 - UML 4
- Specify object values 537
- SpecInclude property 576
- SpecIncludes property 1013
- Spline line 420
- SPT profile 208, 277
- Stamp mode 418
- Standard Diagram toolbar 41, 455
- Standard Headers field 851, 864
- Standard operation 966
- Standard toolbar 35
- StandardOperations property 966
- Statechart mode 650, 725, 732
- Statecharts 4, 5, 726, 728, 1286
 - "And" line in 758
 - active transitions 768
 - And states 726
 - animated 1124
 - basic state 726
 - compound transitions 738
 - created from class for Harmony 1235
 - creating 44
 - creating new 409
 - decision node 727
 - dependency 728
 - description 730
 - diagram connector 728, 762
 - drawing options 727
 - editor 13
 - else branch 760
 - EnterExit point 728
 - events 742
 - events and operations 751
 - flat 865
 - for a use case 523
 - forks 739
 - guards 747
 - history connector 727, 761
 - icon in browser 301
 - inheritance 771
 - initial connector 750
 - inlining 786
 - inlining code 786
 - IntelliVisor information 466
 - IS_IN 781
 - join transitions 728
 - junction connector 727, 762
 - leaf state 726
 - local termination code 732, 733
 - local termination rules 651
 - merge sub-statechart 766
 - messages 751
 - operations 751
 - Or states 726
 - overriding inheritance 776
 - parent 765
 - reusable 865
 - SDLBlock 293
 - semantics 767
 - Send Action 756
 - send action 728
 - separate transitions 728
 - serialization 970
 - single-action 770
 - states 726
 - static reactions 774
 - sub-statecharts 728
 - Tabular 787
 - termination connector 727, 767
 - termination state 728
 - timeout triggers 746
 - transition labels 728, 741
 - transitions 735

- transitions in 735, 741
- triggers 741
- Updating EnterExit points 764
- StateBlock profile 1386
- States 726
 - action 622, 658, 1260
 - activity diagrams 622
 - And and code generation 733
 - basic 726
 - block 628, 660
 - creating 728
 - drawing 728
 - final 650
 - leaf 726
 - names 728
 - Or 726
 - Or and code generation 732
 - Or and local termination code 733
 - subactivity 1261
 - termination 728, 732
 - transitions between 1262
- Static
 - attribute 934
 - attributes 71
 - modifier 70
- Static architecture 959
 - memory allocation algorithm 962
- Static blocks 1213
- Static import 1212
- Static memory allocation 960
 - algorithm 962
 - conditions 963
 - limitations 963
- Static reactions 774
- Statistics page 1187
- Status of an object 1159
- Stereotypes 375, 385, 1248
 - associating a bitmap 387
 - associating with element 385
 - automotiveC profile 1382
 - browser icon 300
 - change order 387
 - composite requirement 1249
 - conform 1226
 - copying MOEs 1235
 - defining a tag 390
 - deleting 388
 - dependency 1256
 - derive 84, 1248
 - derive requirement 1249
 - display for compartment lists 454
 - display with elements in browser 335
 - extend requirement 1249
 - graphical representation 336
 - inheritance 389
 - New Term 505
 - new term 385, 386, 389
- NewTerm 53
- problem 1226
- refine requirement 1249
- satisfy requirement 1249
- SDLBlock 292
- serviceContract 273
- serviceProvide 273
- show label 88
- special 389
- specilized requirement types 1249
- static for variants 242
- SysML allocation 1226
- SysML blocks 1226
- SysML model element 1226
- SysML requirements 1227
- trace requirement 1249
- verify requirement 1249
- store.log file 265
- Straight line 420
- Strategic viewpoint (MODAF) 1345
- StrictExternalElementsGeneration 604
- Structural view 5
- Structure diagrams 4, 5, 238, 239, 829, 1264
 - compared to OMD 829
 - composite class 830
 - creating 44
 - dependencies 835
 - drawing tools 830
 - files 264
 - flows 835
 - icon in browser 301
 - IntelliVisor information 468
 - links 835
 - objects 830, 831
 - ports 835
 - toolbar 830
- Structure type 109
- Structure, modeling as a type 1025
- Structured class 457
- Stub unit 262
- Subactivities 624
- Subactivity 629, 630
 - creating 630
 - creating from action blocks 629
- Subactivity diagrams 630
- Submachine 765
 - opening 765
- Submenu#Listproperty 505
- Submenu#Name property 505
- Submenu1List property 502
- Submenu1Name property 502
- Submenu2List property 502
- Submenu2Name property 502
- Submenu3List property 502
- Submenu3Name property 502
- Submenu4List property 502
- Submenu4Name property 502

- SubmenuList property 502, 505
- Subpackages 316
- Sub-statechart, merge into parent statechart 766
- Subsystems
 - operations allocated to 1238
- Subversion (SVN) 132
- Superclass 300, 462, 545, 548
 - add 549
 - prevent code generation for 549
- SupportExternalElementsInScope 604
- Suspend 1127
- SV-1 System Interface Description 1325
- SV-10a Systems Rules Model 1326
- SV-10b System State Transition Description diagram 1326
- SV-10c System Event-Trace Description diagram 1326
- SV-11 Physical Schema diagram 1326
- SV-2 System Communication Description diagram 1326
- SV-4 System Functionality Description diagram 1326
- SV-8 System Evolution Description diagram 1326
- SVM 132
- Swimlanes 621, 622, 624, 642, 1260, 1262
 - converted to life lines 1234
 - divider 643
 - dividers, deleting 644
 - frame 643
 - frames, deleting 645
 - limitations 645
 - subactivity limitation 1261
 - viewing in browser 644
- Switches, command-line 1444
- Swimlanes
 - consistency checks 1234
- Symbol
 - defined 998
 - preprocessor 995
 - undefined 1000
- Synchronization 638
 - bar constraints 634
 - bars 638
 - between client and server 745
 - code changes 603
 - fork bars 639
 - modifying bars 641
 - Rational Statemate and Rational Rhapsody 1388
- Synchronization bar 640
- Synchronization option 710
- Synchronous events 681
- SysML 2
 - action types 1257
 - activity diagrams 1257
 - activity modeling 1257
 - block definition diagrams 1225
 - diagrams 1226
 - dimension 1267
 - elements 1293
 - flow of control 1257
 - flows 1226
 - model element stereotypes 1226
 - model libraries 1226
 - parametric diagram 1272
 - ports 1226
 - profile 1245
 - project type 1224
 - report template 1201
 - requirements diagrams 1245
 - requirements in 1227
 - SA imported elements 1293
 - specialized requirement types 1249
 - Teamcenter 1298, 1300
 - units 1267
 - units and value types 274
 - valueType 1267
- SysML profile 208, 1224
 - allocation 1226
 - Architect for Software edition 240
 - packages 1226
 - requirements tabular view 1250
 - starting point 1225
 - with NetCentric 275
- System
 - border 676
 - border in ASDs 1121
 - boundary box 520
 - thread 1099
 - viewing snapshot 537
- System Architect (SA) 1293
 - creating SysML diagram from 1296
 - encyclopedia 1295
 - importing DoDAF elements 1295
 - mapping elements to Rational Rhapsody 1293
 - post processing 1297
- System engineering
 - activity view 1232
- Systems engineering 1223
 - accept event action 1259
 - action pins 623, 1259
 - activity diagrams 1257
 - activity diagrams drawing icons 1259
 - activity parameters 623, 1259
 - activity view 1234
 - actors 1252
 - architectural design wizard 1238
 - architecture 1265
 - block definition diagram 1265
 - create ports and interfaces 1236
 - creating a project 1223
 - defining dependencies 1248
 - design structure 1264
 - diagrams 1223
 - display options 1276
 - equations 1276
 - fork node 1263

- Harmony 1234
 - Harmony process 1230
 - initial flow 1261
 - link wizard 1240
 - modeling behavior 1286
 - modeling data types 274
 - organize activities 1262
 - requirements 1242
 - simulation 1286
 - statecharts 1286
 - stereotypes 1256
 - subactivity 1261
 - SysML profile 1224
 - system boundary 1252
 - tracing requirements 1255
 - transitions 1261
 - use case diagrams 1242, 1251
 - using WSDL files 273
 - validation 1286
 - version 1223
 - Systems view (DoDAF) 1313
 - Systems viewpoint (MODAF) 1346
- T**
- Table views 222, 226
 - attributes listed in 224
 - binding view and layout 234
 - customize for MODAF 1360
 - export data 236
 - layouts 223
 - layouts for SysML 1226
 - manage data 236
 - requirements listed in 224
 - Tags 354, 375, 390
 - browser icon 300
 - creating 390
 - deleting 393
 - DoDAF 1327
 - features 390
 - global 391
 - graphical representation 336
 - individual element 391
 - stereotype 390
 - tab 83
 - Target
 - building 916
 - default Tornado server 199
 - field 736
 - main() 172
 - name 172
 - of hyperlink 56
 - type 172
 - Target monitoring 21
 - Targets 916
 - for hyperlinks 58
 - icons for hyperlinks 57
 - Send Action 756
 - Teamcenter 1298
 - creating Rational Rhapsody model 1301
 - importing Rational Rhapsody model 1300
 - integration 1298
 - modifying elements shared with Rational Rhapsody 1301
 - prerequisites for working with Rational Rhapsody 1300
 - SysML 1298, 1300
 - UML 1298, 1300
 - viewing corresponding Rational Rhapsody elements 1302
 - Technical support 1461
 - new customers 1461
 - Technical view (DoDAF) 1313
 - Technical viewpoint (MODAF) 1346
 - Template check box 73
 - Template Class 122
 - Template instantiation 123
 - Template Instantiation Argument window 125
 - Template specialization 122, 123
 - Templates 122, 1196
 - classes 66
 - code generation 126
 - customizing report 1199
 - DiffMerge tool 122
 - for code-based documentation systems 944
 - instantiation 66
 - limitations 126
 - ModafReport.tpl 1370
 - parameters 126
 - reverse engineering (C++ only) 1037
 - roundtripping (C++ only) 1058
 - system model 1201
 - Termination
 - connector 763
 - state 732
 - Terminology
 - in customized profile 385
 - Test bench 1235
 - TestConductor 265
 - Testing
 - application on remote target 1088
 - phase 9
 - Testing profile for OMG 209
 - TestingProfile 1235
 - Text
 - adding to file 854
 - editing 452
 - editor 392
 - format 42, 439
 - in Asian languages 327
 - selecting editor 855
 - Text Box control 806
 - Text Box tool 795
 - Third-party interfaces 16

- this_ variable 653
- Threads 1147
 - active 1100
 - animation 1098
 - focus in animation 1098
 - focus in tracer 1158
 - mainThread 1099
 - multiple 319, 1099
 - naming in animation 1099
 - naming in tracing 1147
 - priority 1100
 - resuming 1158
 - suspending 1162
 - system 1099
- TIFF 364
- Tile, maintaining window content 23
- Time
 - behavior modeling 785
 - interval 674, 695
 - message arrival 710
 - Model field 864
 - model setting 172
 - out trigger 746
 - outs 1, 694
 - real setting 864
 - real-time environment 1
 - simulated setting 864
 - stamp 218, 1127
 - stamp command 1162
- Timeout 674
 - cancelled 674, 694
 - creating 694
 - trigger 746
- timestamp command 1162
- Tip section
 - FilePos variable 198
 - StartUp variable 198
 - TimeStamp variable 198
- tm() keyword 746
- To-many relations 956
- Tool
 - Break 1095
 - Call operations 692, 1108
 - Command Prompt 1095
- Toolbars
 - Animation 40, 1093
 - arrange 448
 - Code 37
 - Favorites 40, 307
 - Format 42, 439
 - Free Shapes 42, 422
 - Layout 42, 450
 - modeling 40
 - Standard 35
 - structure diagram 830
 - VBA 40
 - Windows 39
 - zoom 41, 455
- Tools
 - Common drawing 41
- Tools menu 474, 503
- To-one relations 956
- Tornado 289, 290, 291
 - DefaultTargetServerName variable 199
- Trace 1127, 1156
 - calling operations 692, 1108
 - command 1162
 - field 866
 - requirements 1248
 - selective 866
- trace stereotype 1250
- Tracer 1143
 - break command 1148
 - CALL command 1151
 - commands 1126, 1148
 - display commands 1153
 - ending a session 1145
 - GEN commands 1153
 - go command 1154
 - help command 1155
 - input 1155
 - input command 1155
 - messages 1167
 - output command 1157
 - output destination 1157
 - quit command 1157
 - show command 1159
 - starting a session 1144
 - stepping through the application 1154
 - timestamp command 1162
 - trace command 1162
 - using 1143
 - watch command 1166
- Trade analysis option 1235, 1237
- Transition states 1119
- Transitions 735, 1261
 - active 768
 - activity diagrams 634
 - adding actions on 1277
 - completion 635
 - compound 738
 - conflicts 768
 - context 735
 - context message parameters 783
 - creating activity diagram 634
 - execution 770
 - fork 739
 - in statecharts 735
 - join 624, 728, 739, 747
 - join for activity diagrams 636
 - label 623, 728
 - label for activity diagrams 636
 - label for statechart 741
 - labels 636

- labels in statecharts 741
- loop for activity diagram 635
- null activity diagrams 650
- null in statecharts 747
- priorities 768
- separate 624, 728
- statechart 735
- to self 635
- types 738
- Translative code generation 910
- Transparency 388
- Trigger 741
 - field 736
 - null 747
 - selecting 740
 - timeout 746
- TriggerArgument property 112
- Triggered operations 75, 688
 - applying 745
 - replies 745
 - roundtripping 1055
 - statechart 745
- Troubleshooting
 - clean configuration 921
 - code generation 921
 - connecting to models from the Web 1177
 - controlled files 356
 - DoDAF 1334, 1337
 - MODAF 1373, 1374
 - MODAF and ReporterPLUS 1371
 - Rational Statemate with Rational Rhapsody 1389
 - ReporterPLUS and MODAF 1371
 - Simulink 1434
- Type declarations search option 347
- Type field 851, 1224, 1232
 - block 534, 832
- Typedef 110, 1024
- Types 320
 - \$Type keyword 174
 - block definition diagram value 1267
 - composite 106
 - creating 106
 - creating enumerated 107
 - creating languages 108
 - creating structure 109
 - creating typedef 110
 - creating union 110
 - data 274
 - editing the order of 115
 - field 114
 - language 109
 - language-independent 113
 - logical file 853
 - message 687
 - modifying 70
 - nested 67
 - predefined 113

- primitive operations 73
- receptions 75
- roundtripping user-defined 1055
- transitions 738
- user-defined 1154

U

- UML 2, 1425
 - diagrams 4, 205
 - dynamic behavior view 5
 - export format to XMI 1426
 - export versions 1426
 - import format from XMI 1429
 - MODAF 1342
 - signal 73
 - structural view 5
 - Teamcenter 1298, 1300
 - views 5
- Undefined symbol 1000
- Undo 216
 - limitations 216
 - operations that cannot be undone 216
 - zoom 457
- Undo/Redo internal code editor 405
- Undock Features window 51
- Unicode characters 314
- Unified Modeling Language
 - diagrams 4
 - dynamic behavior views 5
 - structural views 5
- Union 110
 - creating 110
 - reverse engineering 1038
- UNISYS format for diagrams 1426, 1428
- Units 253
 - access privileges 255
 - adding to model 136
 - adding to workspace 262
 - browser icon 299
 - characteristics 254
 - ClassIsSavedUnit property 258
 - comparing 266
 - creating 255
 - elements saved as 246
 - file extension 255
 - icons 246, 256
 - importing 136
 - loading 262
 - loading and unloading 256
 - modifying 256
 - names 254
 - projects 253
 - referencing 246, 254
 - saving in separate directories 258
 - saving individual 256
 - separating project into 255

- stub 262
- SysML 1267
- Unit View 150
- unloaded 262
- unresolved 262
- view 302
- Unloaded unit 262
- Un-override 180
- Unresolved references 261, 278
- Unresolved unit 262
- Update-on-Break mode 1112
- UPDM
 - profile 209
- Upload file 1184
- Upload to a File Server page 1186
- Usage of elements 342
- UsageType property 611
- Use case diagrams 4, 5, 517, 1251
 - associations 527
 - automatically populating 412, 415
 - boundary box 1252
 - creating 410, 519
 - defining SysML requirements 1255
 - dependencies 528, 1255
 - drawing icons 519
 - editor 13
 - files 264
 - flow of information 1256
 - generalizations 527
 - icon in browser 301
 - IntelliVisor information 468
 - packages 526
 - Rational Rhapsody Gateway 1242
 - system boundary box 520
 - tracing requirements 1255
- Use cases 321, 520, 1251
 - actors with 1254
 - attributes 521
 - browser icon 300
 - creating 520
 - creating a statechart 523
 - define features 1253
 - drag and drop 331
 - in requirements diagrams 1247
 - operation 522
 - requirements 1255
 - view in browser 302
- Use Default check box 864
- Use existing type field 69, 73
- UseAsExternal property 1016
- UseCalculatedRootDirectory property 1015
- UseDescriptionTemplates property 1209
- UseIncrementalSave property 218
- User Points option 420
- UseRapidPorts property 292
- User-defined checks 881, 886
- User-defined keywords (reverse engineering) 1001

- User-defined type 1154
 - creating 69
 - generate event of 1096
- Typedefs 1024
- UseRemoteHost property 1086
- Utilities 15, 1318

V

- Validation 1286
- Value field 537
- Value, specifying for instances 537
- VariableInitializationFile property 319
- Variable-length argument list 935
- VariableLengthArgumentList 935
- Variables 319
 - adding to a diagram 596
 - create 319
 - environment used by Rational Rhapsody 196
 - global 319
 - ordering 319
 - roundtripping 1056
 - rpyRetVal 653
 - tab 590
 - this_ 653
- Variants 241, 242
 - static stereotype 242
- Variation points 241
- VB
 - versus VBA programs 481
- VBA 21, 482
 - adding macros 479
 - keyboard shortcuts 1456
 - macros 482
 - project file 265, 481
 - toolbar 40
 - using with the Rational Rhapsody API 481
 - versus VB programs 481
- verify stereotype 1250
- Video capture 1465
- View
 - components 302
 - conform 1228
 - designing 1181
 - diagrams in browser 302
 - entire model 302
 - generated code 921
 - internal code editor options 395
 - model from the Web 1183
 - overridden properties with browser filter 302
 - property filters 180
 - property tables 179
 - removing elements from 451
 - requirements 302
 - scale 457
 - split 403
 - Unit View 150

-
- use case 1409
 - use cases in browser 302
 - Viewpoint 1228
 - Viewing preference 262
 - Viewpoints 5, 1226
 - adding 1228
 - Views
 - packages 1227
 - Viewport 23
 - Views 5, 1226, 1227
 - creating 1227
 - hide empty cells 232
 - include descendants 226, 230
 - include ports 224, 233
 - matrix 230
 - show relations 224, 233
 - specification 457
 - structured 457
 - SysML requirements 1250
 - table 226
 - Visibility
 - attributes 69
 - destructors 79
 - primitive operations 73
 - receptions 75, 77
 - Visual programming environment (VPE) 155
 - Visual Studio 288
 - as IDE 279
 - Visualization Only (Import as External) 986
 - Visualize eternal elements 994
 - VPE 155
- W**
- Warning 881
 - Watch command 1166
 - Watch mode 1094, 1112
 - Web browser, navigating to a model 1176
 - Web GUI 1183, 1184
 - Web pages
 - adding Rational Rhapsody functionality to 1189
 - automatic refresh rate 1194
 - binding embeddable objects 1190
 - changing home page in webconfig.c file 1194
 - Define View 1180
 - device name 1193
 - navigation 1178
 - Personalized Navigation page 1181
 - Web server 1171
 - changing port number 1194
 - customizing 1193
 - generating Web site 1176
 - port 1174
 - Web services 1185
 - libraries 1185
 - Statistics page 1187
 - Upload to a File Server page 1186
 - webconfig.c file 1186, 1193, 1194
 - Web-enable 1288
 - a model 1288
 - configuration 1288
 - interface 1291
 - property 1290
 - sending events to a model 1292
 - setting stereotype 1290
 - Web-enabled devices 1171, 1178
 - adding files to model 1184
 - Advanced Settings 1174
 - connecting to a model 1176
 - controlling 1183
 - customizing the GUI 1184
 - Define View page 1180
 - limitations 1173
 - name/value pairs 1183
 - Personalized Navigation page 1181
 - properties for 1175
 - setting elements as 1172
 - troubleshooting 1177
 - viewing 1183
 - Webify 864, 1172, 1288
 - setting parameters 1174
 - Toolkit 1171, 1185
 - WebManaged property 1175
 - Web-services Definition Language (WSDL) 208, 272
 - exporting 276
 - generating 275
 - Welcome Screen 33
 - Window
 - characteristics 17
 - docking 18
 - drawing area 23
 - keyboard shortcuts 1456
 - maintaining the content 23
 - output 24
 - position 198
 - preference 263
 - properties for internal code editor 395
 - repositioning 18
 - undocking 18
 - viewport 23
 - Windows 127, 352
 - browsers 1201
 - Internet Explorer 1428
 - navigation buttons 39
 - toolbar 39
 - viewing reports 1201
 - Wizards
 - Architectural Design 1238
 - Link 1240
 - Workflow 599, 994
 - Workflow Integration (Rational Rhapsody and Eclipse) 127
 - Workspace 262
 - adding units to 262
-

- creating 262
- file 265
- opening 263
- saving 262
- window preferences 263

X

- XMI 1425
 - examining exported file 1428
 - export action pins 650
 - export activity parameters 650

- export as version 2.1 1426
- importing 1429
- in development 1425
- SysML support for version 2.1 1225
- XML Metadata Interchange (XMI) 1425

Z

- Zoom 41, 455
 - scaling percentage 457
 - undoing 457
- Zoom to Fit button 457