Rational Software Corporation

Add-in, Tool, and Wizard Guide

RATIONAL ROSE® REALTIME

VERSION: 2003.06.00

PART NUMBER: 800-026122-000

WINDOWS/UNIX

support@rational.com
http://www.rational.com

Legal Notices
©2003, Rational Software Corporation. All rights reserved.

Part Number: 800-026122-000
Version Number: 2003.06.00

This manual (the "Work") is protected under the copyright laws of the United States and/or other jurisdictions, as well as various international treaties. Any reproduction or distribution of the Work is expressly prohibited without the prior written consent of Rational Software Corporation.

Rational, Rational Software Corporation, the Rational logo, Rational Developer Network, AnalystStudio, ClearCase, ClearCase Attache, ClearCase MultiSite, ClearDDTS, ClearGuide, ClearQuest, ClearTrack, Connexis, e-Development Accelerators, DDTS, Object Testing, Object-Oriented Recording, ObjecTime, ObjecTime Design Logo, Objectory, PerformanceStudio, PureCoverage, PureDDTS, PureLink, Purify, Quantify, Rational Apex, Rational CRC, Rational Process Workbench, Rational Rose, Rational Suite, Rational Suite ContentStudio, Rational Summit, Rational Visual Test, Rational Unified Process, RUP, RequisitePro, ScriptAssure, SiteCheck, SiteLoad, SoDA, TestFactory, TestFoundation, TestStudio, TestMate, VADS, and XDE, among others, are trademarks or registered trademarks of Rational Software Corporation in the United States and/or in other countries. All other names are used for identification purposes only, and are trademarks or registered trademarks of their respective companies.

Portions covered by U.S. Patent Nos. 5,193,180 and 5,335,344 and 5,535,329 and 5,574,898 and 5,649,200 and 5,675,802 and 5,754,760 and 5,835,701 and 6,049,666 and 6,126,329 and 6,167,534 and 6,206,584. Additional U.S. Patents and International Patents pending.

U.S. GOVERNMENT RIGHTS. All Rational software products provided to the U.S. Government are provided and licensed as commercial software, subject to the applicable license agreement. All such products provided to the U.S. Government pursuant to solicitations issued prior to December 1, 1995 are provided with “Restricted Rights” as provided for in FAR, 48 CFR 52.227-14 (JUNE 1987) or DFARS, 48 CFR 252.227-7013 (OCT 1988), as applicable.

WARRANTY DISCLAIMER. This document and its associated software may be used as stated in the underlying license agreement. Except as explicitly stated otherwise in such license agreement, and except to the extent prohibited or limited by law from jurisdiction to jurisdiction, Rational Software Corporation expressly disclaims all other warranties, express or implied, with respect to the media and software product and its documentation, including without limitation, the warranties of merchantability, non-infringement, title or fitness for a particular purpose or arising

from a course of dealing, usage or trade practice, and any warranty against interference with Licensee's quiet enjoyment of the product.

Third Party Notices, Code, Licenses, and Acknowledgements
Portions Copyright ©1992-1999, Summit Software Company. All rights reserved.

Microsoft, the Microsoft logo, Active Accessibility, Active Client, Active Desktop, Active Directory, ActiveMovie, Active Platform, ActiveStore, ActiveSync, ActiveX, Ask Maxwell, Authenticode, AutoSum, BackOffice, the BackOffice logo, bCentral, BizTalk, Bookshelf, ClearType, CodeView, DataTips, Developer Studio, Direct3D, DirectAnimation, DirectDraw, DirectInput, DirectX, DirectXJ, DoubleSpace, DriveSpace, FrontPage, Funstone, Genuine Microsoft Products logo, IntelliEye, the IntelliEye logo, IntelliMirror, IntelliSense, J/Direct, JScript, LineShare, Liquid Motion, Mapbase, MapManager, MapPoint, MapVision, Microsoft Agent logo, the Microsoft eMbedded Visual Tools logo, the Microsoft Internet Explorer logo, the Microsoft Office Compatible logo, Microsoft Press, the Microsoft Press logo, Microsoft QuickBasic, MS-DOS, MSDN, NetMeeting, NetShow, the Office logo, Outlook, PhotoDraw, PivotChart, PivotTable, PowerPoint, QuickAssembler, QuickShelf, RelayOne, Rushmore, SharePoint, SourceSafe, TipWizard, V-Chat, VideoFlash, Visual Basic, the Visual Basic logo, Visual C++, Visual C#, Visual FoxPro, Visual InterDev, Visual J++, Visual SourceSafe, Visual Studio, the Visual Studio logo, Vizact, WebBot, WebPIP, Win32, Win32s, Win64, Windows, the Windows CE logo, the Windows logo, Windows NT, the Windows Start logo, and XENIX, are either trademarks or registered trademarks of Microsoft Corporation in the United States and/or in other countries.

Sun, Sun Microsystems, the Sun Logo, Ultra, AnswerBook 2, medialib, OpenBoot, Solaris, Java, Java 3D, ShowMe TV, SunForum, SunVTS, SunFDDI, StarOffice, and SunPCi, among others, are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Purify is licensed under Sun Microsystems, Inc., U.S. Patent No. 5,404,499.

Licensee shall not incorporate any GLOBEtrotter software (FLEXlm libraries and utilities) into any product or application the primary purpose of which is software license management.

BasicScript is a registered trademark of Summit Software, Inc.

Design Patterns: Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Copyright © 1995 by Addison-Wesley Publishing Company, Inc. All rights reserved.

Additional legal notices are described in the legal_information.html file that is included in your Rational software installation.

Contents

Preface ix
Audience. ix
Other Resources ix
Rational Rose RealTime Integrations With Other Rational Products x
Contacting Rational Customer Support xi
1 Rational Rose RealTime Add-ins (Not Language-specific) 13
Add Class Dependencies Add-in 13
Features of the Add Class Dependencies Add-In. 14
Considerations 18
Searching for Missing Dependencies 19
Adding Missing Dependencies 19
Specifying Dependencies to Create 20
Add External Java Tool. 20
Add External Java Dialog 21
Accessible Model Locations 21
Import Options 22
Specifying a JAR Utility in Your Path 24
Generate Documentation Add-In 24
Source Code Assists and Make Files Writable Add-in 26
Select Checked Out Units in Browser. 26
Show Unit Versions 27
Submit All Changes to Source Control 27
Make Files Writable 27
Make Files ReadOnly 28
2 Rational Rose RealTime Tools 29
Aggregation Tool. 29
Relationships 29
Aggregation Tab 30
EndA and EndB 33
Aggregation Tool - EndA and EndB Tabs 34
Descriptions 38
Advanced Tab 40

Attribute Tool 42
Attribute Tool: Properties Tab 42
Descriptions 43
Properties Tab: Language-Specific Options 47
Operation Tool 49
Operation Tool: Properties Tab 50
Descriptions 50
Properties Tab: Language-Specific Options 54
Dependency Tab for Attribute Tool and Operation Tool 56
Descriptions 57
Options: (C, C++) 57
Dependencies Tab: Language-Specific Options 59
Trace Tool 60
Configuring Rational RequisitePro for Traceability 61
Using the Trace Tool in Rational Rose RealTime 69
Using the Trace Tool in Rational RequisitePro 72
Updating Rational RequisitePro Requirements when a Model File Location Changes 73
3 Rational Rose RealTime Wizards 75
Component Wizard 75
TargetRTS Wizard 83
Understanding the TargetRTS 83
Maintaining TargetRTS Libraries using the TargetRTS Wizard 83
Managing Your TargetRTS Configurations 86
Duplicating a Configuration 86
NoRTOS Target Base 90
Editing a Configuration 90
Understanding the makefiles 91
Editing the Target 92
Descriptions 94
Editing the Libset 95
Descriptions 96
Editing a Configuration 96
Building Configurations 97
Descriptions 98

Deleting Configurations 99
Creating Ports Between C and C++ 100

Index 105

Preface

This manual describes the Rational Rose RealTime Add-ins (excluding the language-specific add-ins), tools, and wizards.

This manual is organized as follows:

- *Rational Rose RealTime Add-ins (Not Language-specific)* on page 13
- *Rational Rose RealTime Tools* on page 29
- *Rational Rose RealTime Wizards* on page 75

Audience

This guide is intended for all readers including managers, project leaders, analysts, developers, and testers.

This guide is specifically designed for software development professionals familiar with the target environment they intend to port to.

Other Resources

- Online Help is available for Rational Rose RealTime.

  Select an option from the **Help** menu.

  All manuals are available online, either in HTML or PDF format. To access the online manuals, click **Rational Rose RealTime Documentation** from the **Start** menu.

- To send feedback about documentation for Rational products, please send e-mail to `techpubs@rational.com`.

- For more information about Rational Software technical publications, see: `http://www.rational.com/documentation`.

- For more information on training opportunities, see the Rational University Web site: `http://www.rational.com/university`.

- For articles, discussion forums, and Web-based training courses on developing software with Rational Suite products, join the Rational Developer Network by selecting **Start > Programs > Rational Suite > Logon to the Rational Developer Network.**

Rational Rose RealTime Integrations With Other Rational Products

Integration	Description	Where it is Documented
Rose RealTime–ClearCase	You can archive Rose RT components in ClearCase.	▪ *Toolset Guide: Rational Rose RealTime* ▪ *Guide to Team Development: Rational Rose RealTime*
Rose RealTime–UCM	Rose RealTime developers can create baselines of Rose RT projects in UCM and create Rose RealTime projects from baselines.	▪ *Toolset Guide: Rational Rose RealTime* ▪ *Guide to Team Development: Rational Rose RealTime*
Rose RealTime–Purify	When linking or running a Rose RealTime model with Purify installed on the system, developers can invoke the Purify executable using the **Build > Run with Purify** command. While the model executes and when it completes, the integration displays a report in a Purify Tab in RoseRealTime.	▪ Rational Rose RealTime Help ▪ *Toolset Guide: Rational Rose RealTime* ▪ *Installation Guide: Rational Rose RealTime*
Rose RealTime–RequisitePro	You can associate RequisitePro requirements and documents with Rose RealTime elements.	▪ *Addins, Tools, and Wizards Reference: Rational Rose RealTime* ▪ *Using RequisitePro* ▪ *Installation Guide: Rational Rose RealTime*
Rose RealTime–SoDa	You can create reports that extract information from a Rose RealTime model.	▪ *Installation Guide: Rational Rose RealTime* ▪ *Rational SoDA User's Guide* ▪ SoDA Help

Contacting Rational Customer Support

If you have questions about installing, using, or maintaining this product, contact Rational Customer Support.

Your Location	Telephone	Facsimile	E-mail
North, Central, and South America	+1 (800) 433-5444 (toll free) +1 (408) 863-4000 Cupertino, CA	+1 (781) 676-2460 Lexington, MA	support@rational.com
Europe, Middle East, Africa	+31 20 4546-200 Netherlands	+31 20 4546-201 Netherlands	support@europe.rational.com
Asia Pacific	+61-2-9419-0111 Australia	+61-2-9419-0123 Australia	support@apac.rational.com

Note: When you contact Rational Customer Support, please be prepared to supply the following information:

- Your name, company name, telephone number, and e-mail address
- Your operating system, version number, and any service packs or patches you have applied
- Product name and release number
- Your Service Request number (SR#) if you are following up on a previously reported problem

When sending email concerning a previously-reported problem, please include in the subject field: "[SR#XXXXX]", where XXXXX is the Service Request number of the issue. For example, "[SR#0176528] - New data on rational rose realtime install issue ".

Rational Rose RealTime Add-ins (Not Language-specific)

1

Contents

This chapter is organized as follows:

- *Add Class Dependencies Add-in* on page 13
- *Add External Java Tool* on page 20
- *Generate Documentation Add-In* on page 24
- *Source Code Assists and Make Files Writable Add-in* on page 26

Note: You can click **Add-Ins > Add-In Manager** to activate or deactivate these add-ins.

Add Class Dependencies Add-in

The **Add Class Dependencies** add-in helps you identify missing dependencies in your model. With this add-in, you can examine your model to find dependencies that should exist but currently do not.

Missing dependencies are a common source of compilation errors. As Rational Rose RealTime compiles a capsule or class, the compiler must find the definition of the classes it uses. Consequently, you need to identify the capsules and classes that depend on other classes in your model. Also, if the interface for a class changes, the build process automatically rebuilds all the capsules and classes that depend on that class.

Use the **Add Class Dependencies** add-in to:

- Search for selected elements
- Search for a selected component
- Search all model elements
- Examine a component

The **Add Class Dependencies** add-in is available by clicking **Build > Add Class Dependencies**.

Note: The source code for this add-in is available in the $ROSERT_HOME/Scripts directory. You can customize the code in this script, as required.

The **Add Class Dependencies** add-in determines if dependencies are missing; however, it does not:

- find dependencies that are no longer required
- set a dependency to the correct strength (forward inclusion)

Features of the Add Class Dependencies Add-In

The **Add Class Dependencies** add-in includes the following features:

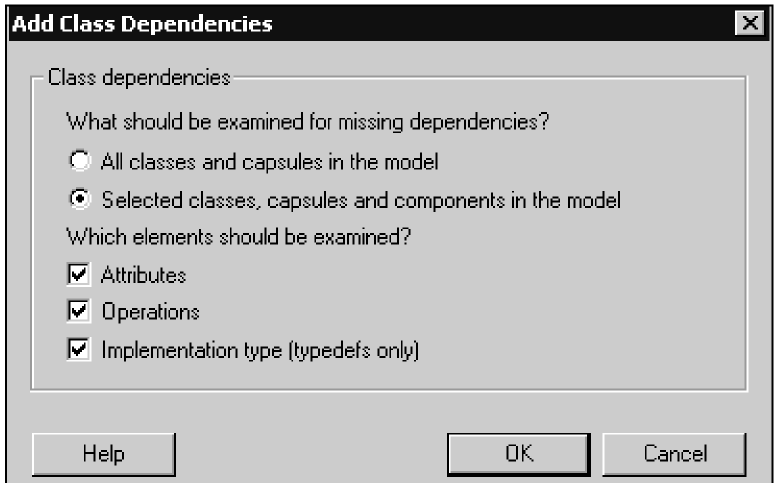
- Support for:
  - **RTJava** models
  - Read-only models
  - Fully qualified names
- A missing dependency with an ambiguous supplier. For example, if capsule A depended on class B and there is more than one class B in the model (for example, in different logical packages), the **Add Class Dependencies** add-in creates a dependency from capsule A to the first instance of class B. As a result, the following warning appears in the log:

  ```
  Warning: A dependency from A to the ambiguous Classifier B
  must be added manually.
  ```

  Double-click on the log entry to open the **Specification** dialog box for the provider of the missing dependency.
- You can scan the entire model, or only selected model elements for missing dependencies. For example, you can select one, or more, classes or capsules in the toolset, and then use the **Add Class Dependencies** add-in to identify missing dependencies for the selected model elements.

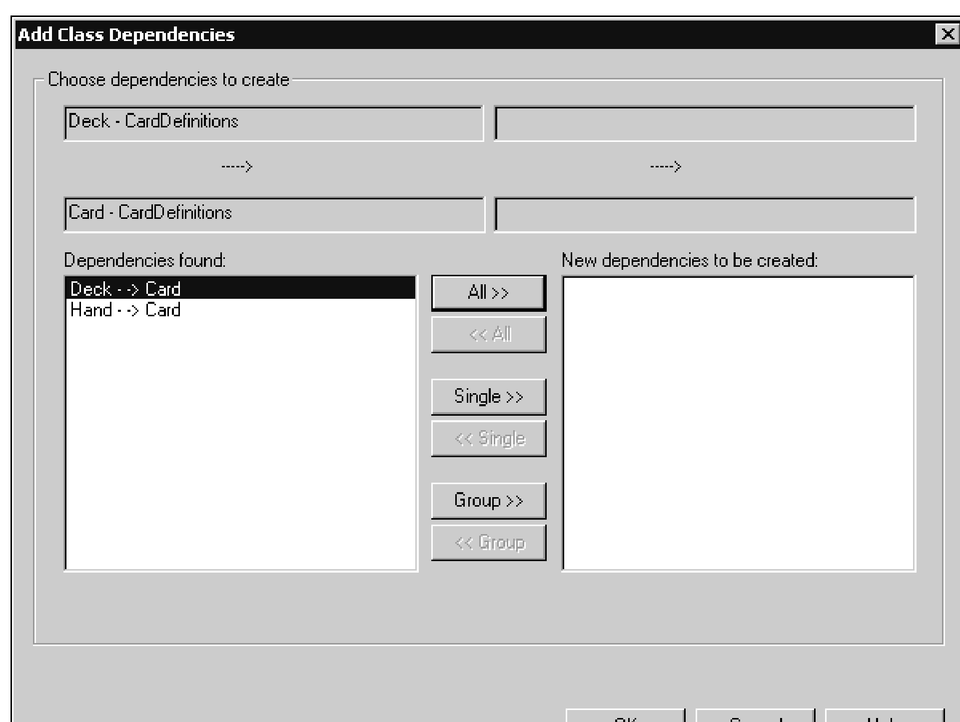
  Figure 1 shows that the **Selected classes, capsules and components in the model** option is automatically set when you select objects in the model prior to launching the **Add Class Dependencies** add-in.

Figure 1 Add Class Dependencies Add-in - First Panel



- The **Add Class Dependencies** add-in supports long model element names. Figure 2 shows the second panel for the **Add Class Dependencies** add-in. If you select a dependency from the **Dependencies found** or **New dependencies to be created** list boxes, the upper box displays the name of the provider, and the next box displays the name of the supplier.

Figure 2 Add Class Dependencies Add-in - Second Dialog Box



Move the cursor left to right to see any part of the name. The format for the name is:

<*unqualified_name*> - <*location_in_model*>

For example, the model element:

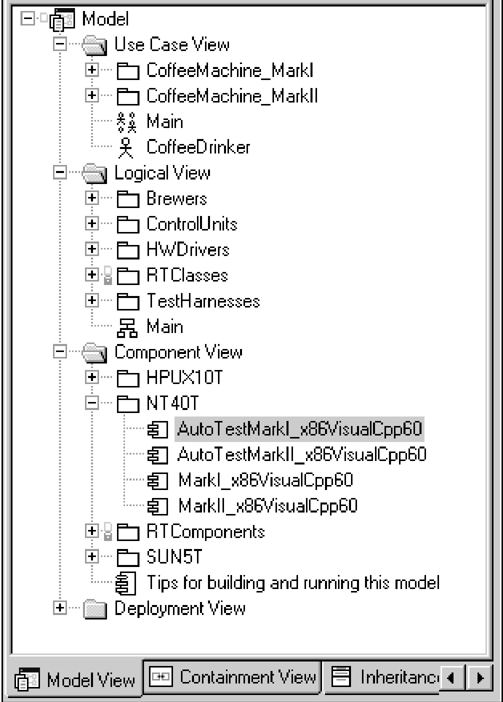
`LogicalView::PkgA::PkgB::myClass`

displays as

`myClass – PkgA::PkgB`

- You can search by component by selecting the desired component in the model, and then click **Build > Add Class Dependencies**. You can select more than one component to search multiple components at the same time.

Figure 3 Model View Tab in Browser



The **Add Class Dependencies** add-in examines all of the classes and capsules built by the component, including those:

- Directly referenced by the component by their name
- Contained in a package
- Referenced by aggregated components

- When the **Add Class Dependencies** add-in creates a new dependency in the model, it updates the **Log** tab in the **Output** window. You can double-click on a log entry to view the **Specification** dialog for the new dependency. The text of the log entry is prefixed by "`Warning`".
- When searching by component, the **Add Class Dependencies** add-in searches the classes and capsules assigned to the selected component for missing dependencies.

- The **Add Class Dependencies** add-in informs you of all of the changes made in the model after execution is completed:
  - The **Add Class Dependencies** add-in generates various log entries (for example, for missing dependencies with an ambiguous supplier, and for new dependencies added to the model). Additionally, the **Add Class Dependencies** add-in updates the **Log** tab in the **Output** window with entries to show:
    - the start of the operation ("-- `Starting Add Class Dependencies Operation` --")
    - the completion of the operation (for example, "-- `Add Class Dependencies Operation Complete --- Missing Dependencies Found:  3 ( 2 ambiguous). Dependencies added:  1.`")

    These types of messages created by the **Add Class Dependencies** add-in makes it easier for you to quickly separate log output generated by other activities. When the **Add Class Dependencies** add-in activities are complete, the entry in the **Log** tab for the completion phase is a summary report on the number of missing dependencies found (including those with an ambiguous supplier) and the number of dependencies added to the model.
  - At the end of the operation, the **Add Class Dependencies** add-in informs you that the operation is completed and refers you to the log for more information.

Considerations

The **Add Class Dependencies** add-in has the following limitations:

- The detail code in transitions or operations is not searched for possible missing dependencies. You must find and add those manually.
- Source code is not examined.
- Language properties are not examined (for example, **C++::ImplementationType** and **C::ImplementationType**).
- Dependencies are created with the default language property settings. In some cases, you may have to change these settings for individual dependencies.

Searching for Missing Dependencies

When configuring a search, you want to determine which classes and capsules to examine for missing dependencies. Because missing dependencies are a common source of compilation errors, consider examining the entire model for completeness. Alternatively, if you work on a very large model or a portion of a model, you may be more interested in examining only the classes and capsules you intend to build.

To examine a component:

1. Select a package containing a component to activate.

   The names appear indented to reflect the component hierarchy.

2. From the list of components, select a component.

   The **Add Class Dependencies** add-in examines all of the classes and capsules built by the selected component, including those directly referenced by the component by their name, or a containing package, as well as those referenced by aggregated components.

Specifying the Level of Dependency Checking

You can configure the **Add Class Dependencies** add-in to examine the attributes of the class and capsules. The search determines if the class or capsule currently has a dependency on the attribute's type (class). If there is no dependency, you are prompted to create one; however, it is optional.

You can configure the **Add Class Dependencies** add-in to examine the operations of the class and capsules. The search determines if the class or capsule currently has a dependency on the attribute's return type (class), and parameter types (class). If there is no dependency, you are prompted to create one; however, it is optional.

Note: The detail code in transitions or operations is not searched for possible missing dependencies. You must find and add those manually.

Adding Missing Dependencies

After the **Add Class Dependencies** add-in completes its search for missing dependencies, you are presented with a list of dependencies to add to the model. The class or capsule at the arrow base (Figure 2 on page 16) refers to the class pointed to. For example:

```
Deck --> Card
```

The class **Deck** has a dependency with class **Card**.

When looking at the list, you may decide to modify the capsule or class to remove the dependency. Therefore, it is not necessary to add that particular dependency.

Note: You want to keep the number of dependencies in your model to a minimum. Large numbers of dependencies increase the amount of time it takes to compile and build your model.

Specifying Dependencies to Create

Figure 2 on page 16 shows two lists: **Dependencies found** and **New dependencies to be created**. Only those dependencies that appear in the **New dependencies to be created** list are added to the model.

All

Moves all the dependencies back and forth between the lists.

Group

Moves all of a particular class or capsule's missing dependencies back and forth between the lists. Select one of the class or capsule's missing dependencies and click **Group**.

Single

Moves a single dependency back and forth between the lists. Select a dependency and click **Single**.

After you are satisfied with your choices, click **OK** to create the dependencies in the model.

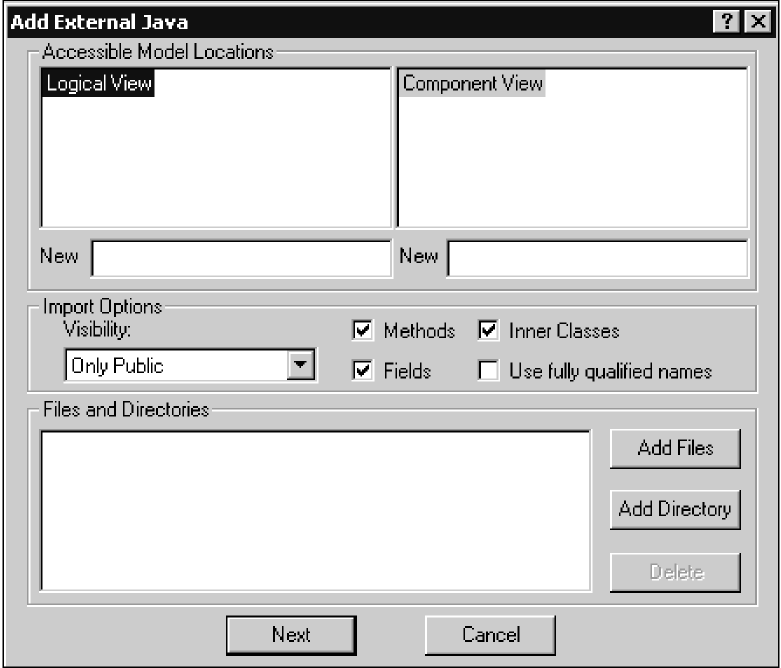
Add External Java Tool

The **Add External Java Tool** allows you to add external .class files and the .class files within .jar files into your existing model. To add external java .class files to a model, from the **Tools** menu, click **Add External Java** (see Figure 4 to view the **Add External Java** dialog).

Add External Java Dialog

Figure 4 shows the **Add External Java** dialog.

Figure 4 Add External Java Dialog



Accessible Model Locations

LogicalView

LogicalView displays the current structure of the various packages and classes that make up your model design. Expand the hierarchy to select the desired location to add classes.

To add a new package, see *New (Logical View)* on page 22.

ComponentView

ComponentView shows the current list of components in your model. Select a component to assign the classes.

To add a new component, see *New (Component View)* on page 22.

New (Logical View)

Adds a package to the model in the **Logical View**. Select an item from the **LogicalView** hierarchy to specify a location, then type a valid name in the **New** box. A valid name includes only alphanumeric characters and is unique.

New (Component View)

Adds a component to the model in the **Component View**. Select a package from the **ComponentView** hierarchy to specify a location, and then type a valid name in the **New** box. A valid name includes only alphanumeric characters and must be unique.

Import Options

Visibility

Specifies the type of visibility for the imported classes, inner classes, attributes, and operations. Specifying the visibility determines what is added. For example, if a class is public and you set the visibility to **Only Public**, only those attributes, operations, and inner classes within that class that are public are added into the package.

There are four types of visibility:

- **Only Public** - Only those classes and their inner classes, operations (method) and attributes (fields) that are public are added into the package; all others (default, protected, and private) are excluded.
- **Protected & Higher** - Only those classes and their inner classes, operations (method) and attributes (fields) that are public and protected are added into the package; all others (default and private) are excluded.
- **Default & Higher** - Only those classes and their inner classes, operations (method) and attributes (fields) that are public, default, and protected are added to the package. All others (private) are excluded.
- **All** - Classes and their inner classes, operations (method) and attributes (fields) that are public, default, protected, and private are added to the package. There is no filter and all items are added.

Methods

Specifies the level of detail to use. When selected, this item includes all of the methods for the classes being added into this model.

Fields

Specifies the level of detail to use. When selected, **fields** are included when adding classes.

Inner Classes

Specifies the level of detail to use. When selected, inner classes are included when you add items. If you use inner classes, the **Add External Java Tool** ensures that it includes all required .class files.

Note: The use of inner classes may significantly increase the total number of classes. Anonymous inner classes are not imported.

Use fully qualified names

Specifies that the fully qualified name for field names, parameters of methods, and exceptions that a method can throw, are displayed in the **LogicalView**.

Files and Directories

Shows the list of selected .class files, .jar files, and directories to add to this model. Click **Add Files** to add .class files, or click **Add Directory** to add all the .class files and the .class files within .jar files, in a directory. Click **Delete** to remove a .class file, a .jar, file, or directory from this list.

Add Files

Specifies the external .class files and .jar files to add to this model. Click **Add Files** to select all .class files and the class files within a .jar file in a directory.

Add Directory

Specifies a directory to add any .class files it contains.

Delete

Removes the selected file or directory from the **Files and Directories** list.

Next

Verifies the information on this dialog and displays the **Confirm Settings** dialog.

Note: If the toolset is unable to read your **TEMP** environment variable, it will prompt you to specify a temporary directory to uncompress any .jar files. If you selected a .jar file and you do not have a JAR Utility specified in your path (such as jar.exe), after you click **Next**, the .jar file(s) will not be uncompressed.

Cancel

Closes this dialog without saving any changes.

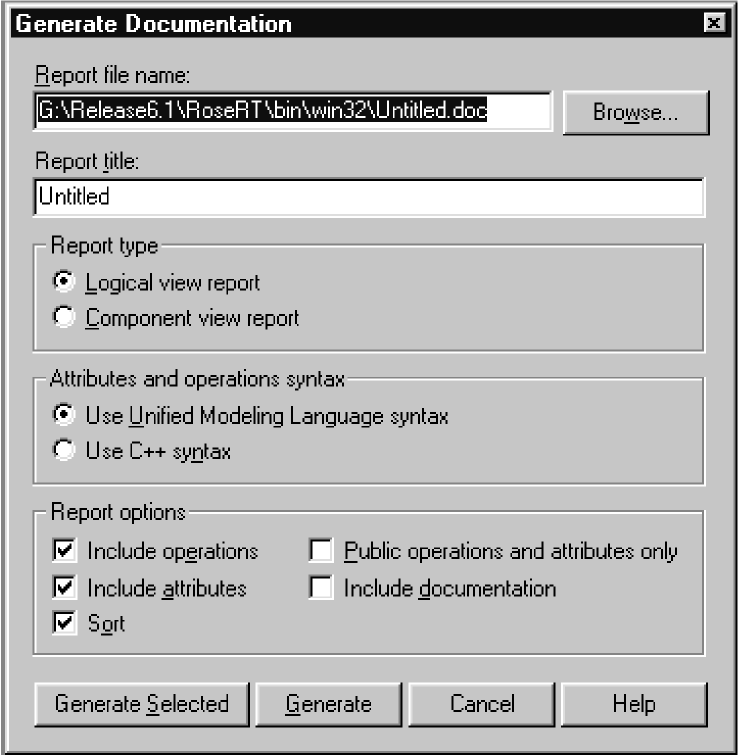
Specifying a JAR Utility in Your Path

If you selected a .jar file to add and your path does not include a JAR utility (such as the jar.exe utility that comes with JDK1.3), then you need to configure your path.

Generate Documentation Add-In

The **Generate Documentation** add-in generates a data dictionary from a model using Microsoft Word OLE Automation objects. To start the **Generate Documentation** add-in, click **Report > Documentation Report**, and the **Generate Documentation** dialog appears (Figure 5).

Figure 5 Generate Documentation Dialog



Report File Name

Shows the name of the Word document created by Microsoft Word.

Report Title

Shows the title text that appears on the cover sheet of the report.

Report Type

Specifies the view type to use to create a report:

- **Logical view report** - The resulting report lists the packages and each class contained within that package. Any nested packages use the same format. The information displayed about a given class depends on the **Report Options** settings.
- **Component view report** - The resulting report lists the packages and each component contained within that package. Each component contains information about the class assigned to the component. The information displayed about a given class depends on the **Report Options** settings.

Attributes and Operations Syntax

Select the **Use Unified Modeling Language Syntax** option to format attributes and operations using the UML syntax. For example:

```
myAttribute:AttributeType=initval
myOperation ():ReturnValue
```

Select the **Use C++ Syntax** option to format attributes and operations using the C++ syntax. For example:

```
AttributeType myAttribute = initval
ReturnValue myOperation ()
```

Report Options

- **Include operations** - Adds operations to each class in the report.
- **Include attributes** - Adds attributes to each class in the report.
- **Public operations and attributes only** - Only reports on public operations and attributes. This option depends on the settings of the **Include operations** and **Include attributes** options. For example, if **Include operations** and **Public operations and attributes only options** are selected, and the **Include attributes** option is not selected, only public operations appear in the report.

- **Include documentation** - Includes documentation for the associated elements being reported. For the logical report, this includes the package, class, operation, and attribute documentation. For the physical report, this includes the package, component, class, operation, and attribute documentation.
- **Sort** - Modifies the order of the operations and attributes so that they appear in alphabetical order in the report.

Source Code Assists and Make Files Writable Add-in

The **Source Code Assists** and **Make Files Writable** add-ins allow you to automate common source control tasks. The BasicScript source files for the add-ins can be found in the following directory:

$ROSERT_HOME/Scripts

Before modifying a script, ensure that you make a backup copy of the original. The source control add-ins add the following menu items to the toolset:

- *Select Checked Out Units in Browser* on page 26
- *Show Unit Versions* on page 27
- *Submit All Changes to Source Control* on page 27
- *Make Files Writable* on page 27
- *Make Files ReadOnly* on page 28.

Note: All of these options are available from the **Tools > Source Control** menu. By default, the **Make Files Writable** add-in is not enabled. To enable the add-in, select **Add-ins > Add-in Manager...** and select **Make Files Writable**.

Select Checked Out Units in Browser

Script File

$ROSERT_HOME\scripts\srcassists.ebs

Purpose

Selects all the current units checked out of source control. If a user made changes that involved a number of different units in different packages, this script simplifies the selection of all units for check-in.

Show Unit Versions

Script File

$ROSERT_HOME\scripts\srcassists.ebs

Purpose

Displays a dialog showing the source control version of each element in the model. It is possible for a user to obtain old versions of elements from source control. Some users may have versions of elements that are not the most recent version. You can use this option to identify the versions of the elements that they use.

Submit All Changes to Source Control

Script File

$ROSERT_HOME\scripts\srcassists.ebs

Purpose

Automatically checks in all elements currently checked out, and it also checks in any elements not yet added to source control instead of selecting each model element from the browser to perform a check-in operation.

Make Files Writable

Script File

$ROSERT_HOME\scripts\makewriteable.ebs

Purpose

Ensures that files for all modified units that are not currently checked out of source control are writable. It is possible to make changes to files without checking them out of source control. You may want to make changes to your local copy without checking out the files. Use this add-in to make these files writable so that you can save them.

Make Files ReadOnly

Script File

$ROSERT_HOME\Scripts\makewriteable.ebs

Purpose

Changes all writable files that are not currently checked out of source control to read-only. If you used **Make Files writable** to perform local changes, you can use this option to make the files read-only again when you are finished. Later, if you want these changes in source control, you can perform a check out without obtaining a local copy, and then check in these changes.

Rational Rose RealTime Tools 2

Contents

This chapter is organized as follows:

- *Aggregation Tool* on page 29
- *Attribute Tool* on page 42
- *Operation Tool* on page 49
- *Trace Tool* on page 60

Note: To activate or deactivate these tools, click **Add-Ins > Add-In Manager** .

Aggregation Tool

The **Aggregation Tool** lets you quickly create aggregate and composite associations. Aggregations are used to model a containment relationship between model elements. The generated code contain one or more attributes with the appropriate inclusions. The attributes created, and their types, depend on the details of the relationship.

You can use the **Aggregation Tool** to create a new aggregation or to modify an existing one. To access the **Aggregation Tool**, do one of the following:

- Select a class on a class diagram, and then click **Tools > Aggregation Tool.**
- Select an association in a class diagram, then right-click and select **1 Aggregation Tool.**
- Right-click on a class and select **1 Language Details > Aggregation Tool.**
- Select two classes from the **Model View** tab in the browser, then right-click and select **1 Language Details > Aggregation Tool.**

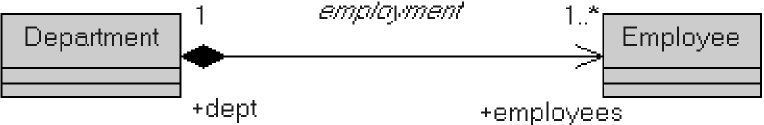
Note: If you use the **Aggregation Tool** to create or modify an aggregation, the model diagrams are not automatically updated. To update your model diagrams, you must use **Query > Filter Relationships**.

Relationships

Relationships establish a formal linkage between model elements. Associations are stronger forms of relationships that capture aggregation relations. An aggregation association is a special form of association that specifies the whole-part relationship

between an aggregate (whole) and the component (part). There are many examples of aggregation relationships, such as within a department there are employees (Figure 6), and a computer is composed of a number of devices.

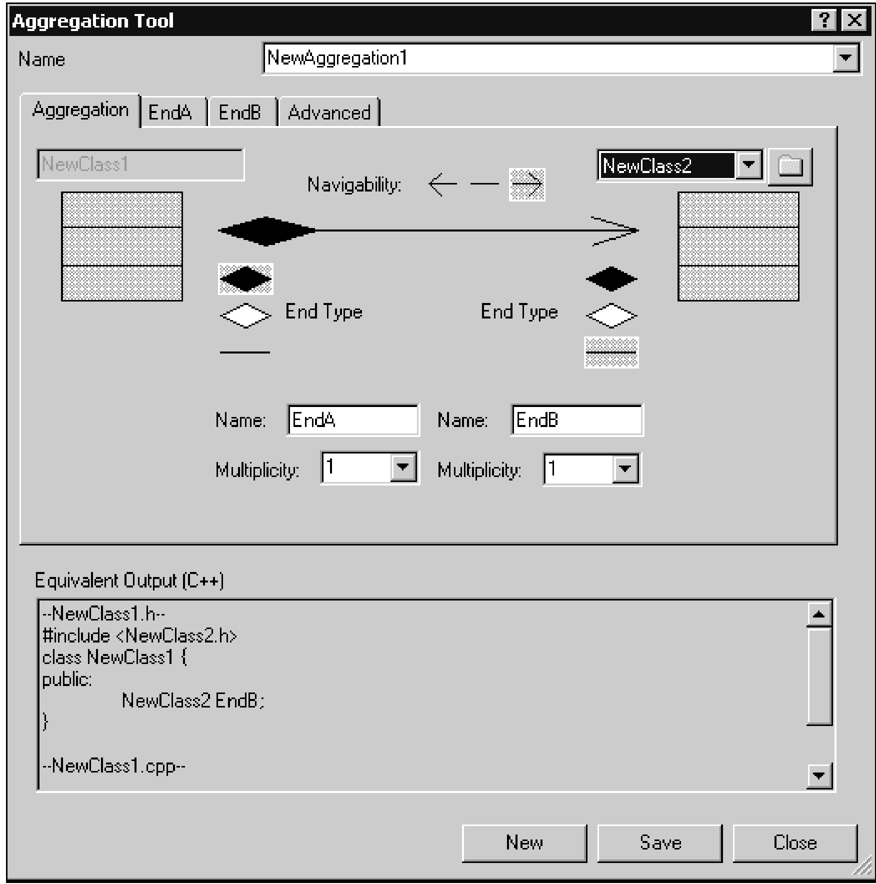
Figure 6 Aggregation Association



Aggregation Tab

Figure 7 shows the **Aggregation** tab for C++. The **Aggregation** tab for C, Java, and an Empty framework are similar to Figure 7.

Figure 7 Aggregation Tool - Aggregation Tab



Name

Specifies the name of the association. This name should describe the nature of the relationship between the two classes.

Class Name

Specifies the name of a class included in the association.

Navigability

Specifies the direction that affects the generated code which traverses to that end. It is the direction that the association is traversed. By default, the navigability is in one direction →.

The navigation options are:

- → Navigation is limited to one direction, from Class A to B.
- — Navigation is bidirectional.
- ← Navigation is limited to one direction, from Class B to A.

End Type

Specifies the type for the association end.

The end types are:

- ◇——— - Represents an aggregation. Aggregation means that the member is generated as a pointer to the other end class. It is the physical containment of a pointer or reference to the part.
- ◆——— - Represents a composition. Composition means that the member is generated as an embedded object. When the containing class is destroyed, the composite is also destroyed. It is the physical containment of a value to the part.
- ——— - Represents an association end. This type of physical containment is not specified. This is typical for the side of the association being contained in the other class.

Note: When specifying an **End Type**, only one end of the relationship can be aggregate.

Name

Specifies the name for the association end. The end of each association is called an association end or an end. You can label ends with an identifier that describes the role that an associate class plays in the association. This name should describe the nature of the end for the specific class.

End names are used by the code generator as a potential name for an attribute on a class.

Note: The code generation does not generate attributes for ends that are not named.

Multiplicity

Specifies the number of instances that can exist for this end of the association at any given time. You can either select a multiplicity from the list or specify your own by typing directly into this box.

When you specify a multiplicity at one end of an association (the near end), for each object of the class at the opposite end (the far end), there may be the same number of objects at the other end (the near end).

To view an approximation of the output for the specified **Visibility**, see the **Equivalent Output** box in the **Aggregation Tool** dialog box.

Equivalent Output

Displays a "best" approximation of the expected output for the options selected in the **Aggregation Tool** dialog box.

For a new aggregation, the **Equivalent Output** boxes display the following:

```
## Insufficient Information For Output ##
```

This means that there are insufficient options specified for the selected or new aggregation.

To view all of the **Equivalent Output** code in a single pane for the selected aggregation, use the grip (lower right corner of the dialog box) to resize the window.

Note: You can copy code from the **Equivalent Output** box; however, the code within this box is only an approximation and may not represent the precise code segment. Therefore, use caution when copying from the **Equivalent Output** box.

New

Creates a new aggregation.

Note: To save any changes to the previous aggregation, ensure that you click **Save** before you click **New**.

Save

Saves the current settings for the selected aggregation.

Note: If you use the **Aggregation Tool** to create an aggregation, the model diagrams are not automatically updated. To update your model diagrams, you must click **Query > Filter Relationships**.

Close

Closes the **Aggregation Tool** dialog box.

EndA and EndB

An association is a relationship among two or more elements. The ends of each association are called association ends. You can label the ends with an identifier that describes the role that an associate element plays in the association. An end has both generic and language-specific properties that affect the generated code that traverses to that end.

Note: If you have the default names, EndA and EndB, the names of the **EndA** and **EndB** tabs also change**.**

To view the dialog box containing the **EndA and EndB** tabs for the **Aggregation Tool**, select one of the following:

- *Aggregation Tool for C++* on page 35
- *Aggregation Tool for C* on page 37
- *Aggregation Tool for Java* on page 36
- *Aggregation Tool for an Empty Framework* on page 38

C++ Class to Class (Data Member) Associations

By default, the code generator generates a data member (attribute) for navigable and named ends of associations. Several factors affect the code that is actually generated:

- The **AssociationEndKind** (Role, C++) property determines whether a member or global data member is generated.
- The cardinality affects whether an array of attributes should be created.
- The containment affects whether the attribute should be a reference (a pointer) or an object.

C Class to Class (Data Member) Associations

By default, the code generator generates a data member (attribute) for navigable and named ends of associations. Several factors affect the code that is actually generated:

- The scope property determines whether a member or global data member is generated.
- The multiplicity affects whether an array of attributes should be created. If the multiplicity for an association is specified as x..z, only the upper bound is used.
- The containment affects whether the attribute should be a reference (pointer) or an object.

Java Class to Class

By default, the code generator generates a data member (attribute) for navigable and named ends of associations. The multiplicity affects the code that is actually generated. The multiplicity is used to specify the size of the array to create.

Aggregation Tool - EndA and EndB Tabs

To view the dialog box containing the **EndA and EndB** tabs for the **Aggregation Tool**, select one of the following:

- *Aggregation Tool for C++* on page 35
- *Aggregation Tool for C* on page 37
- *Aggregation Tool for Java* on page 36
- *Aggregation Tool for an Empty Framework* on page 38

Figure 8 Aggregation Tool for C++

Figure 9 Aggregation Tool for Java

Figure 10 Aggregation Tool for C

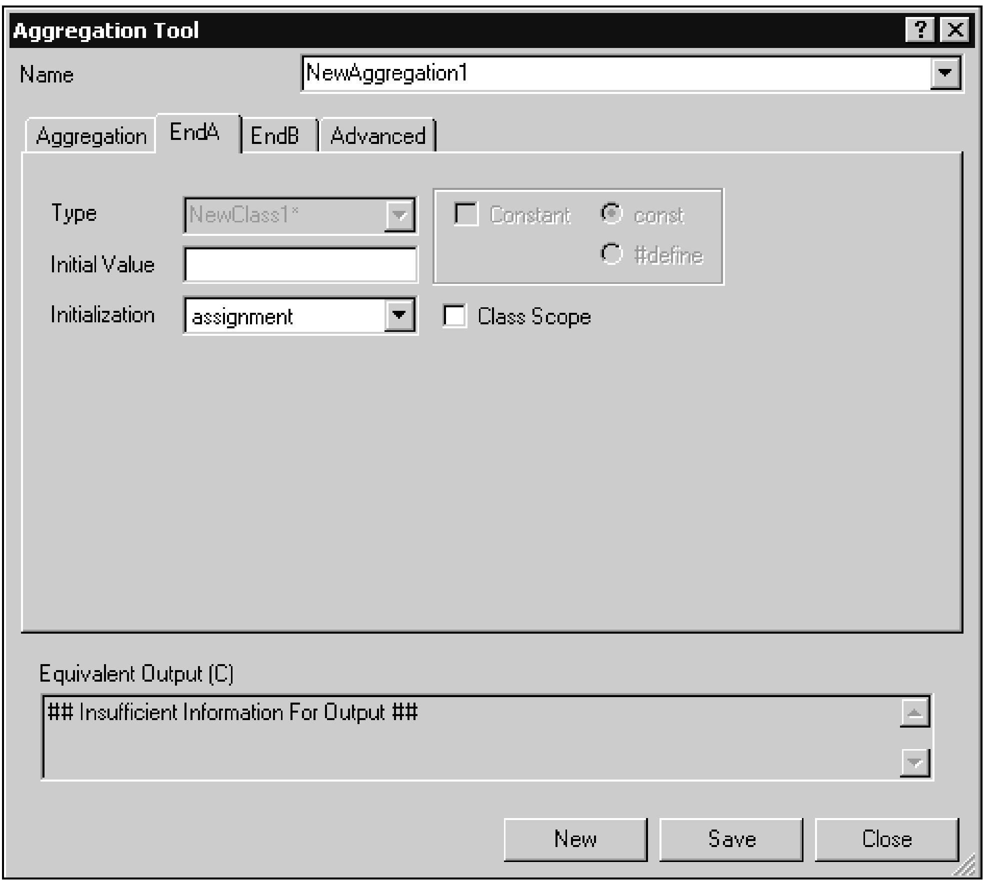


Figure 11 Aggregation Tool for an Empty Framework

Descriptions

Type

Not available in the **Aggregation Tool**.

Initial Value

Assigns an initial value to your class attribute. Type directly in the **Initial Value** box. Rational Rose RealTime displays the initial value opposite the attribute name and updates the information in the model.

Observe the results in the **Equivalent Output** box to view an approximation of the output.

Initialization

Specifies that if an initial value was assigned to this relationship, there may be a choice regarding the method used to initialize the relationship.

There are two types of initialization: **Assignment** and **Constructor**. Selecting **Assignment** assigns the value directly. Selecting **Constructor** means that a call is made to a constructor to initialize the attribute.

Visibility

Specifies the type of visibility for each end of an aggregation. There are four types of visibility:

- **Public** - The objects at both ends are visible to each other.
- **Protected** - The objects at this end are not accessible to any object outside the association, except for the children of the other end.
- **Private** - The objects at this end are not accessible to any object outside the association.
- **Implementation** - The objects at either end are never visible to other classes.

See the **Equivalent Output** box in the **Aggregation Tool** dialog box to view an approximation of the output for the specified **Visibility**.

Class Scope

Specifies the target scope; that is, whether there is only a single instance of the feature for all instances of the classifier. When selected, the data member is scoped to the classifier.

Global (C++)

Specifies that the relationship is global. It is accessible beyond the boundaries of the current class.

Volatile (Java)

Specifies that the attribute is modified asynchronously by concurrently running threads.

Transient (Java)

Indicates that the values of its transient fields are not included in the serial representation while the values of its non-transient fields are included. This means that the resource is released when it is not being used.

Constant (C, C++, Java, Empty)

Not available in the **Aggregation Tool**.

const (C, C++)

Not available in the **Aggregation Tool**.

#define (C, C++)

Not available in the **Aggregation Tool**.

Generate:

get Method (C++, Java, Empty)

Not available in the **Aggregation Tool**.

set method (C++, Java, Empty)

Not available in the **Aggregation Tool**.

Initialization Code (Java)

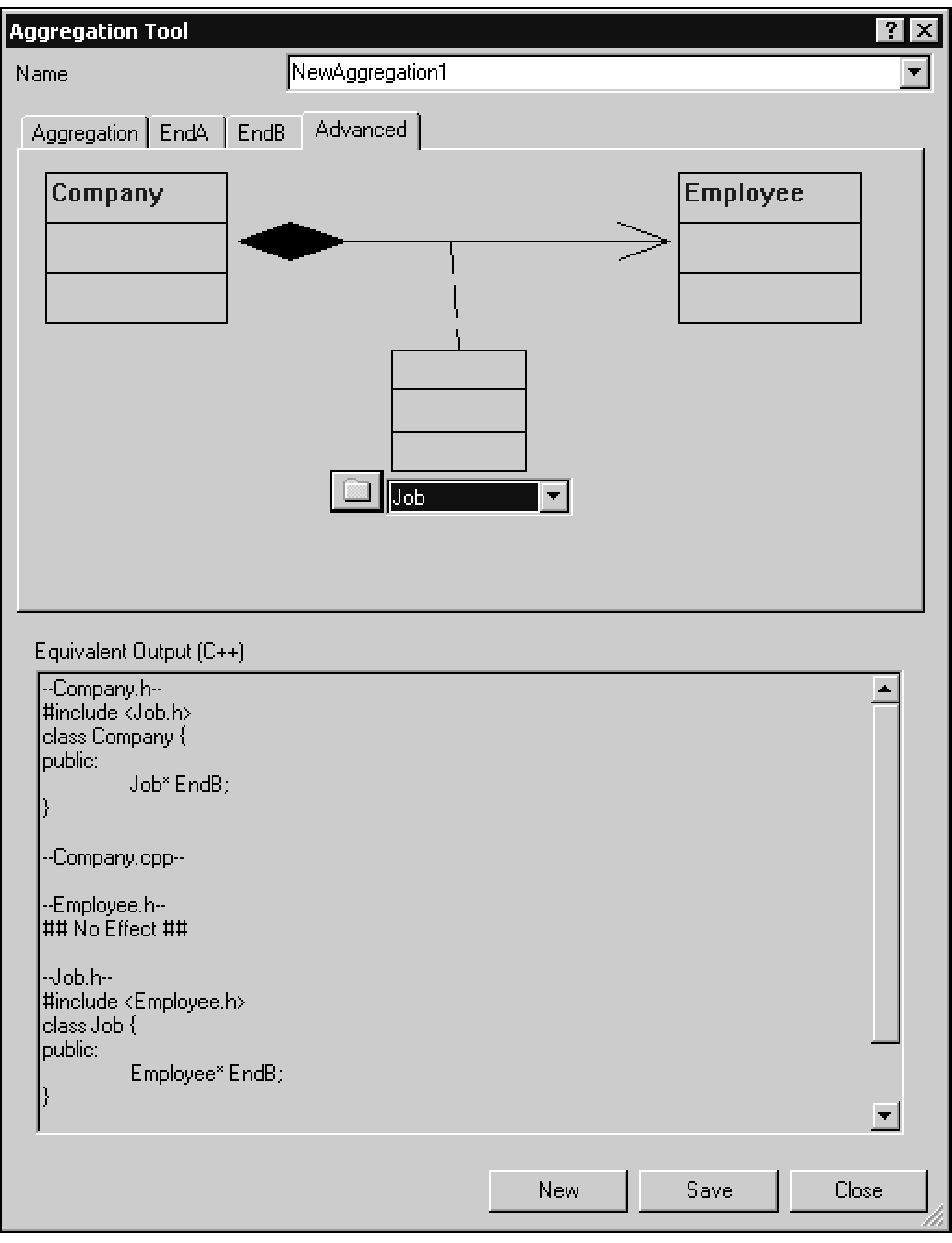
Specifies any code used to initialize this End.

Advanced Tab

An association class specifies the name of a class that defines the relationship between the two classes. Use the association class to model properties of associations. The properties are stored in the class and linked to the association relationship. Linked attributes are degenerate association classes comprised only of attributes.

When an association exists between two classes, that association itself can have properties; called an association class. An association class is an association that has class properties. For example, a company has employees, and the employees have jobs. A specific instance of a job could effectively be linked to the relationship of company to employee.

Figure 12 Aggregation Tool - Advanced Tab



Note: You cannot attach an association class to more than one association; the association class is the association itself. However, you can define a class, such as ClassZ, then have each association class that requires those features inherit from ClassZ, or have them use ClassZ as the type for an attribute.

Attribute Tool

The **Attribute Tool** enables you to quickly create and set options for an attribute. An attribute is a named property of a class that defines the values that instances of the property can hold.

Use the **Attribute Tool** to create a new attribute, or modify an existing attribute. The **Attribute Tool** is available from the C++, C, Java, and Empty frameworks. You can access the **Attribute Tool** from the shortcut menu for the following model elements:

- attribute
- class
- capsule
- classifier role
- capsule role
- interaction instance

To access the **Attribute Tool**, press the **Shortcut Menu** key (or press **ALT** + **ALT** to access the shortcut menu), and click **Attribute Tool**. When more than one tool is available, select **1 Language Details** to obtain a submenu listing the available tools: **Attribute Tool**, **Operation Tool**, and **Aggregation Tool**.

Note: If you use the **Attribute Tool** to add an attribute, the model diagrams are not automatically updated. The **Model View** tab in the browser updates immediately and displays the new attribute. To update your model diagrams, select the class or capsule from the **diagram**, right-click and click **Options > Show All Attributes**. If **Show All Attributes** is currently selected, de-select it, and then re-select it again.

Attribute Tool: Properties Tab

To view the dialog containing the **Properties** tab for the **Attribute Tool**, select one of the following framework types:

- C, see Figure 15
- C++, see Figure 16
- Java, see Figure 17
- Empty framework, see Figure 18

Descriptions

Name (C, C++, Java, Empty)

The name of an attribute (**Attribute Tool**) or operation (**Operation Tool**). Use the name box to:

- Specify a name for a new attribute or operation
- Select an existing attribute or operation from the list
- Change the name of an existing attribute or operation

For the **Operation Tool**, this item specifies the name as well as any parameters and functions for the operation. Each parameter must have a valid type and name; however, a value is optional. For example, the following declarations are valid:

- `myDec1(int foo)`
- `myDec2(int foo = 55, bool barA = False)`
- `myDec3(RTTime t)`
- `myDec3(RTTime& t)`

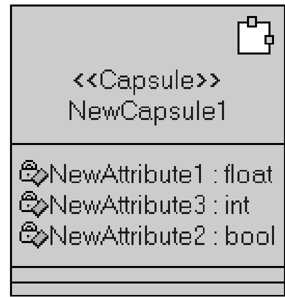
If you specify a variable type and variable name that is a class in a parameter, you must click the **Dependencies** tab and satisfy any required dependencies.

Type (C, C++, Java, Empty)

Attribute types can be classes or language-specific types. When the attribute is a data value, the type is defined as a language-specific type. You can enter the type directly in the **Type** box, or select a type from the list. Use the UP and DOWN arrow keys to scroll through the list. In the **Class** diagram, Rational Rose RealTime displays the type opposite the attribute name and updates the information in the model (see Figure 13 for an example).

Observe the results in the **Equivalent Output** box in the **Attribute Tool** dialog to view an approximation of the output for the attribute.

Figure 13 Display of attributes and corresponding types

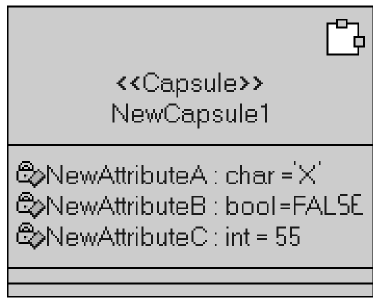


Initial Value (C, C++, Java, Empty)

Assigns an initial value to your class attribute. Type directly in the **Initial Value** box. Rational Rose RealTime displays the initial value opposite the attribute name and updates the information in the model (see Figure 14). If you select **Constant**, you must specify a value in the **Initial Value** box.

Observe the results in the **Equivalent Output** box in the **Attribute Tool** dialog to view an approximation of the output.

Figure 14 Display of Initial Values for attributes



Initialization (C, C++)

Specifies that if an initial value was assigned to this attribute, there may be a choice regarding the method used to initialize the attribute.

There are two types of initialization for an attribute: **Assignment** and **Constructor**. Selecting **Assignment** assigns the value directly to the attribute. Selecting **Constructor** means that a call is made to a constructor to initialize the attribute.

Visibility (C++, Java)

Specifies the type of visibility for an attribute. There are four types of visibility:

- **Public** - The attribute is visible to other classes.
- **Private** - The attribute is not visible to other classes (except designated *friend* classes in C++).
- **Protected** - The attribute is visible only to subclasses (and *friend* classes in C++).
- **Implementation** - The attribute is never visible to other classes.

Observe the results in the **Equivalent Output** box in the **Attribute Tool** dialog to view an approximation of the output for the specified **Visibility**.

Class Scope (C, C++, Java, Empty)

Specifies the class scope for the attribute. Selecting this option means that there is a single instance of the attribute for all instances of the class (for example, a static member in C++). If this option is not selected, then each instance of the class has a separate attribute instance.

Observe the results in the **Equivalent Output** box in the **Attribute Tool** dialog to view an approximation of the output when selecting **Class Scope**.

Global (C++)

Specifies that the attribute is global. This means that the attribute is accessible beyond the boundaries of the current class.

Constant (C, C++, Java, Empty)

Specifies whether the attribute is a constant. This means that this attribute cannot take on a new value. If you select **Constant**, you must specify a value in the **Initial Value** box.

See the **Equivalent Output** box in the **Attribute Tool** dialog to view an approximation of the output when selecting **Constant**.

const (C, C++)

Specify the value cannot be changed. This means that the attribute of a declaration makes the entity to which it refers read-only.

#define (C, C++)

Provides an efficient way to create symbolic constants.

Volatile (Java)

Specifies that the attribute is modified asynchronously by concurrently running threads.

Transient (Java)

Indicates that the values of its transient fields are not included in the serial representation, while the values of its non-transient fields are included. This means that the resource is released when it is not being used.

Generate:

get Method (C++, Java, Empty)

Generates an operation to retrieve the value of this attribute.

Observe the results in the **Equivalent Output** box in the **Attribute Tool** dialog to view an approximation of the output when selecting **get method**.

- **const** (C++): Sets the return value to **const** which does not allow the value to be modified.
- **inline** (C++): Modifies an operation definition so that it is expanded into the body of the calling function. It notifies the compiler that this function should be inlined.

set method (C++, Java, Empty)

Generates an operation to assign a value to this attribute.

Observe the results in the **Equivalent Output** box in the **Attribute Tool** dialog to view an approximation of the output when selecting **set method**.

- **Inline** (C++): Modifies an operation definition so that it is expanded into the body of the calling function. It notifies the compiler that this function should be inlined.

Initialization Code (Java)

Specifies any code used to initialize the attribute.

Equivalent Output (C, C++, Java, Empty)

Displays a "best" approximation of the expected output for the options selected in the **Attribute Tool** dialog.

For a new attribute, the **Equivalent Output** box displays the following:

```
## Insufficient Information For Output ##
```

This means that there is insufficient information specified for the selected or new attribute.

To view all of the **Equivalent Output** code in a single pane for the selected attribute, resize the dialog box.

Note: You can copy code from the **Equivalent Output** box; however, the code within this box is only an approximation and may not represent the precise code segment. Therefore, when copying from the **Equivalent Output** box, use caution.

New

Adds a new attribute (**Attribute Tool**) or operation (**Operation Tool**).

Note: To save any changes to the existing attribute or operation, ensure that you click **Save** before you click **New**.

Save

Saves the settings for the selected attribute or operation.

Close

Closes this dialog without saving any changes you made.

Properties Tab: Language-Specific Options

Figure 15 Attribute Tool: Properties Tab for C

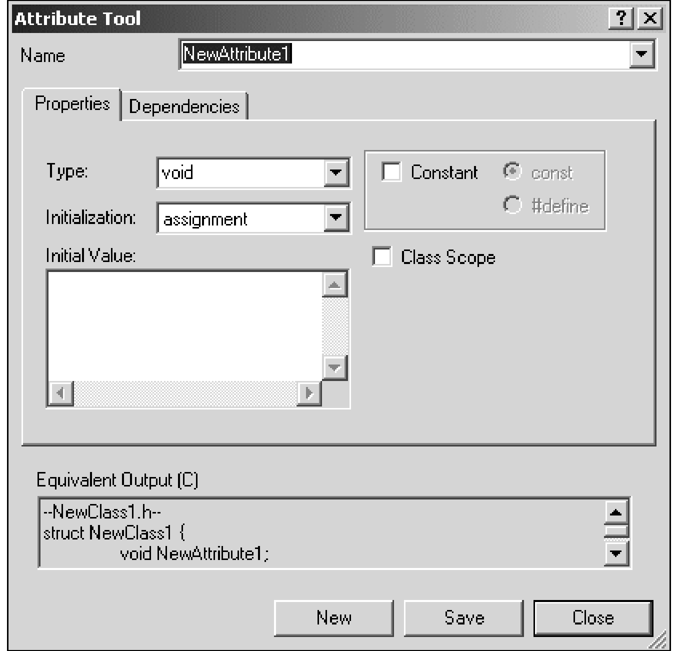


Figure 16 Attribute Tool: Properties Tab for C++

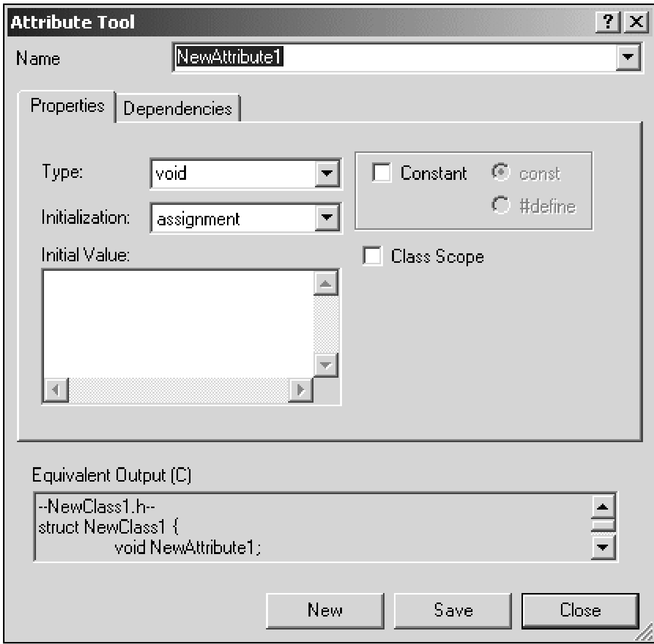


Figure 17 Attribute Tool: Properties Tab for Java

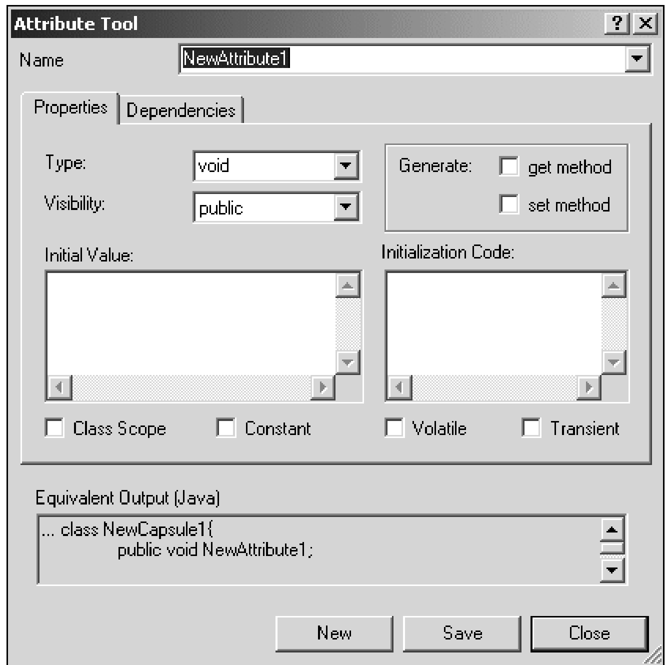


Figure 18 Attribute Tool: Properties Tab for an Empty Framework

Operation Tool

The **Operation Tool** lets you quickly create and set options for an operation. An operation is the implementation of a service requested from any object of the class that affects its behavior.

Use the **Operation Tool** to create a new operation, or modify an existing operation. You can access the **Operation Tool** from shortcut menu for the following model elements:

- operation
- class
- capsule
- classifier role
- capsule role
- interaction instance

To access the **Operation Tool**, press the **Shortcut Menu** key (or press **ALT** + **ALT** to access the shortcut menu), and click **Operation Tool**. When more than one tool is available, select **1 Language Details** to obtain a submenu listing the available tools: **Attribute Tool**, **Operation Tool**, and **Aggregation Tool**.

Note: If you use the **Operation Tool** to add an operation, the model diagrams are not automatically updated. The **Model View** tab in the browser updates immediately and displays the new operation. To update your model diagrams, select the class or capsule from the **diagram**, right-click and click **Options > Show All Operations**. If **Show All Operations** is currently selected, de-select it, and then re-select it again.

Note: The **Operation Tool** does not handle pointers to functions or templates; only simple parameters.

Operation Tool: Properties Tab

To view the dialog containing the **Properties** tab for the **Operation Tool**, select one of the following:

- C, see Figure 19
- C++, see Figure 20
- Java, see Figure 21
- Empty framework, see Figure 22

Descriptions

Name (C, C++, Java, Empty)

The name of an operation (**Operation Tool**). Use the name box to:

- Specify a name for a new operation
- Select an existing operation from the list
- Change the name of an existing operation

This item specifies the name as well as any parameters and functions for the operation. Each parameter must have a valid type and name; however, a value is optional. For example, the following declarations are valid:

- `myDec1(int foo)`
- `myDec2(int foo = 55, bool barA = False)`
- `myDec3(RTTime t)`
- `myDec3(RTTime& t)`

If you specify a variable type and variable name that is a class in a parameter, you must click the **Dependencies** tab and satisfy any required dependencies.

Return Type (C, C++, Java, Empty)

For operations that are functions, set this field to identify the class or type of the function's result. You can specify a class name that does not yet exist in your model; however, Clicking Save and closing the Operation Tool does not automatically create the class.

Observe the results in the **Equivalent Output** box in the **Operation Tool** dialog to view an approximation of the output when specifying a **Return Type**.

Query (C, C++, Java, Empty)

Specifies that the operation is read-only and does not modify the object's state.

Observe the results in the **Equivalent Output** box in the **Operation Tool** dialog to view an approximation of the output when selecting **Query**.

Class Scope (C, C++, Java, Empty)

Specifies class scope for the operation. Selecting this option means that the operation behaves the same way regardless of the state of any individual object in the class. Otherwise, the operation operates on individual class instances because its calculations are based on the object state, or because it modifies the object state.

Observe the results in the **Equivalent Output** box in the **Operation Tool** dialog to view an approximation of the output when selecting **Class Scope**.

Global (C++)

Specify that the operation is global. This means that the operation is accessible beyond the boundaries of the current class.

Friend (C++)

Specifies access to all the members of the class for which the attribute belongs. It does not matter which visibility level the attribute has because when the attribute is a **Friend**, the visibility is ignored.

final (Java)

Declares the operation a constant. This means that its initial value cannot be changed.

strictfp (Java)

Lets you have more predictable control over floating-point arithmetic. This item may be used as a modifier to classes, interfaces and methods.

native (Java)

Specifies that the operation implementation is in a language other than Java. The **native** keyword signals to the Java compiler that the function is a native language function.

Open specification dialog after close (C, C++, Java, Empty)

Informs the **Operation Tool** to open the **Specification** dialog of the current operation after closing the tool. A **Specification** dialog opens for each operation that you set this option. Opening the specification after closing this dialog provides convenient access to the **Code** box for adding code to this operation.

Note: When opening the **Operation Specification** dialog, the tab that you selected for the last **Operation Specification** dialog you opened is the tab that displays.

Visibility (C, C++, Java, Empty)

Specifies the type of visibility for an operation. There are four types of visibility for operations:

- **Public** - The operation is visible to other classes.
- **Private** - The operation is not visible to other classes (except designated *friend* classes in C++).
- **Protected** - The operation is visible only to subclasses (and *friend* classes in C++).
- **Implementation** - The operation is never visible to other classes.

Observe the results in the **Equivalent Output** box in the **Operation Tool** dialog to view an approximation of the output for the specified **Visibility**.

Abstract (C, C++, Java, Empty)

When selected, it indicates that the operation is an abstract definition that is required to be overridden by specific implementations in subclasses. Also, no instances of the current class are allowed.

For Java, if **Abstract** is selected, the method is declared abstract, and any code in the code setting is ignored.

Observe the results in the **Equivalent Output** box in the **Operation Tool** dialog to view an approximation of the output when selecting **Abstract**.

Polymorphic (C, C++, Java, Empty)

Indicates that the operation could be overridden by specific implementations in subclasses.

Observe the results in the **Equivalent Output** box in the **Operation Tool** dialog to view an approximation of the output when selecting **Polymorphic**.

Inline (C++)

Declares the operation **inline** to allow the compiler to optimize the generation.

Throws (Java)

Specifies an comma separated list of exception classes that the operation may throw. This list is placed in the throws clause of the method declaration.

Note: You must use a comma to separate the exception classes. You can either enter the exception classes on a single line, or you can press ENTER after each entry in this box. For example, these are both valid:

```
excClassA, excClassB, excClassC
```

or

```
excClassA,
excClassB,
excClassC
```

Equivalent Output

Displays a "best" approximation of the expected output for the options selected in the **Operation Tool** dialog.

For a new operation, the **Equivalent Output** box displays the following:

```
## Insufficient Information For Output ##
```

This means that there is insufficient information specified for the selected or new operation.

To view all of the **Equivalent Output** code in a single pane for the selected operation, resize the dialog box. The **Equivalent Output** box resizes to contain all of the code from the **Equivalent Output** box.

Note: You can copy code from the **Equivalent Output** box; however, the code within this box is only an approximation and may not represent the precise code segment. Therefore, when copying from the **Equivalent Output** box, use caution.

New

Adds a new operation.

Note: To save any changes to the existing operation, ensure that you click **Save** before you click **New**.

Save

Saves the settings for the selected operation.

Close

Closes this dialog without saving any changes you made.

Properties Tab: Language-Specific Options

Figure 19 Operation Tool: Properties tab for C

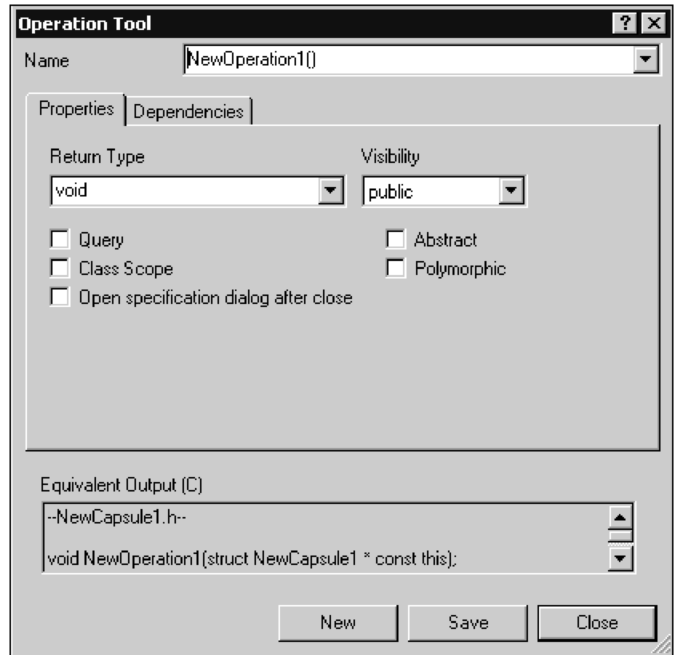


Figure 20 Operation Tool: Properties Tab for C++

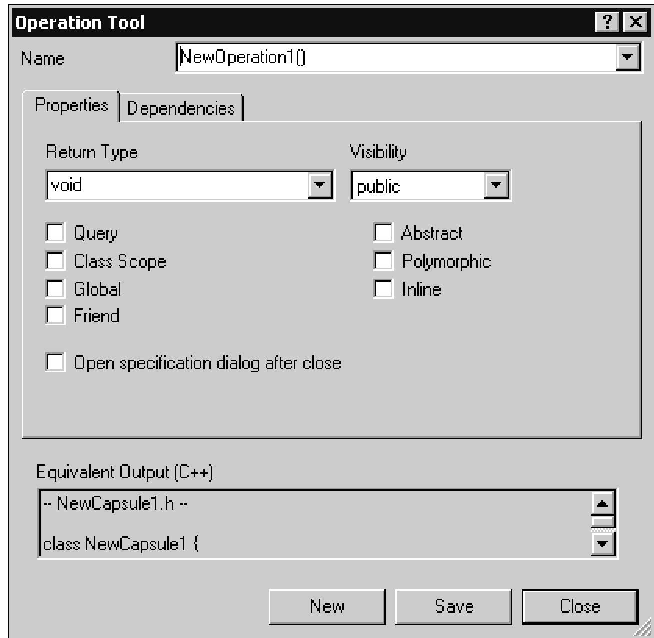


Figure 21 Operation Tool: Properties tab for Java

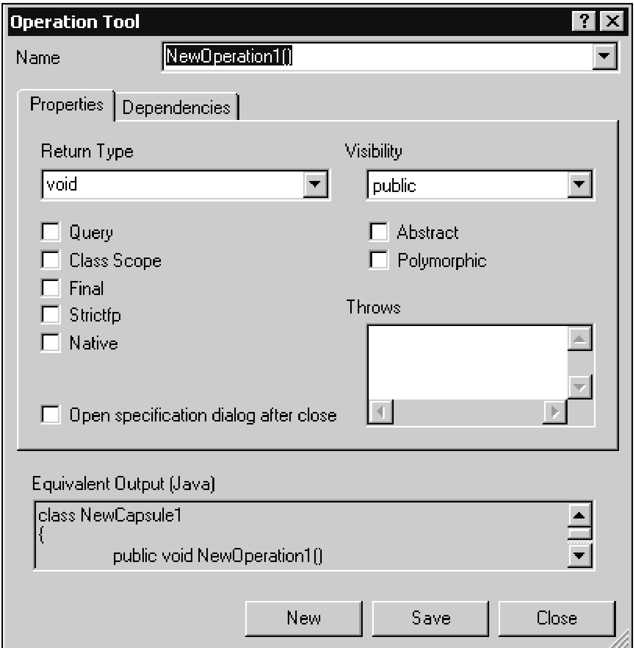
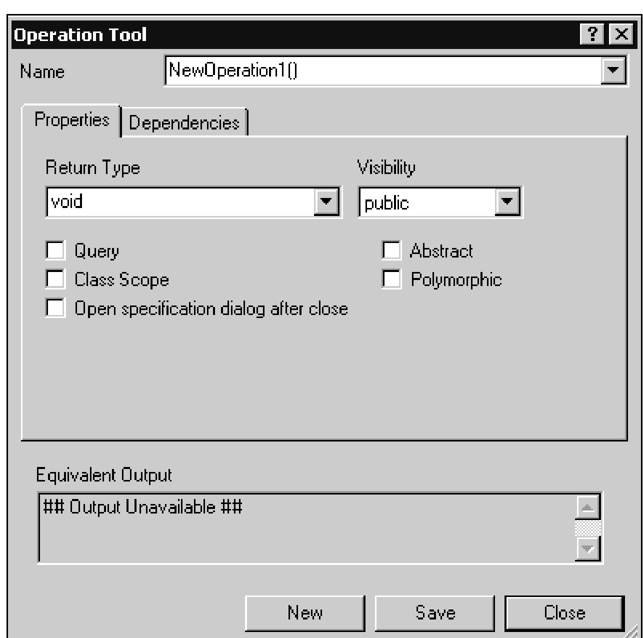


Figure 22 Operation Tool: Properties tab for an Empty Framework



Dependency Tab for Attribute Tool and Operation Tool

The **Dependencies** dialog lets you create any required dependencies for the following:

- attributes (using the **Attribute Tool**)
- operations (using the **Operation Tool**)

A dependency is a relationship that indicates that a change to one thing may affect another thing that uses it.

You can create and remove dependencies for an attribute or operation by selecting the **Dependencies** tab from the **Attribute Tool** dialog or the **Operation Tool** dialog.

To access the **Attribute Tool** or the **Operation Tool**, press the **Shortcut Menu** key (or press **ALT** + **ALT** to access the shortcut menu), and click the **Attribute Tool** or the **Operation Tool**. When more than one tool is available, select **1 Language Details** to obtain a submenu listing the available tools: **Attribute Tool**, **Operation Tool**, and **Aggregation Tool**.

To view the dialog containing the **Properties** tab for the **Operation Tool**, select one of the following:

- C, see Figure 23
- C++, see Figure 23
- Java, see Figure 24
- Empty framework, see Figure 24

Descriptions

Required Dependencies (C, C++, Java, Empty)

Specifies the name of a possible dependency for the selected operation or attribute. The drop-down list contains the names of the classes for which a dependency can be created. All satisfied dependencies are prefixed with **(satisfied)**.

Matching Classes (C, C++, Java, Empty)

Provides a list of classes that have the same name as the required dependency. If a dependency already exists, the name is prefixed with two asterisks (**).

Options: (C, C++)

Header (C, C++)

Specifies the directive that is generated in the header file. When the C++ generator produces code for an element (the client) that uses another element (the supplier), the C++ generator produces either an **include** directive referencing the file that contains the supplier class, or a **forward reference** to the supplier.

You can configure which directive (**include statement** or **forward reference**) is generated in the header file (.h).

Implementation (C, C++)

Specifies the directive that is generated in the implementation file. When the C++ generator produces code for an element (the client) that uses another element (the supplier), the C++ generator produces either an **include** directive referencing the file that contains the supplier class, or a **forward reference** to the supplier.

You can configure which directive (**include statement, forward reference**, or **none**) is generated in the implementation file (.cpp).

Create Dependency (C, C++, Java, Empty)

Generates the dependency selected in the **Required Dependencies** box. If this option is not selected, a dependency will not be generated for the selected class. Use this option to exclude the generation of dependencies. For example, select only those required dependencies for which you do not want a dependency, click this option for each required dependency so that it is set, and then click **Generate Chosen**.

Remove Dependency (C, C++, Java, Empty)

Removes an existing dependency for the selected class.

Equivalent Output (C, C++, Java, Empty)

Displays a "best" approximation of the expected output for the options selected in the **Attribute Tool** and **Operation Tool** dialog.

For a new operation, the **Equivalent Output** box displays the following:

```
## Insufficient Information For Output ##
```

This means that there is insufficient information specified for the selected or new attribute or operation.

To view all of the **Equivalent Output** code in a single pane for the selected attribute or operation, resize the dialog box. The **Equivalent Output** box resizes to contain all of the code from the **Equivalent Output** box.

Note: You can copy code from the **Equivalent Output** box; however, the code within this box is only an approximation and may not represent the precise code segment. Therefore, when copying from the **Equivalent Output** box, use caution.

Save

Saves the settings for the selected attribute or operation.

Close

Closes this dialog without saving any changes you made.

Dependencies Tab: Language-Specific Options

Figure 23 Dependencies Tab for C and C++

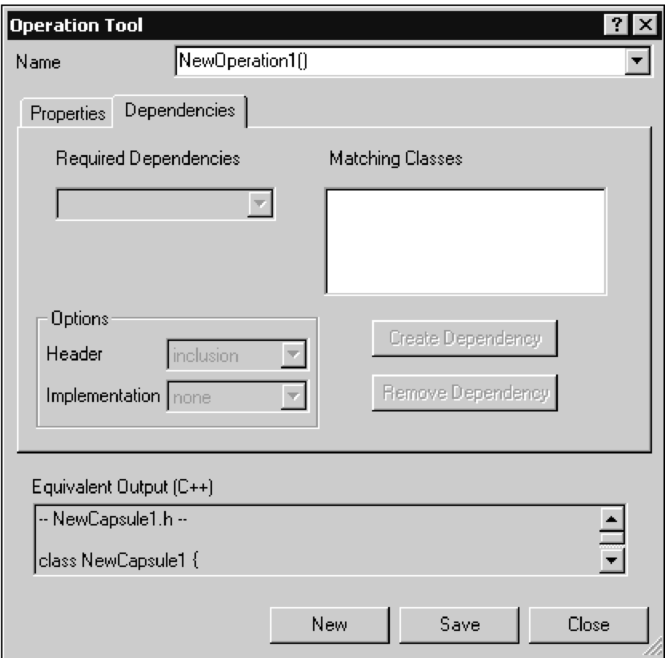
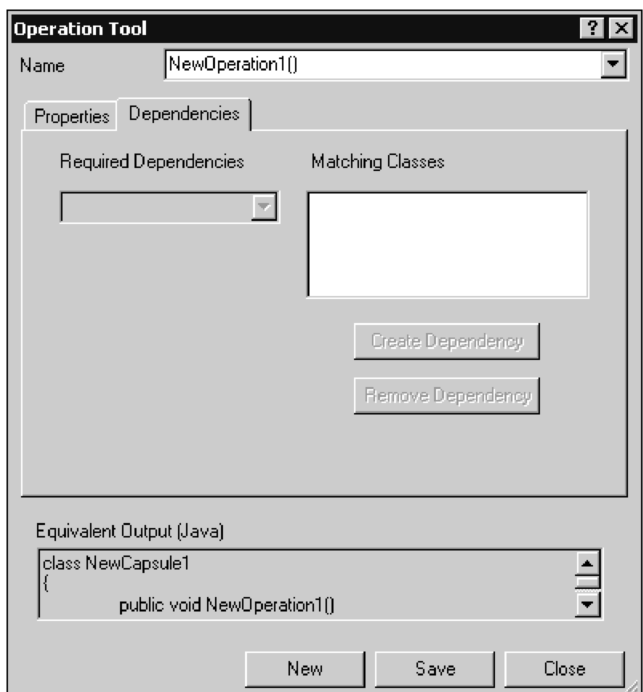


Figure 24   Dependency Tab for Java and an Empty Framework



Trace Tool

The Trace Tool provides you with the ability to establish and maintain traceability between DESIGN requirements (a special requirement type) in Rational RequisitePro and Rational Rose RealTime elements, such as classes, operations, and diagrams. The Trace Tool allows you to select an element and have the specification for the corresponding element in the other application appear. For example, in Rational Rose RealTime, you can open a model and select a capsule, then you can request the display of the corresponding requirement in Rational RequisitePro.

Note: The Trace Tool is only available for Windows configurations. To activate or deactivate this tool, click **Add-Ins > Add-In Manager**, and then select **Trace Tool**.

Configuring Rational RequisitePro for Traceability

To configure Rational RequisitePro and Rational Rose RealTime to use the Trace Tool, you must perform the following activities:

- *To modify the Rational RequisitePro project to include a requirement type called DESIGN with the appropriate attributes:* on page 61
- *To add customized menu commands to Rational RequisitePro:* on page 67
- *To associate a model to a project:* on page 68

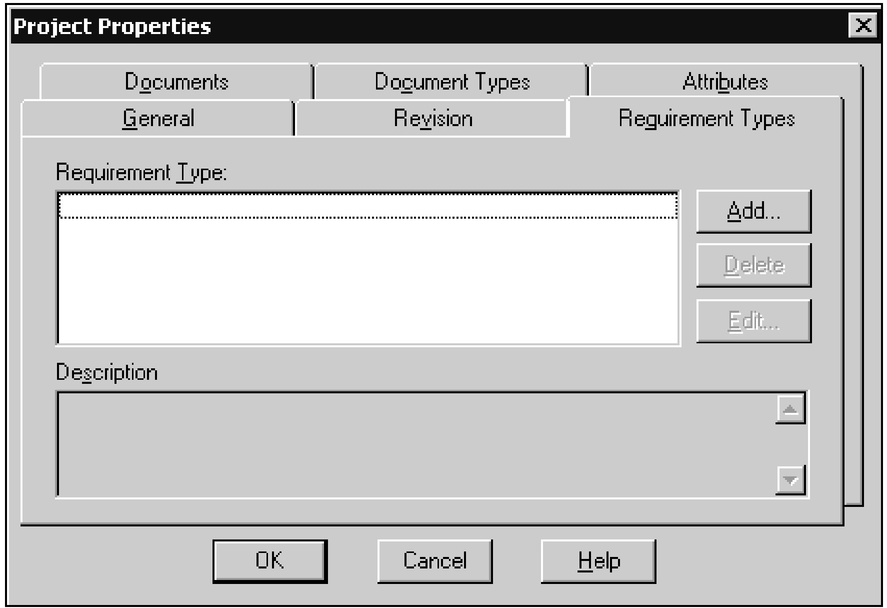
Note: A Rational RequisitePro project file must currently exist before you can attempt to configure.

To modify the Rational RequisitePro project to include a requirement type called DESIGN with the appropriate attributes:

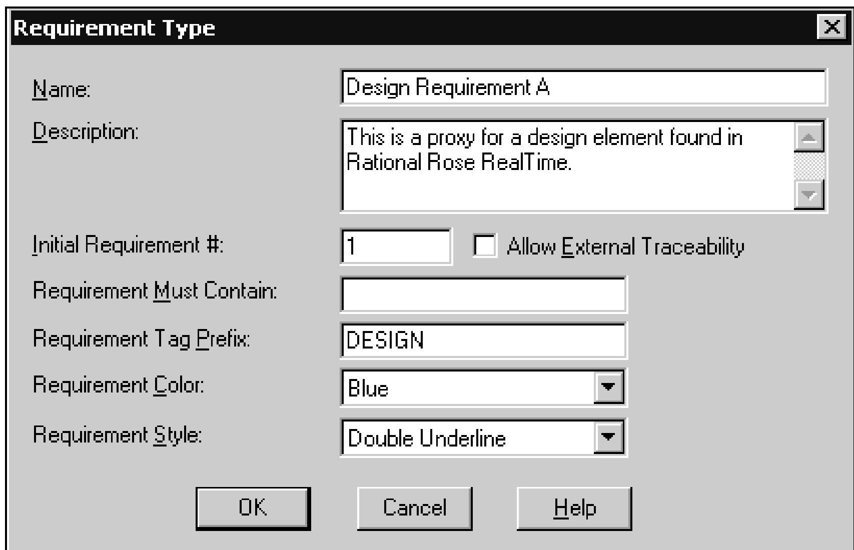
1. Start Rational RequisitePro.
2. Click **File > Properties**.

   Note: If **File > Properties is** unavailable, you must open an existing project or create a new one.

3. In the **Requirement Types** tab, click **Add**.

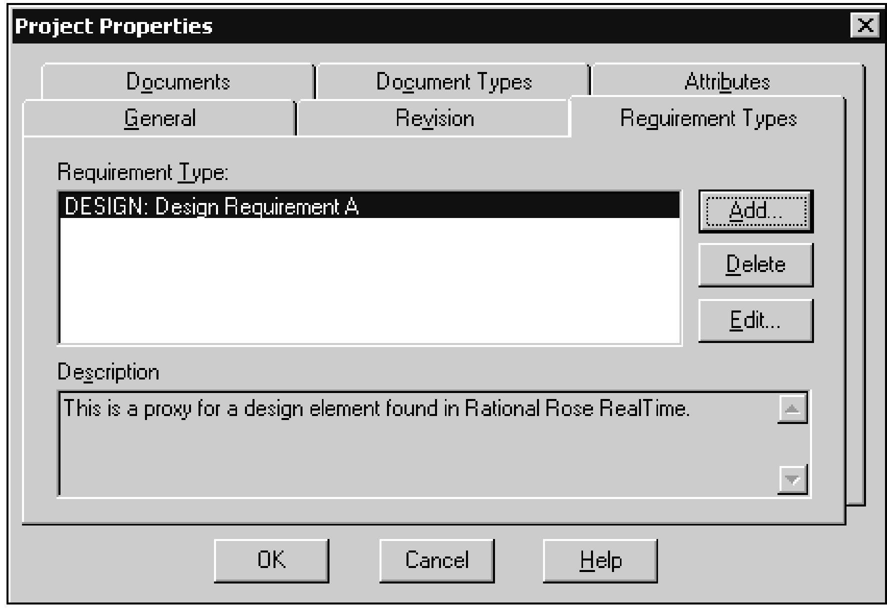


4 Enter the appropriate information in the fields in the **Requirement Type** dialog (as shown below).



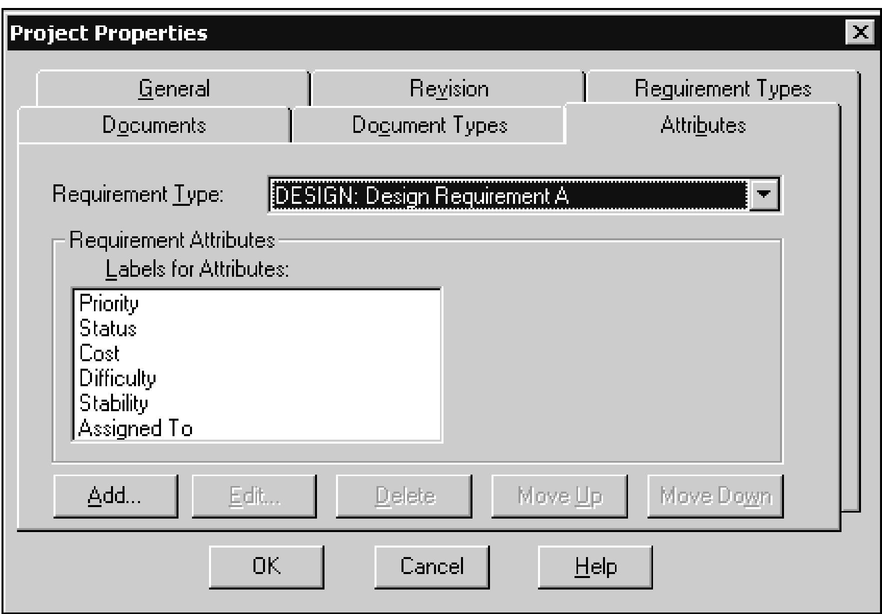
Note: The text in the **Requirement Tag Prefix** box is case-sensitive.

5 Click **OK**.

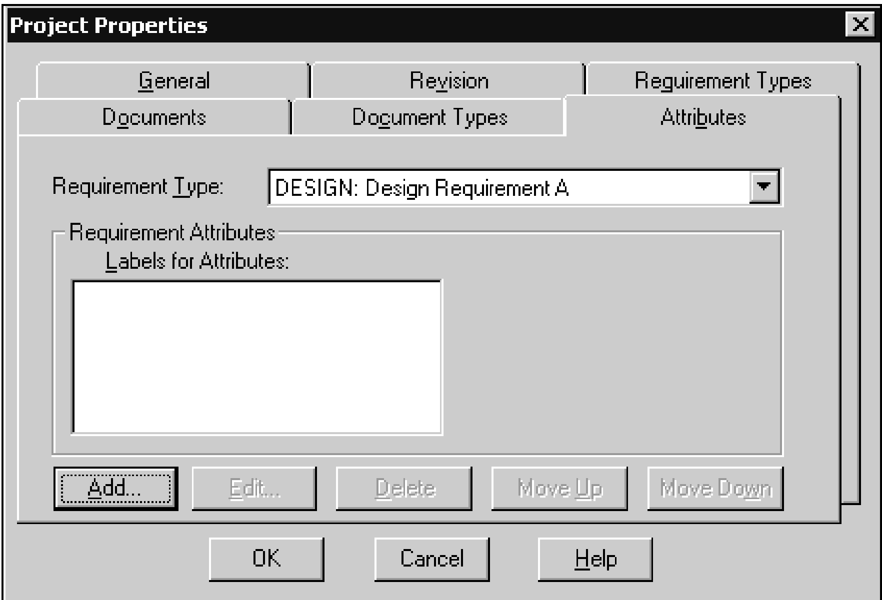


6 Click the **Attributes** tab.

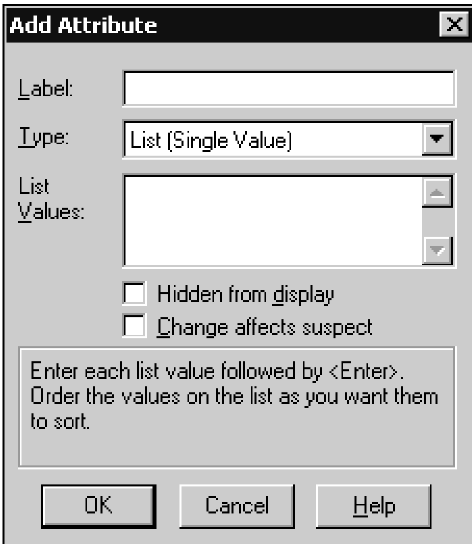
7 Select the **DESIGN** requirement type from the list.



8 If there are any default attributes that appear in the **Labels for Attributes** list in the **Requirement Attributes** box, delete them.



9 Click **Add**.

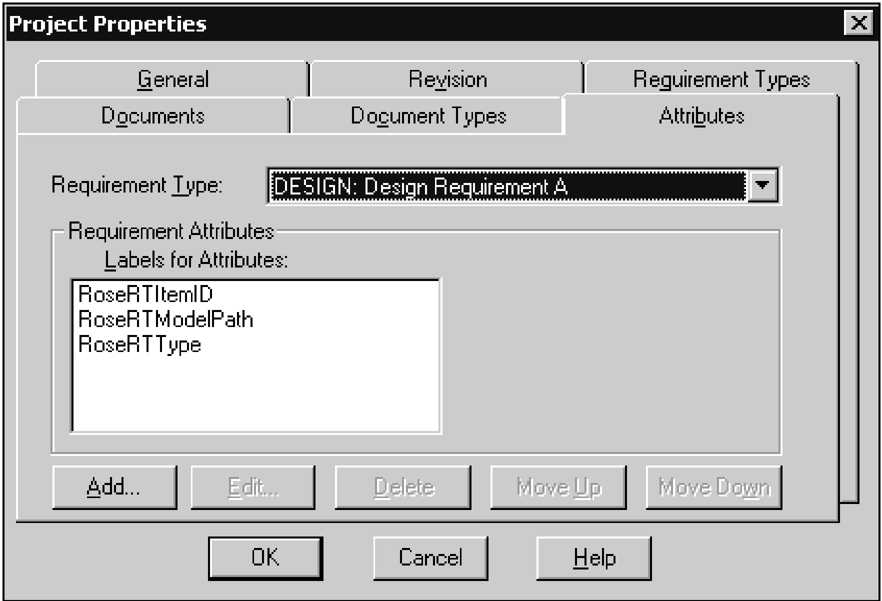


Next, you want to define the attributes specific to the requirement type you created earlier. These attributes appear on the **Attributes** tab on the **Requirement Properties** dialog.

10 Add the following attributes:

Label	Type	Description
RoseRTItemID	Text	Indicates the unique id associated with the model element.
RoseRTModelPath	Text	The location of the model file containing this element.
RoseRTType	Text	Specifies the type of the Rational Rose RealTime element, such as capsule or class.

Note: This requirement type has no document types associated with it. The attributes are case-sensitive.

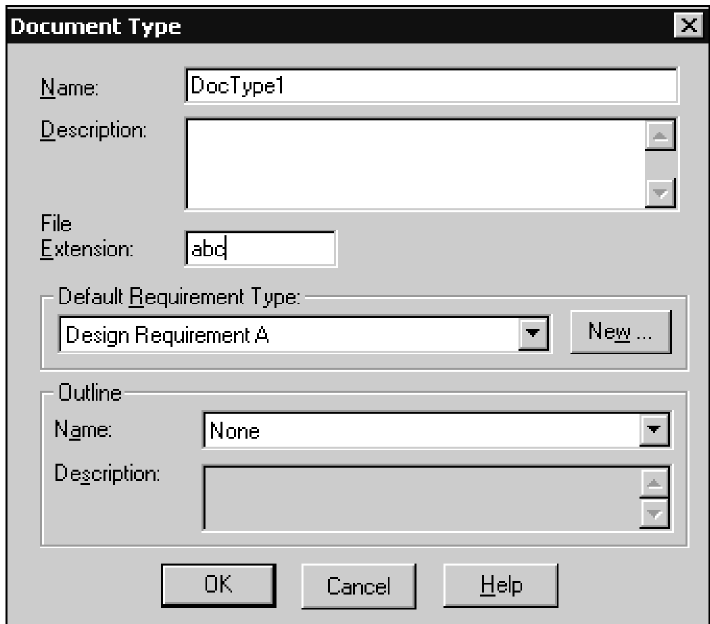


11 Click the **Documentation Types** tab.

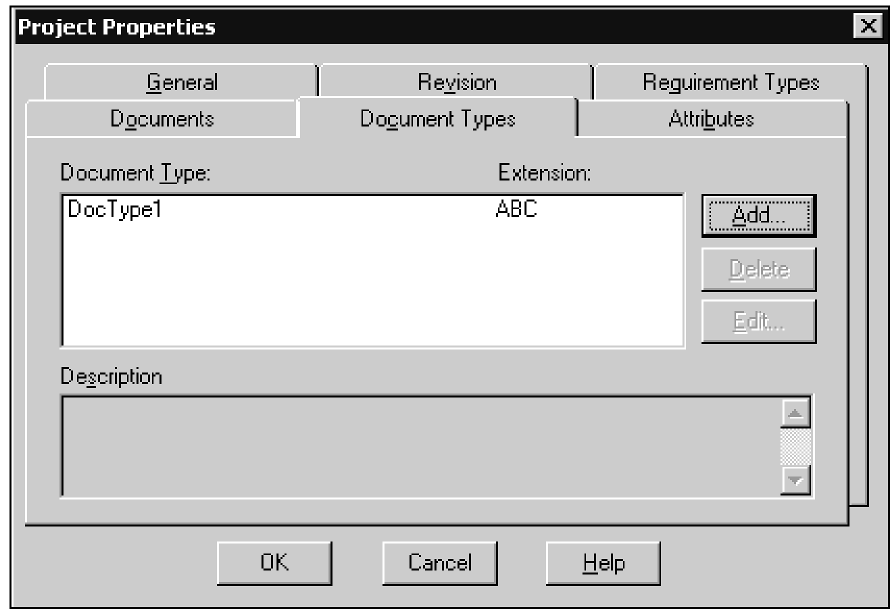
12 Click **Add**.

13 In the **Name** box, specify a name for the documentation type.

14 In the **File Extension** box, specify the file extension to use for the document type.



15 Click **OK**.

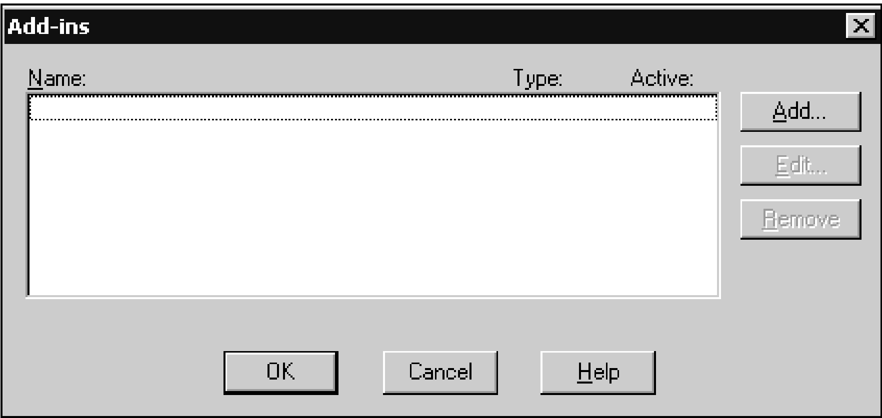


16 Click **OK**.

Next, you want to reference an external menu file. This file will contain customized menu commands and add menu options in Rational RequisitePro.

To add customized menu commands to Rational RequisitePro:

1 Click **Tools > Add-ins**.



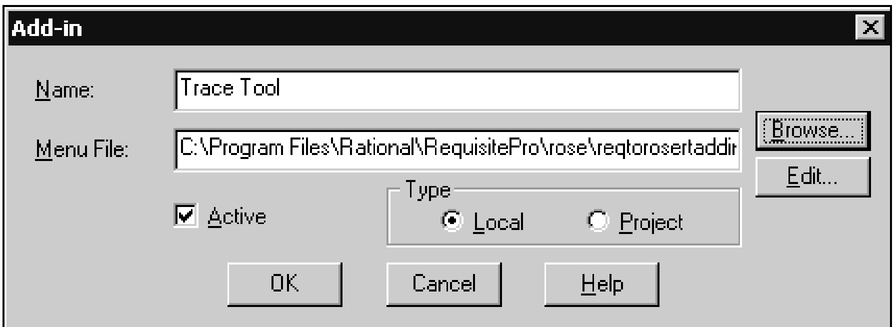
2 Click **Add**.

3 In the **Name** box, type Trace Tool.

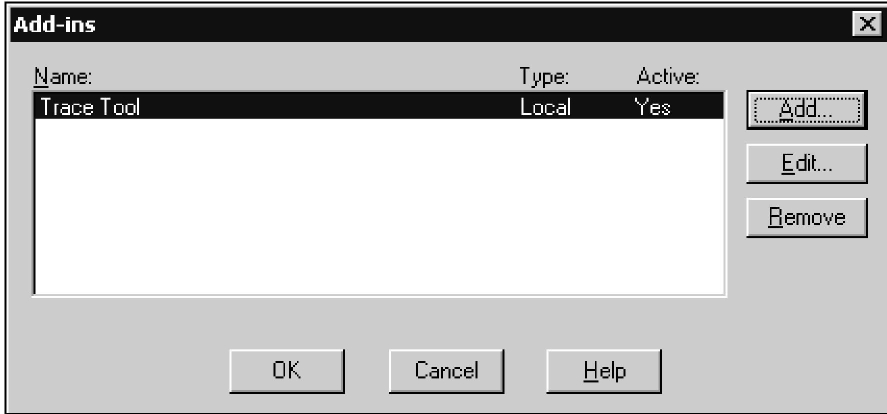
4 Click **Browse**.

5 Browse to the directory where you installed Rational RequisitePro, then locate the file called reqtorosertaddin.mnu in RequisitePro\rose.

6 Select the file reqtorosertaddin.mnu, and click **Open**.



7 Click **OK**.



8 Click **OK**.

Next, you want to associate a Rational Rose RealTime model with a Rational RequisitePro project.

To associate a model to a project:

1 Start Rational Rose RealTime.

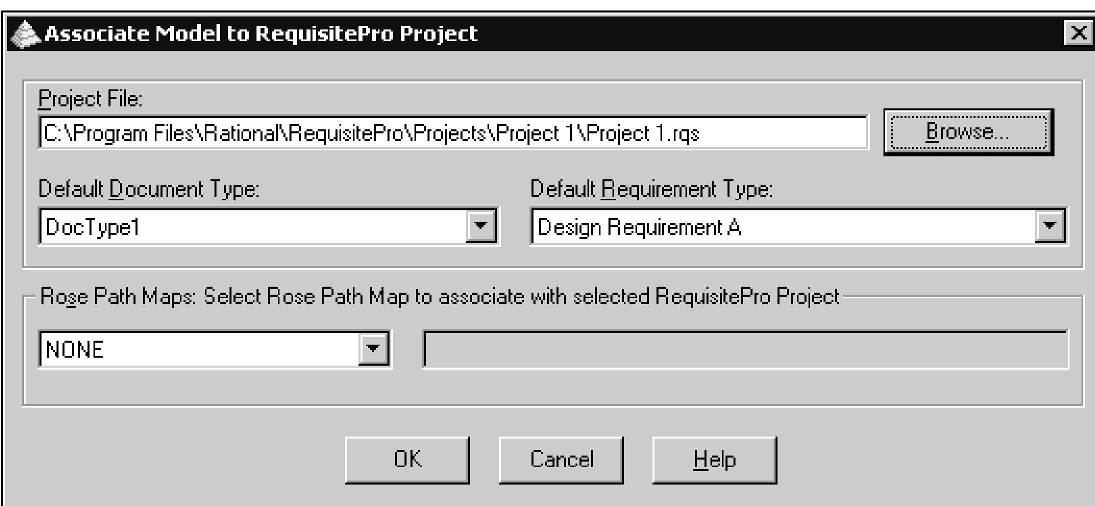
2 Click **Tools > Rational RequisitePro**, and select **Associate Model to Project**.

3 Click **Browse**.

4 Browse to the location of a Rational RequisitePro project that has the desired requirement type.

   For example, we created the requirement type called DESIGN in Rational\RequisitePro\Projects\Project 1\Project 1.rqs.

Note: To associate a model with a project, Rational Rose RealTime requires a Rational RequisitePro project to have at least one document type defined.



5 Click **OK**.

Note: The requirement information is stored within the Rational RequisitePro database. Rational Rose RealTime is not aware of the existence of the corresponding element in Rational RequisitePro. DESIGN requirements (those associated with Rational Rose RealTime elements) only exist in the Rational RequisitePro database provided that the relationship currently exists. If you modify objects in the Rational Rose RealTime model, you can synchronize the data to update the Rational RequisitePro database.

For information on synchronizing data, see *To synchronize Rational RequisitePro data with a Rational Rose RealTime model:* on page 72.

Using the Trace Tool in Rational Rose RealTime

To demonstrate how to use the Trace Tool in Rational Rose RealTime, we will walk through an example that adds a model element to a Rational RequisitePro requirement.

To use the Trace Tool in Rational Rose RealTime, you need:

- *To associate a Rational Rose RealTime model element to a Rational RequisitePro DESIGN requirement:* on page 70
- *To retrieve data from an existing Rational Rose RealTime model element:* on page 71
- *To synchronize Rational RequisitePro data with a Rational Rose RealTime model:* on page 72

To associate a Rational Rose RealTime model element to a Rational RequisitePro DESIGN requirement:

1 Start Rational Rose RealTime.

2 Open a model, and create a new Capsule called **NewCapsule1**.

3 Right-click on **NewCapsule1** from the **Model View** tab in the browser, or from the **Class** diagram.

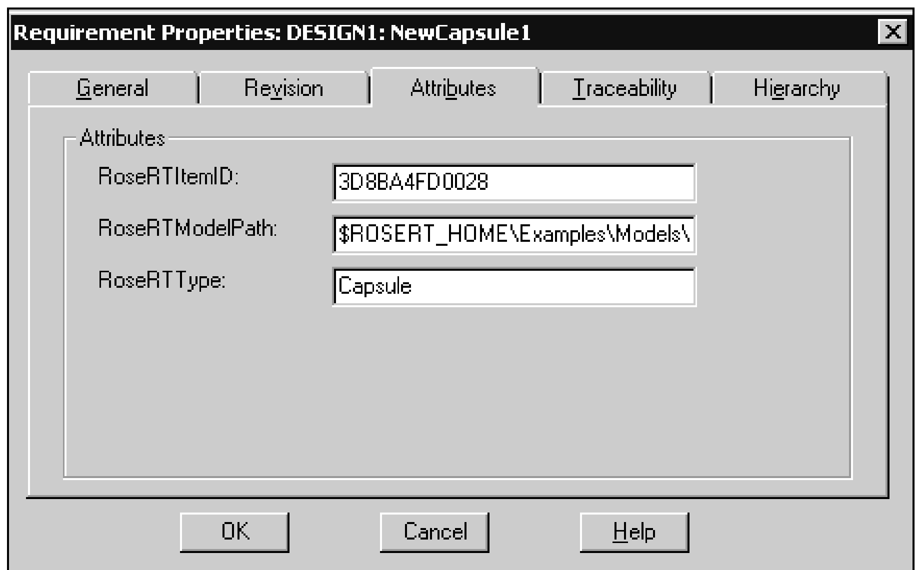
4 Click **Rational RequisitePro Trace Tool > Access Traceability Information**.

Note: If you created a new model, you must save the model before you can select the **Rational RequisitePro Trace Tool > Access Traceability Information** option.

If you have not logged into Rational RequisitePro, you are prompted to log on.

5 Enter the correct logon information.

6 Click **OK** to add a new design requirement to Rational RequisitePro.



The **Requirement Properties** dialog displays the Rational RequisitePro data for the new **DESIGN** requirement.

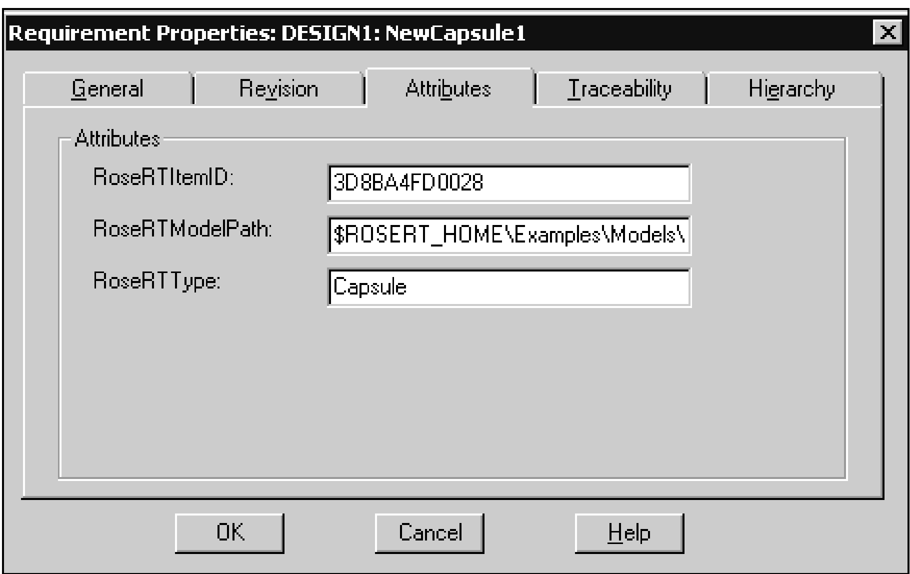
7 Click **OK** to save the new information.

8 To view the element in the corresponding Rational RequisitePro project, click **View > Refresh**.

To retrieve data from an existing Rational Rose RealTime model element:

1 In Rational Rose RealTime, select a model element that currently has a DESIGN requirement associated with it.

2 Right-click on the model element and select **Rational RequisitePro Trace Tool > Access Traceability Information**.



The **Requirement Properties** dialog displays the Rational RequisitePro data for the existing DESIGN requirement.

3 Click **OK**.

To synchronize Rational RequisitePro data with a Rational Rose RealTime model:

1. Open a Rational Rose RealTime model associated with a Rational RequisitePro project.
2. Modify Rational Rose RealTime model elements associated with a Rational RequisitePro requirement, such as deleting or renaming an element in a model.
3. In Rational Rose RealTime, click **Tools > Rational RequisitePro**, and select **Sync Traceability Information**.

The **Trace Tool** dialog shows information about the data updated in Rational RequisitePro.

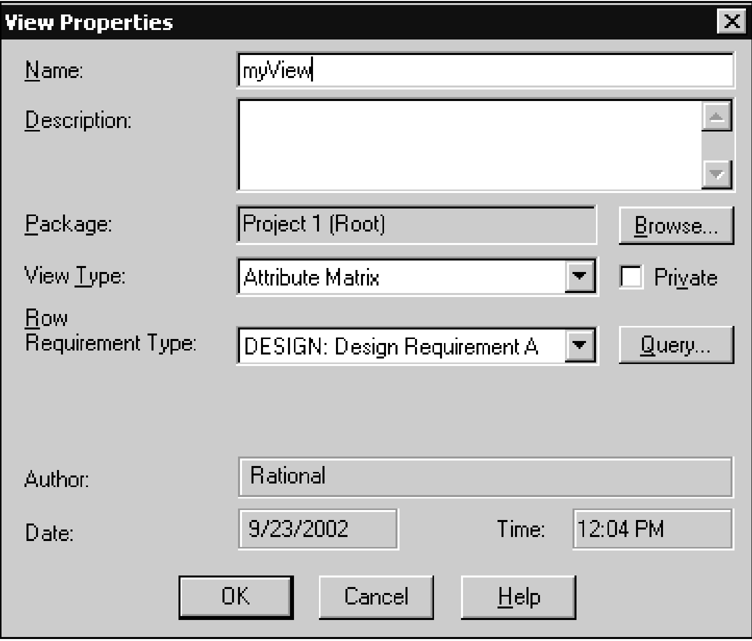
Using the Trace Tool in Rational RequisitePro

The following procedures demonstrate how to use the **Trace Tool** in Rational RequisitePro by adding a model element as a Rational RequisitePro requirement.

To use the Trace Tool in Rational RequisitePro:

1. Start Rational Rose RealTime.
2. Start Rational RequisitePro and open a project that is currently associated with a Rational Rose RealTime model.
3. If a view does not currently exist, click **File > New > View**.

4 In the **Name** box, enter a name for the view.



5 Click **OK**.

6 Open the view by double clicking on it in the Rational RequisitePro explorer.

7 Select one of the items in the view list.

8 From the **Requirement** menu, click **RoseRT Trace Tool > Go To RoseRT Element**.

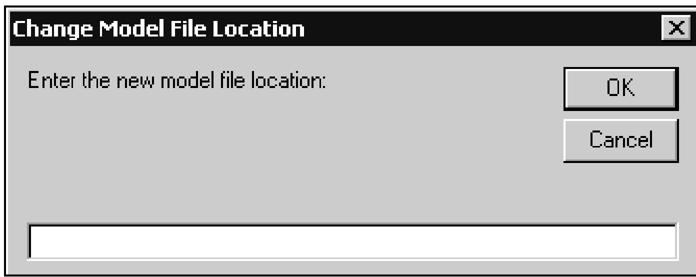
If the selected element was a diagram, that diagram opens in Rational Rose RealTime. Otherwise, the associated Rational Rose RealTime **Specification** dialog opens for the selected element.

Updating Rational RequisitePro Requirements when a Model File Location Changes

Rational RequisitePro stores the location of a Rational Rose RealTime model in each of its requirements. If the location of the model file changes, you must update each of the requirements with the new location. Rational RequisitePro allows you to review all of the requirements in a project, and update the model file location for specified requirements.

To update all requirements with the new model file location:

1. Start Rational Rose RealTime.
2. Start Rational RequisitePro, and open a project associated with a Rational Rose RealTime model.
3. Click **Requirement > RoseRT Trace Tool > Change Model File Location**.



4. Specify a valid location.
5. Click **OK**.
6. If the name is correct, click **Yes**, otherwise click **No**.

Note: If you click **No**, the next model name in Rational RequisitePro appears. These dialogs will continue until there are no model names remaining in the database, or until you click **Yes**.

Rational Rose RealTime Wizards

3

Contents

This chapter is organized as follows:

- *Component Wizard* on page 75
- *TargetRTS Wizard* on page 83

Component Wizard

The **Component Wizard** helps you to quickly create C++ and C Executable components. A component describes how to build a set of capsules and classes. To run a model, you must build it (by selecting a component to build) and then execute it on a processor.

The **Component Wizard** allows you to configure the following component properties:

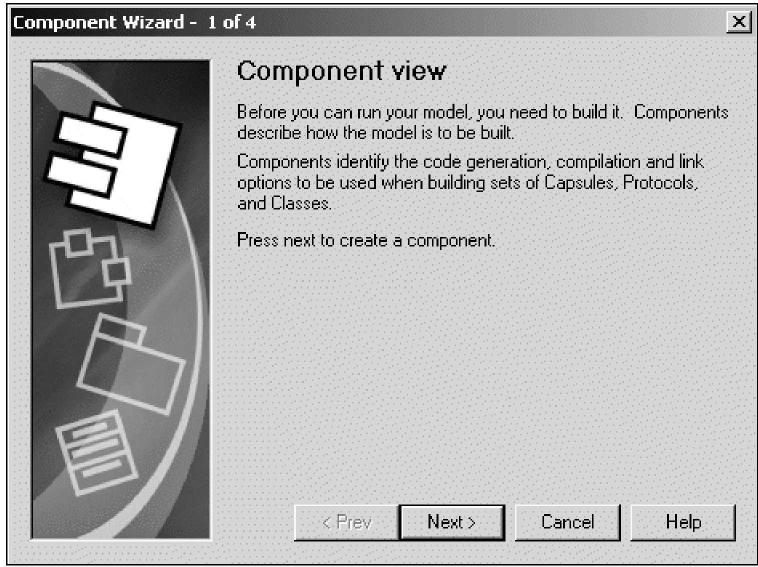
- Language
- Top level capsule
- Output directory
- Executable name
- Target Configuration

To access the **Component Wizard**, click **Build > Component Wizard**.

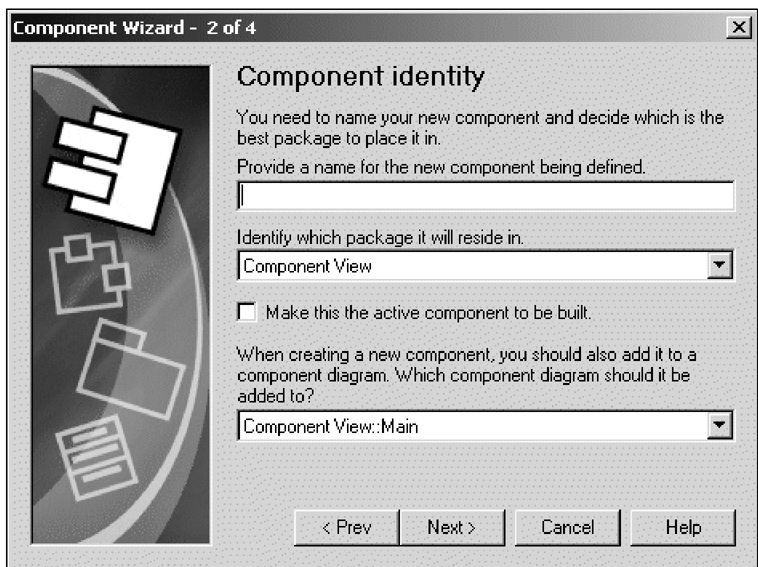
Note: You cannot create Libraries and External Libraries with the **Component Wizard**.

To provide the initial information for a component:

1 From the **Build** menu, click **Component Wizard.**



2 Click **Next** to start.



3 Specify a meaningful name for the component.

4 Specify the location of the package in the **Model View** tab in the browser. The default location is the **Component View** package.

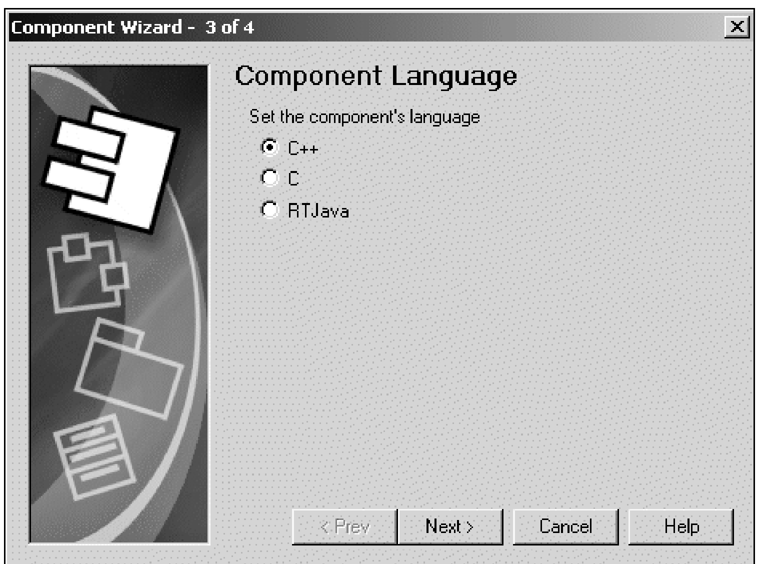
5 Indicate whether this component is the active component.

   If you frequently build and run the same component and component instances, set this component as an active component. When a component is configured as being active, the Toolbar build icons and menu items become available for easy access to common build and run commands. In addition, you can configure which component instances (executables) automatically run when you click **Run**. You can set this option later by selecting the component from the **Model View** tab in the browser, right-clicking, and then selecting **Set As Active**.

6 Select the name of the diagram to add the component instance.

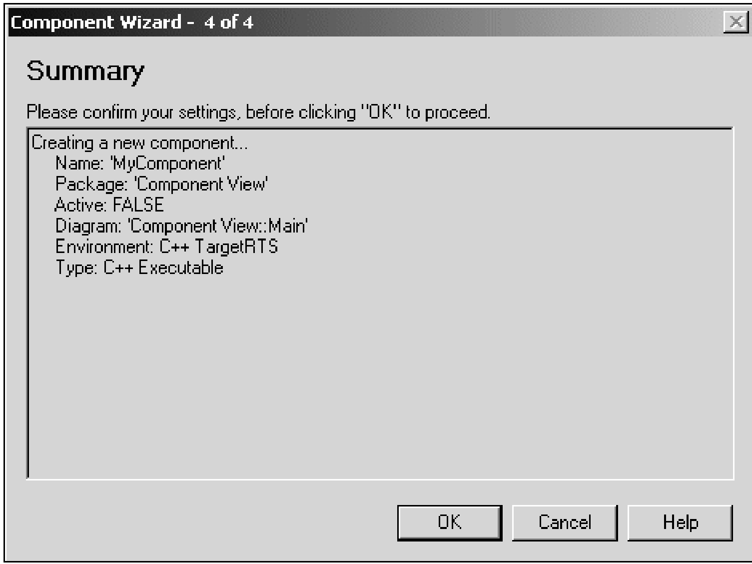
   Adding the component instance to the diagram provides you with a graphical representation of the components in your model.

7 Click **Next**.



8 Select a language for the component.

9 Click **Next**.



10 Review the summary of specified settings.

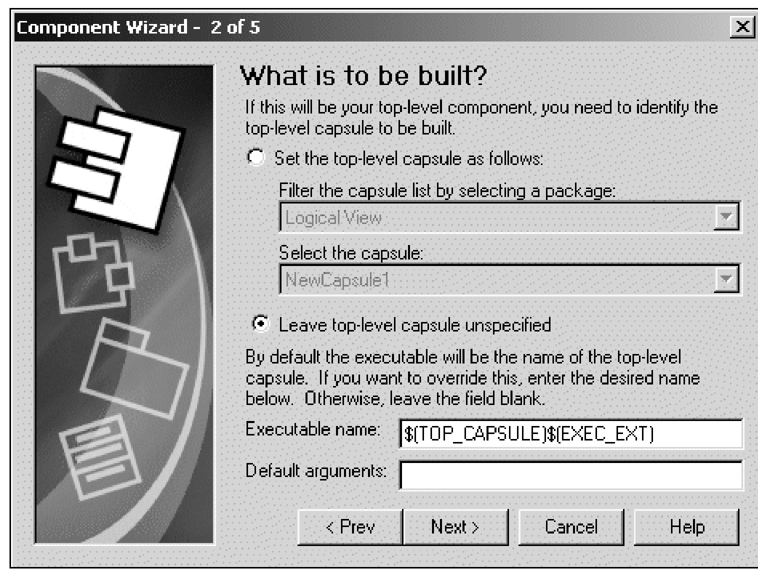
11 Click **OK** to proceed.

12 Do one of the following:

- Click **Next** to customize the component (recommended).

Or . . .

- Click **Cancel** to create the component without customization and to close the **Component Wizard**.



13 You can either set the top level capsule or leave it unspecified.

If you specify a top level capsule to compile for this component, the top capsule will define the compilation closure for the component. All classes, including capsule and protocol classes referenced directly or indirectly by the top capsule are then compiled as part of the component.

14 Specify an **Executable name**.

You can specify the name, or a name with an absolute path, of the executable that is created when the component is built. By default, the executable name is set to the name of the component's top level capsule.

Note: If an absolute path is not used in the **Executable name** box, the location of the executable will be in the following component build output directory:

<output_dir>/build

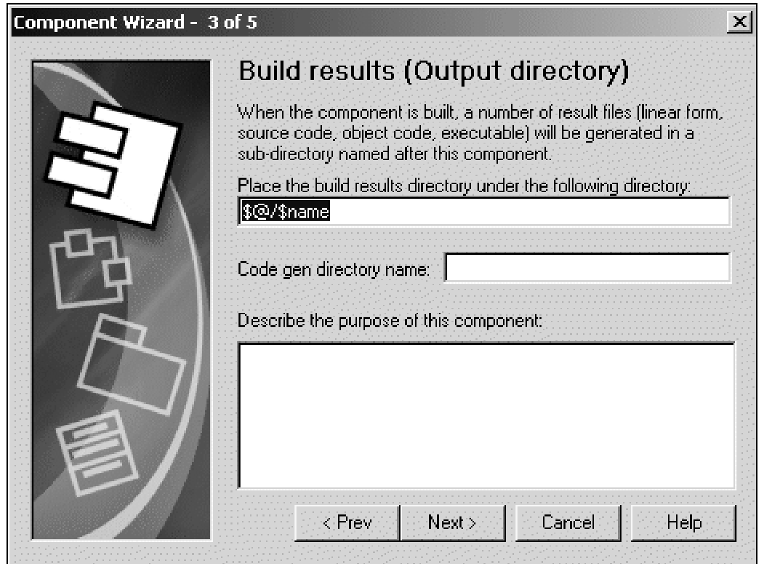
15 Specify any **Default arguments**.

Some platforms do not permit the passing of command line arguments to an executable at load time. As a result, the **Default arguments** box provides a mechanism for getting execution arguments into the executable. You can use `RTMain::argStrings()` to retrieve any passed command line argument within your model. Type a comma-separated list of quoted arguments into this box, such as:

"134.434.344.4","barneyht","delay=98"

The **Default arguments** box is only for targets that cannot accept command line arguments. Targets that can accept command line arguments ignore anything in the **Default arguments** box.

16 Click **Next**.

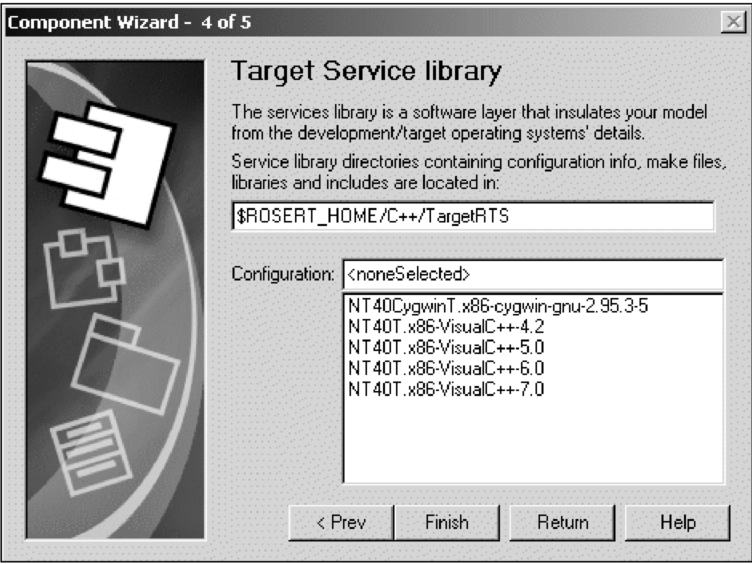


17 Specify the location for the build results.

18 Specify the name of the **Code gen** directory.

19 Provide a description that identifies the purpose of this component.

20 Click **Next**.



21 Specify the location of the Services Library directories that contains the configuration information, make files, libraries and include files.

Specify the path to the root directory for the specific Services Library desired. This name must be specified as a full path to the root directory of the Services Library.

The Target Services directory contains all the scripts and programs to generate and compile a component. If this directory is not configured correctly, you will not be able to successfully generate or compile.

By default this field references the Services Library in your Rose RealTime home directory $ROSERT_HOME/C++/TargetRTS. You can change this location to any other directory that contains the C++ Services Library.

22 Select a **Configuration**.

This property uniquely identifies the configuration of the Services Library used to compile and link the component. The configuration name is composed of three parts: os.processor-compiler-version.

For example, the configuration for a Windows NT 4.0 multi-threaded platform with an x86 processor built with Microsoft Visual C++ version 6.0 is:

```
NT40T.x86-VisualC++-6.0
```

To view the valid configuration names, examine the directories located in the \lib subdirectory of the Services Library root. If you build different configurations of the Services Library, the new configuration appear in this list.

23 Do one of the following:

- Click **Finish**.

Or . . .

- Click **Return** to modify previous settings.

24 Review the contents of the **Summary** window.

25 Click **OK** to create the component.

26 Click **OK** and verify your component in the **Model View** tab in the browser.

TargetRTS Wizard

The **TargetRTS Wizard** facilitates the management of the TargetRTS source tree, allows easy customization of existing TargetRTS libraries, and simplifies porting of the TargetRTS to new targets. With the **TargetRTS Wizard**, you can create a new TargetRTS configuration, modify or duplicate an existing configuration, or delete an existing configuration that is no longer required.

Note: Porting to a new operating system or a **libset** is not a trivial process, even with the help of the **TargetRTS Wizard**. You must be familiar with the operating system, the toolchain, the TargetRTS, and its layout.

For additional information, see the book *Adapting for Target Environments, Rational Rose RealTime*.

Note: All figures that appear in this topic are for the C++ language.

Understanding the TargetRTS

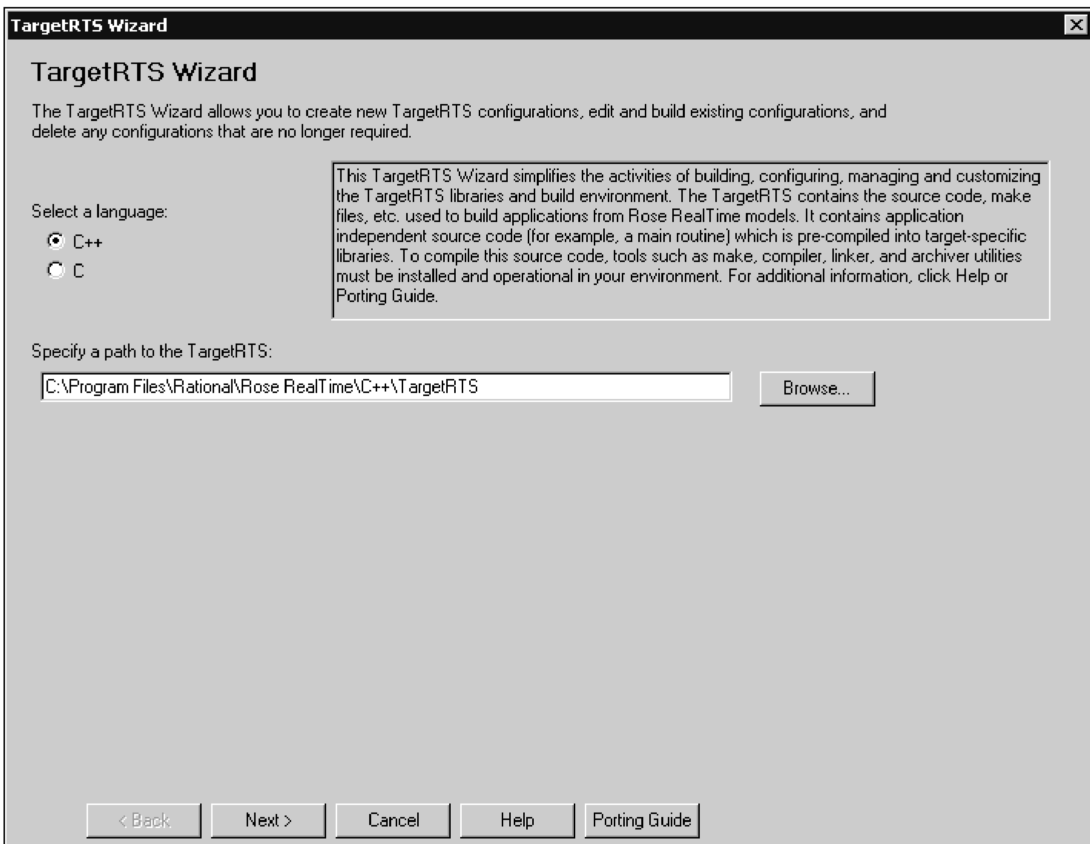
The TargetRTS is the set of run-time services that provide a framework in which a Rational Rose RealTime model can run. The **TargetRTS Wizard** simplifies the activities of building, configuring, managing, and customizing the TargetRTS libraries and build environment.

The TargetRTS contains the required parts, such as source code and **makefiles**, used to build applications from Rational Rose RealTime models. It contains application-independent source code which is pre-compiled into target-specific libraries. To compile this source code, tools such as **make**, compiler, linker, and archiver utilities must be installed and operational in your environment.

Maintaining TargetRTS Libraries using the TargetRTS Wizard

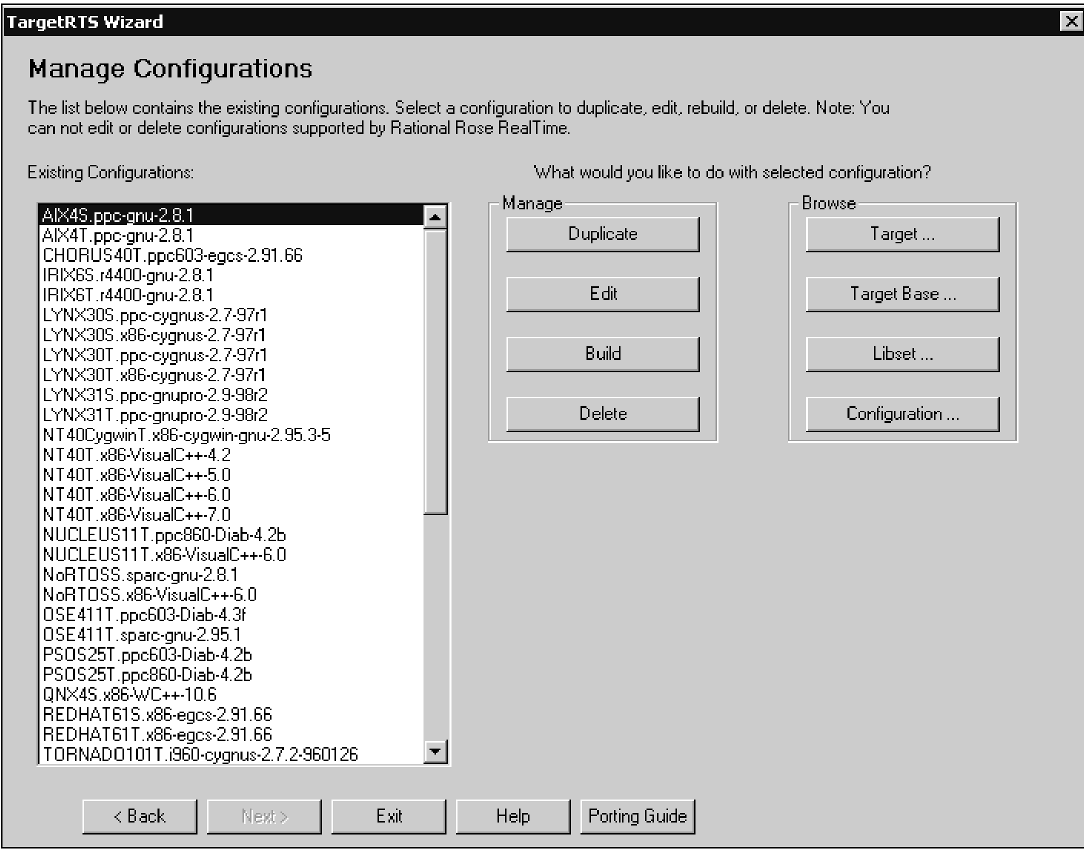
To access the TargetRTS Wizard, click **Tools > TargetRTS Wizard**. Figure 25 shows the first pane in the TargetRTS Wizard.

Figure 25 TargetRTS Wizard - First Pane



Locate the TargetRTS tree for the **TargetRTS Wizard**, and then click **Next**.

Figure 26 TargetRTS Wizard - Manage Configurations panel



The **Existing Configurations** box contains a list of all your configurations. For some configurations, you can **Duplicate**, **Edit**, **Build**, or **Delete**, as required.

Note: Those configurations distributed with Rational Rose RealTime are read-only and cannot be edited or deleted. To modify a Rational Rose RealTime configuration that is read-only, select the configuration and click **Duplicate**.

For additional information on modifying a Rational Rose RealTime configuration, see *Duplicating a Configuration* on page 86.

Managing Your TargetRTS Configurations

When managing configurations with the **TargetRTS Wizard**, you can:

- Click **Duplicate** for *Duplicating a Configuration* on page 86
- Click **Edit** for *Editing a Configuration* on page 90
- Click **Build** for *Building Configurations* on page 97
- Click **Delete** for *Deleting Configurations* on page 99
- Click a browse option for *Browsing Directories* on page 86

Browsing Directories

You can browse other directories for configurations to quickly view the files necessary for each configuration. The **TargetRTS Wizard** opens the files in the external editor you specified in the **Path** box on the **Editor** tab by clicking **Tools > Options**.

Duplicating a Configuration

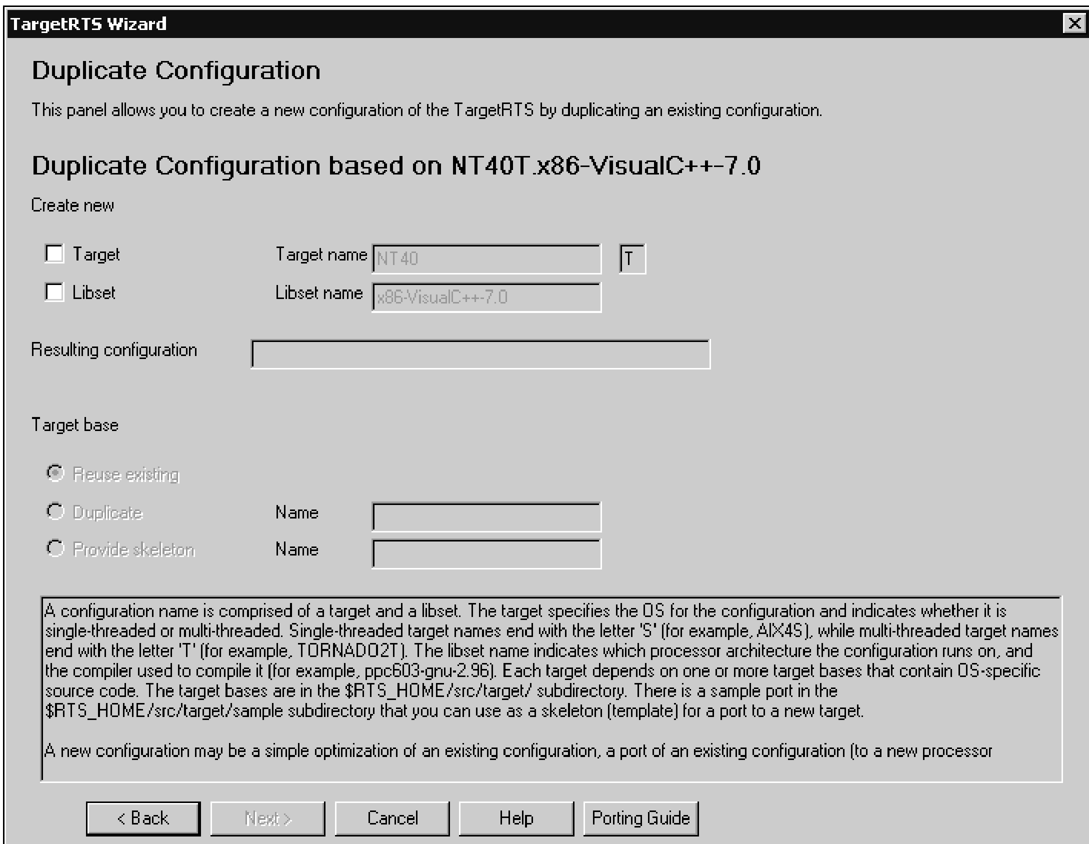
Duplicating an existing configuration is the first step to creating new configurations for new ports, or for a custom version of the same configuration.

Note: The configuration name is an important identifier of the TargetRTS. It identifies the operating system, hardware architecture, and compiler.

To duplicate a configuration:

1. In the **Existing Configuration** box on the **Manage Configuration** pane, select a configuration.
2. In the **Manage** box, click **Duplicate**.
3. Click **Next**.

Figure 27 TargetRTS Wizard - Duplicate Configuration panel



A new configuration can be:

- a simple optimization of an existing configuration
- a port of an existing configuration (to a new processor architecture or to a new compiler)
- a port to an entirely new OS

Since the new configuration must have a new name, you must create a new **Target**, a new **Libset**, or both.

The **Target** specifies the OS for the configuration and indicates whether it is single-threaded or multi-threaded. Single-threaded target names end with the letter 'S' (for example, AIX4S), while multi-threaded target names end with the letter 'T' (for example, TORNADO2T). The **Libset** name indicates which processor architecture the configuration runs on, and the compiler used to compile it (for

example, ppc603-gnu-2.96). Each target depends on one or more target bases that contain OS-specific source code. The **Target bases** are in the $ROSERT_HOME/src/target/ directory.

Note: There is a sample port in $ROSERT_HOME/src/target/sample that you can use as a skeleton (a template) for a port to a new target.

4 In the **Create new** label, select **Target** to specify a new name in the **Target name** box.

The **Target name** represents the implementation-specific components of the TargetRTS. These components are generally specific to a given configuration, of a given version, of a given operating system. The **Target name** is also used to name the configuration of the target, such as single-threaded versus multi-threaded. The target name is defined as follows:

<target> ::= *<OS_name><OS_version><RTS_config>*

The components of `<target>` are defined as follows:

<OS_name> identifies the operating system (for example, SUN)

<OS_version> identifies the major version of that operating system.

> **Note:** Do not use periods in the OS version because this will confuse the **make** utility when it attempts to build the TargetRTS.

<RTS_config> is a single letter that identifies the configuration; "**S**" for a single-threaded configuration, and "**T**" for a multi-threaded configuration.

For example:

```
SUN5T
```

If you select **Target**, the **Target base** area of the panel becomes enabled. The **Target base** controls the OS-specific source code used for the new target. If the duplicate configuration is a port to a different operating system, a new target base will be necessary. Duplicating a target base copies the target base used for the original target, and you may have to modify the new base. A skeleton target base contains only stubs for functions that are required for any target. These functions must be fully implemented and you may need to add additional functions.

You can specify a **NoRTOS** target base that does not use any OS-specific calls. For more information on using a **NoRTOS** target base, see *NoRTOS Target Base* on page 90.

Note: To reuse existing targets to create new configurations, you can specify the name of an existing **target** in the **Target name** box. The **TargetRTS Wizard** creates a new configuration (using the selected libset and the existing **target**), and the **target** will not be copied.

5 In the **Create new** label, select **Libset** to specify a new name in the **Libset name** box.

Although the actual **libset** names can be chosen arbitrarily, by convention, those used by Rational Rose RealTime are defined as follows:

<libset> ::= *<processor>-<compiler_name>-<compiler_version>*

The components of *<libset>* are defined as follows:

<processor> identifies the processor architecture name

<compiler_name> identifies the compiler product name, or the vender for the compiler.

<compiler_version> identifies the compiler version. It is acceptable to use periods in the compiler version text.

For example:

```
sparc-gnu-2.8.1
```

Note: To reuse existing **libsets** to create new configurations, you can specify the name of an existing **libset** in the **Libset name** box. The **TargetRTS Wizard** creates a new configuration (using the selected target and the existing **libset**), and the **libset** will not be copied.

The **Resulting Configuration** box contains the name of the configuration.

6 Click **Next**.

The **TargetRTS Wizard** presents a **Summary** dialog that identifies all of the actions it will perform.

7 Click **Next**.

When appropriate, the **TargetRTS Wizard** displays a **Work Order** dialog containing a list of items that may require user intervention.

8 Click **Next**.

NoRTOS Target Base

Both the C and C++ TargetRTS have a NoRTOS target base that does not use any OS-specific calls. This means that a NoRTOS target base will work with any OS, or it will work without an OS. A single-threaded target (NoRTOSS) uses the NoRTOS target base.

Often when porting to a new operating system, it is useful to create the **libset**, and then use it with the NoRTOSS target to verify that the toolchain works properly. After the OS-independent version of the port is complete, you can use its **libset** with a new target to make the full port.

To create a configuration that uses a NoRTOS target base using the TargetRTS Wizard:

1. In the **Existing Configuration** box in the **Manage Configuration** dialog, select a configuration that uses the **NoRTOSS** target.
2. In the **Manage** box, click **Duplicate**.
3. In the **Create new** box, select **Libset**.
4. In the **Libset name** box, specify an appropriate name for the **libset**.

   Note: For some situations where the new **libset** is similar to an already existing **libset**, it may be useful to specify the name of that existing **libset** into the **Libset name** box. The **TargetRTS Wizard** will then reuse that **libset** in the new configuration. The resulting configuration can then be duplicated to properly name the new **libset**. The **TargetRTS Wizard** will then use this **libset** with the new target to create the new configuration.

Editing a Configuration

After you duplicate a configuration, you can edit the new configuration. You can edit the **target**, the **libset**, or only the configuration itself.

Note: You cannot edit the configurations that are included with Rational Rose RealTime, nor the targets and **libsets** that these configurations use. You can only edit the configurations that you duplicated previously.

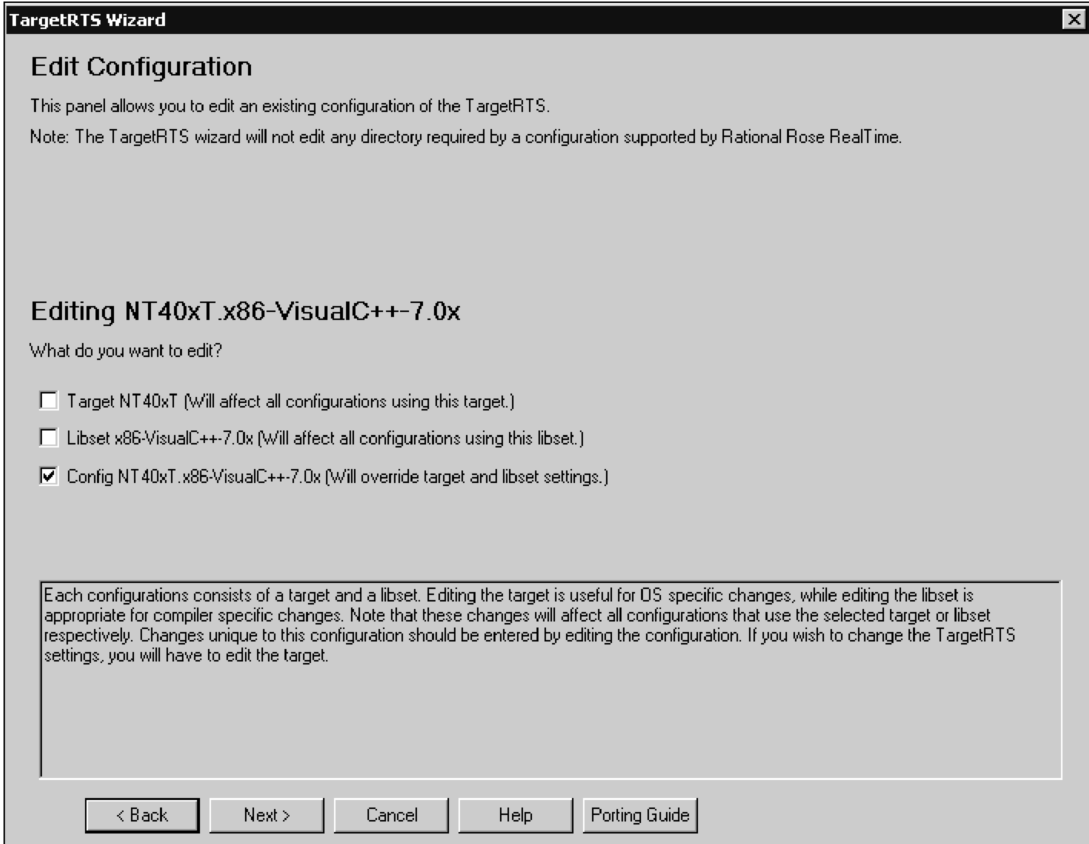
Every configuration is comprised of a target and a **libset**. Editing the target is useful for OS-specific changes, while editing the **libset** is appropriate for compiler-specific changes. To change the TargetRTS settings, you will need to edit the target.

Note: These changes affect all configurations that use the selected target or **libset**.

Figure 28 shows the **Edit Configuration** pane in the **TargetRTS Wizard**. From this pane, you can specify whether you want to edit a combination of the target, **libset**, or the configuration itself. For more information on editing, see the following:

- *Editing the Target* on page 92
- *Editing the Libset* on page 95
- *Editing a Configuration* on page 96

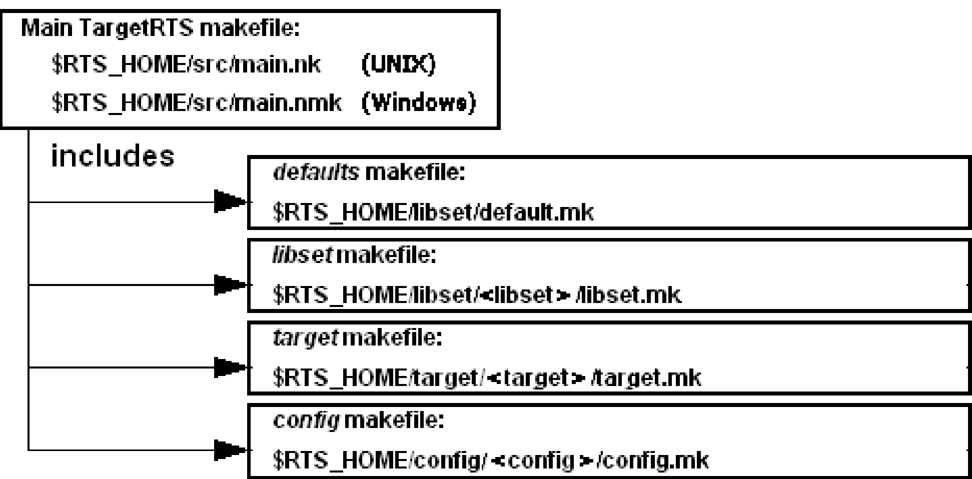
Figure 28 TargetRTS Wizard - Edit Configuration panel



Understanding the makefiles

When you edit a configuration using the **TargetRTS Wizard**, you are modifying properties in one or more **makefiles**. Figure 29 shows the **makefiles** that you can update when specifying particular options while using the **TargetRTS Wizard**.

Figure 29 TargetRTS makefiles



The default.mk, libset.mk, target.mk, and config.mk **makefiles** are used to compile both the TargetRTS libraries and the model. The target.mk, libset.mk and config.mk **makefiles** override the defaults defined in $ROSERT_HOME/libset/default.mk. These are the **makefiles** that you can edit using the **TargetRTS Wizard**.

The main.nmk (**nmake** for Windows) or main.mk (**make** for UNIX) is the main definition for compiling the TargetRTS libraries. These **makefiles** should not be customized, and will not be discussed further in this document.

The default.mk file contains the default macro definitions that may be overridden by the platform-specific **makefiles**.

The target.mk file contains the definition specific to the target operating system.

The libset.mk file contains the definition specific to the compiler.

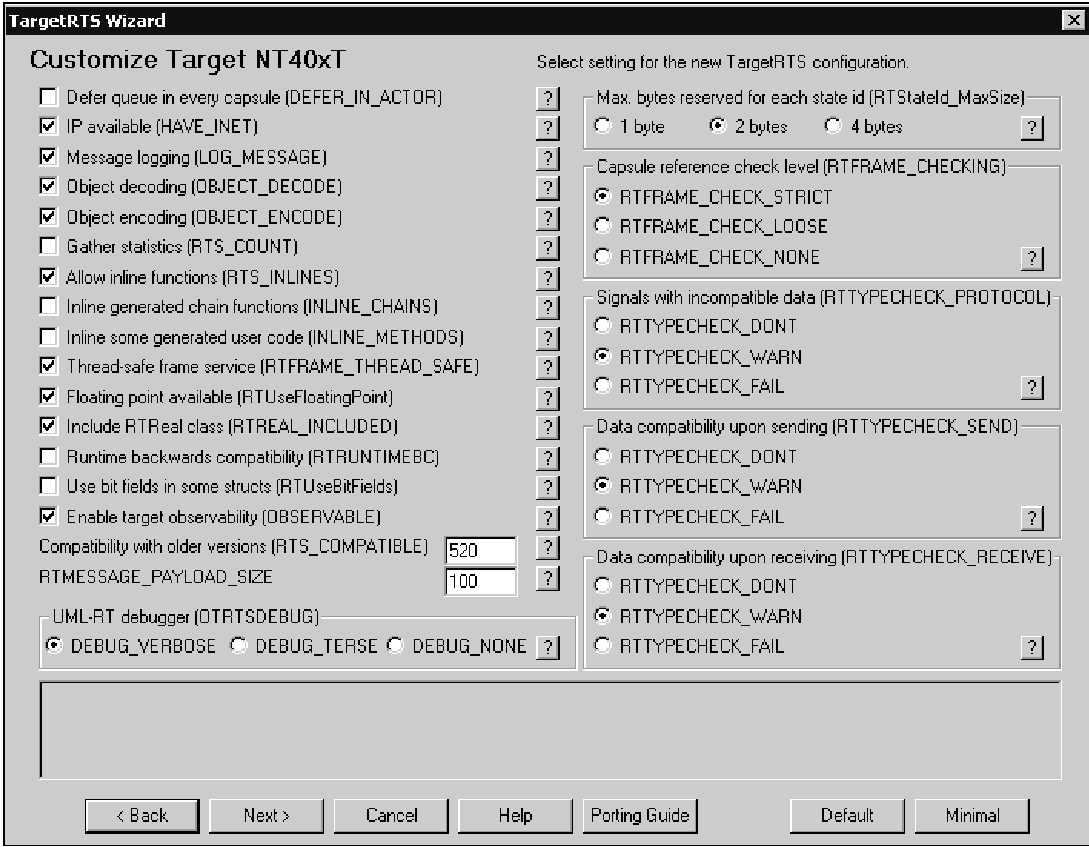
The config.mk file contains the definition specific to the combination of the compiler, operating system, and TargetRTS configuration.

Editing the Target

You can edit the target to create a custom TargetRTS library. Figure 30 shows the C++ options used to configure the run-time system.

Note: The Customize Target panel in the **TargetRTS Wizard** for C is similar to C++; however, some of the individual options differ. For additional information, click the question mark opposite each option.

Figure 30 TargetRTS Wizard - Customize Target panel



Note: Each entry is associated with a macro that controls that particular option in the TargetRTS source. Click **Default** to set all the options back to their defaults. Click **Minimal** to set the options for a much smaller and faster run-time system.

After you specify your required target options, click **Next**.

Figure 31 shows the **Target Settings** panel used to control compiler and linker flags for the target. The **Set** options control which variables are defined in the target.mk file for that particular target.

Figure 31 TargetRTS Wizard - Target Settings panel

Descriptions

Target Compiler Flags (TARGETCCFLAGS)

Adds target-specific compilation flags in the file target.mk.

Target Linker Flags (TARGETLDFLAGS)

Redefines the target linker flags in the target.mk file.

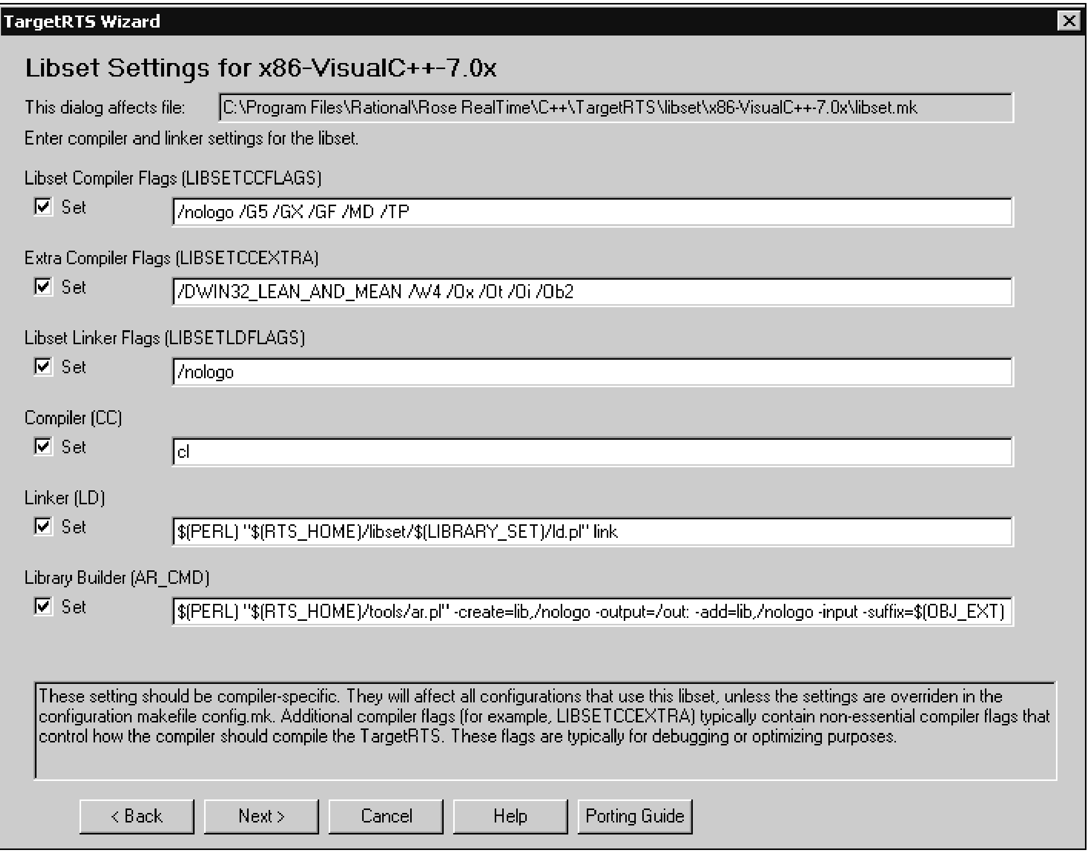
Note: These flags should be target-specific. They will affect all configurations that use this target unless you override them on the **Configuration Setting** panel of the **TargetRTS Wizard**.

Editing the Libset

You want to edit a **libset** to change the it to a different CPU architecture or a different compiler, or to change how the TargetRTS library is built (for example, changing compiler flags).

Figure 32 shows the options for configuring the **libset**. The **Set** options control which variables are defined in the libset.mk file for that particular **libset**. The text boxes to the right of the **Set** options contain their current values.

Figure 32 TargetRTS Wizard - Libset Settings panel



Descriptions

Libset Compiler Flags (LIBSETCCFLAGS)

Adds compiler-specific compilation flags in the file libset.mk.

Extra Compiler Flags (LIBSETCCEXTRA)

Specifies any non-essential compiler flags that control how the compiler should compile the TargetRTS. These flags are used to compiles the TargetRTS library, but do not compile the models. Typically, you would specify optimization flags in this box.

Libset Linker Flags (LIBSETLDFLAGS

Adds compiler-specific linker flags in the libset.mk file.

Compiler (CC)

Specifies the name of the C or C++ compiler executable.

Linker (LD)

Specifies when a linker must be different from compiler (most compilers can invoke the linker), or if a preprocessing script is necessary.

Library Builder (AR_CMD)

Specifies a command to run the library utility.

Editing a Configuration

Editing a configuration overrides settings from the target.mk and libset.mk files. The overridden settings apply only to the selected configuration, and they are stored in that configuration's config.mk file.

Figure 33 shows the override options for the configuration. These are the same options that appear on the **Libset Settings** and the **Target Settings** panels in the **TargetRTS Wizard**.

Figure 33 TargetRTS Wizard - Configuration Settings panel

Building Configurations

To build an existing configuration of the TargetRTS, you must specify the **make** command used by the build. Figure 34 shows the **Build Configuration** pane which you can use to compile the TargetRTS libraries.

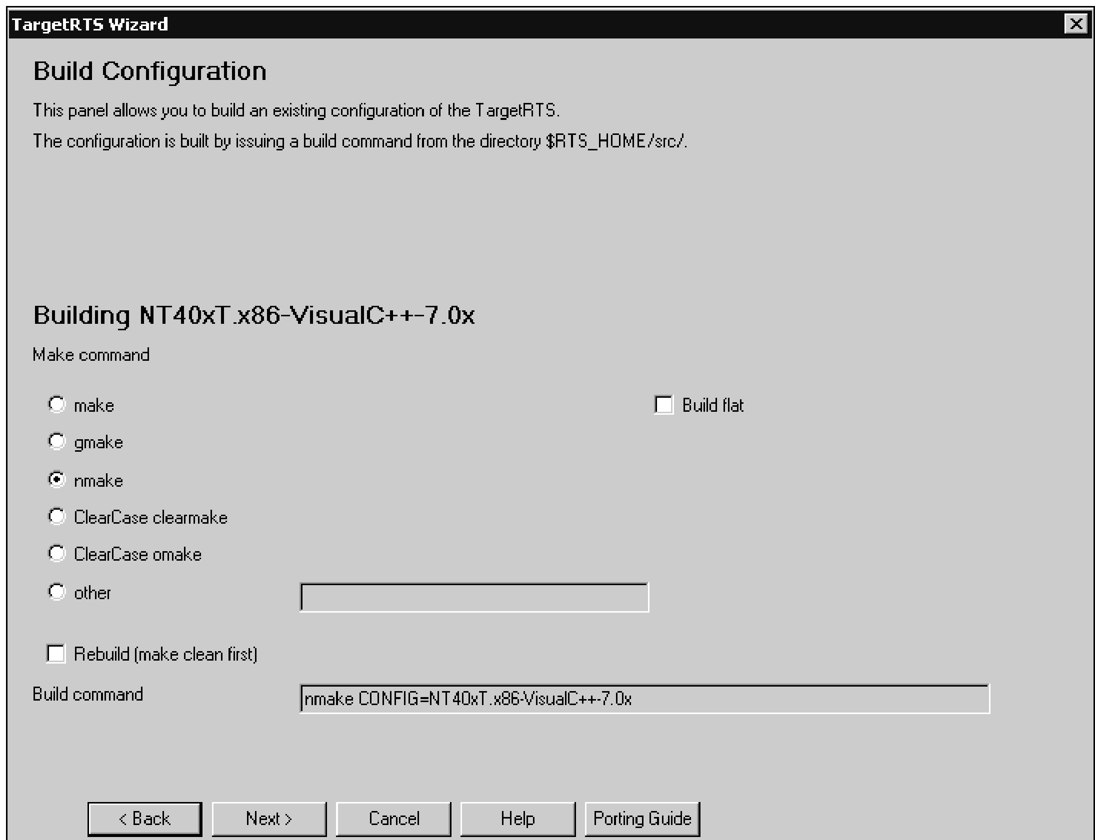
Building a selected configuration creates a directory with the following format:

```
$ROSERT_HOME/build-<target>-<libset>
```

This directory contains the dependency file and object files for the TargetRTS. When the build completes successfully, the resulting Rational Rose RealTime libraries save to a directory that uses the following format:

```
$ROSERT_HOME/lib/<target>.<libset>
```

Figure 34 TargetRTS Wizard - Build Configuration panel



Descriptions

make

Specifies a UNIX implementation of a **make** utility (**make**).

gmake

Specifies the GNU implementation of **make**.

nmake

Specifies a Microsoft implementation of a **make** utility (**nmake**).

ClearCase clearmake

Specifies the UNIX implementation of a **make** utility for building software whose file are under ClearCase version control.

ClearCase omake

Specifies the Windows implementation of a **make** utility for building software whose files are under ClearCase version control.

other

Specifies a alternate **make** utility to build the TargetRTS.

Rebuild (make clean first)

Ensures a clean build. When selected, all intermediate files are deleted first.

Build flat

Copies all source files into a single directory (one file per class) and builds the libraries from that location. This option is useful for debugging because some debuggers do not work properly with the TargetRTS source directory structure.

Note: Setting this option also decreases the build time considerably because fewer source files need to be opened and closed.

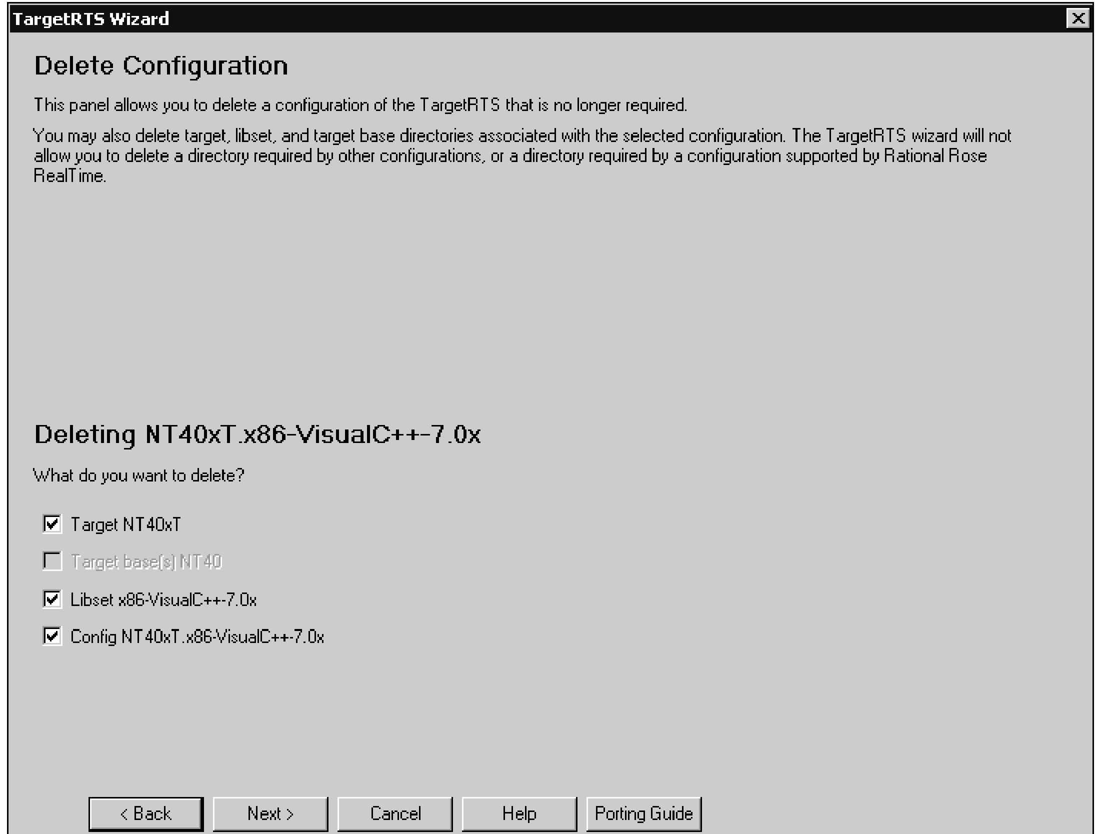
Deleting Configurations

For any duplicated configuration that you create, you can also delete those configurations.

Note: The configurations distributed with Rational Rose RealTime are read-only and cannot be deleted.

Figure 35 shows the **Delete Configuration** panel from which you can selectively delete the target, target base, **libset**, or the configuration-specific files for the selected configuration.

Figure 35 TargetRTS Wizard - Delete Configuration panel



Creating Ports Between C and C++

There is no automatic method of creating a C TargetRTS port form an existing C++ port to the same, or similar OS. You can use the existing port to identify how the OS-specific parts of the TargetRTS were implemented for the particular target. Because the C TargetRTS and C++ TargetRTS have a similar structure, this can save you time.

To make a C TargetRTS port based on a C++ port for the same, or similar OS, you need to:

- create a directory structure for the new port
- configure the new port for the intended toolchain
- configure the OS-specific parts of the port
- build the toolset

Note: The process of creating a C++ port from a C port is similar.

To create the directory structure for the new port:

1 Click **Tools > TargetRTS Wizard**.

2 Specify a language for the new port.

3 Verify that the path to the TargetRTS is correct, and click **Next**.

4 In the **Manage Configurations** panel, select a **NoRTOS** configuration from the **Existing Configurations** list.

5 Click **Duplicate**.

6 Click **Next**.

7 Create a port called:

<*new_target*>`S.`<*new_libset*>

where:

new_target is the name of the OS followed by its version.

Select **Target** and specify a name in the **Target name** box.

new_libset consists of the following format:

<*CPU_name*>-<*compiler_name*>-<*compiler_version*>

Select **Libset** and specify a name in the **Libset name** box.

Note: The "**S**" after the target name denotes a single-threaded configuration; the **TargetRTS Wizard** does not allow the creation of multi-threaded targets from single-threaded ones.

8 In the **Target base** box, select either **Provide skeleton** or **Duplicate** (depending on your preferences).

9 In the **Name** box, specify a name for the target base.

Typically, the name is the name of the OS.

After the duplication process completes, you want to configure the new port for the intended toolchain.

To configure the new port for the toolchain:

1 In the **Manage Configurations** panel, select the new configuration.

2 Click **Edit**.

3 In the **Edit Configuration** pane, select the options to edit the **libset** and the **configuration**.

4 In the following panels, change the values as appropriate for the new toolchain.

5 You may have to edit the $ROSERT_HOME/libset/<new_libset>/libset.mk file to finish configuring the toolchain.

Note: You may have to create a file called $ROSERT_HOME/libset/RTLibSet.h to define compiler-specific macros.

To configure the OS-specific parts of the port:

1 Because the **TargetRTS Wizard** does not permit the creation of a multi-threaded target from a single-threaded one, if the final port is for a multi-threaded environment, change the name of the following directory from:

$ROSERT_HOME/target/<*new_target*>S

to

$ROSERT_HOME/<*new_target*>T

and change the name of the following directory from:

$ROSERT_HOME/config/<*new_target*>S.<*new_libset*>

to

$ROSERT_HOME/config/<*new_target*>T.<*new_libset*>

2 To properly configure the $ROSERT_HOME//target directory, use the contents of the file $ROSERT_HOME/target/<*old_target*>T in the original port's TargetRTS to determine what the $ROSERT_HOME//target/<*new_target*>T file in the new port's TargetRTS should be.

Note: Some configuration macros are not the same in C and C++. However, all of the options are described in the file $ROSERT_HOME/include/RTPublic/Config.h. Also, you will want to review the contents of the file $ROSERT_HOME/target/<*new_target*>T/target.mk. If the new target is multi-threaded, the file $ROSERT_HOME/target/<new_target>T/RTTarget.h will require the **USE_THREADS** macro set to 1, and must also define default priorities for the main, debugger, and timer threads. Typically, you can obtain these values from the original port.

3 Some ports also require configuration-specific settings. These are defined in the file $ROSERT_HOME/config/<*new_target*>T.<*new_libset*>/config.mk. The file $ROSERT_HOME/config/<*new_target*>T.<new_libset>/setup.pl controls the environment configuration required for building the TargetRTS libraries (and possibly, the building of models) for the new platform. The setup.pl file from the old port may provide you with some assistance, but you will have to use your OS and compiler documentation to properly configure the environment.

4 You must write the OS-specific code for the new port. All such code resides in the following directory:

   $ROSERT_HOME/src/target/<*new_target_base*>/

   where:

   new_target_base is the name assigned to the target base during the duplication process in the **TargetRTS Wizard**. This name is stored in the setup.pl script as a value of the **$target_base** variable. The skeleton implementation contains only stubs for functions necessary for all ports. This particular port will likely require you to define additional OS-specific functions. Use the target base from the original port to see how to implement these OS-specific functions. Almost every C TargetRTS "class" has a corresponding class in the C++ TargetRTS.

   Note: Header files in the C target base must be in the RTPubl or RTPriv directories. Also, if some files appear only in this target base, they must be declared in the RTPriv/TGTMFEST.c file in the same manner as other files are declared in the file src/MANIFEST.c.

Note: It may be necessary to further configure that target, **libset**, or **config** settings.

After you finish configuring the target, **libset**, and **config**, you are ready to build the TargetRTS.

To build the TargetRTS:

1 In the **Manage Configurations** panel, select the new configuration in the **Existing Configurations** list.

2 Click **Build**.

3 Specify a **make** utility and click **Next**.

4 Fix any errors encountered during the build process until the TargetRTS successfully builds, and the models link and run.

You may want to create Perl scripts for error parsing in the directory:

$RTS_HOME/codegen/compiler/<*vendor_name*>

<*vendor_name*> is defined in the $RTS_HOMElibset/<*new_libset*>/libset.mk file as a value of VENDOR.

Index

Symbols

#define 40, 45

A

Add Class Dependencies Add-In 13, 14
- considerations 18
- enhancements 14

Add External Java Tool 20
add-in
- Add Class Dependencies 13, 14
- automating common source control tasks 26
- generate documentation 24

adding
- class files 20
- dependencies 19
- external classes 20
- missing dependencies 19

aggregation
- association class 40
- end types 31
- multiplicity 32
- scope 39
- tool for creating 29
- visibility 39

Aggregation Tool 29, 34
AR_CMD 96
AssociationEndKind 33
attribute
- #define 40, 45
- class scope 45
- const 40, 45
- constant 40, 45
- Generate Documentation Add-in 25
- get method 40, 46
- global 45
- implementation 44
- initial value 44
- initialization code 40, 46
- name 43
- private 44
- protected 44
- public 44
- set method 40, 46
- tool for creating 42
- transient 39, 45
- type 43
- visibility 44

Attribute Tool 42
- dependencies 56
- Properties Tab 42

B

build
- configurations 97

Build flat 99

C

class scope
- operation 51

class scope (attribute) 45
classes
- Add Class Dependencies Add-In 14

ClearCase
- clearmake 99
- omake 99

clearmake 99
Compiler (CC flag) 96
component
- tool for creating 75

Component Wizard 75
configuration
- building 97
- deleting 99
- duplicating 86
- editing 90, 96

managing 86
types 87
configuring
RequisitePro for Traceability 61
const 40, 45
constant
attribute 40, 45
contacting Rational customer support xi
creating
ports between C and C++ 100
customized menu commands 67

D

Default arguments 79
deleting
configurations 99
dependencies
adding missing dependencies 19
Attribute Tool 56
configuring search for missing dependencies 19
creating 58
header file 57
implementation 57
matching Classes 57
Operation Tool 56
removing 58
required 57
specifying creation of 20
duplicating
configurations 86

E

editing
configuration 96
configurations 90
libset 95
target 92
EndA 33
EndB 33
Extra Compiler Flags 96

F

files
make Read only 28
make writable 27

G

get method
attribute 40, 46
gmake 98

I

Implementation 39
implementation
attribute 44
Initial Value 38
initial value (attribute) 44
Initialization 39
initialization code 40, 46
Inner Classes 23

J

JAR Utility 24

L

libraries
maintaining TargetRTS 83
Library Builder 96
libset
Compiler Flags 96
defined 87
editing 95
Linker Flags 96
name 89
LIBSETCCEXTRA 96
LIBSETCCFLAGS 96
LIBSETLDFLAGS 96
Linker (LD) 96

M

make 92, 98
makefiles 91
multiplicity
  aggregation association 32
multi-threaded configuration 88

N

nmake 92, 98
NoRTOS 89
NoRTOS target base 90

O

omake 99
operation
  abstract 52
  class scope 51
  final 51
  friend 51
  global 51
  implementation 52
  name 50
  native 52
  polymorphic 53
  private 52
  protected 52
  public 52
  query 51
  return type 51
  strictfp 51
  visibility 52
Operation Tool 49
  dependencies 56
  Properties Tab 50
operations
  Generate Documentation Add-in 25

P

polymorphic
  operation 53
ports
  creating between C and C++ 100
private
  attribute 44
  operation 52
  visibility 39
protected
  attribute 44
  operation 52
  visibility 39
public
  attribute 44
  operation 52
  visibility 39

Q

query
  operation 51

R

Rational customer support
  contacting xi
Rebuild (make clean first) 99
removing
  dependencies 58
report
  options (Generate Documentation Add-in) 25
  type (Generate Documentation Add-in) 25
required dependencies 57
RequisitePro
  configuring for Traceability 61
RoseRTItemID 64
RoseRTModelPath 64
RoseRTType 64
RTS_config 88

S

scripts
  automating source control tasks 26
set method
  attribute 40, 46
single-threaded configuration 88
source control
  automating common tasks 26

T

Target 87
Target bases 88
Target Compiler Flags 94
Target Linker Flags 94
Target name
  definition 88
Target Settings 93
TARGETCCFLAGS 94
TARGETLDFLAGS 94
TargetRTS
  building configurations 97
  configuration types 87
  creating ports between C and C++ 100
  deleting configurations 99
  description 83
  duplicating a configuration 86
  editing a configuration 90, 96
  editing the libset 95
  editing the target 92
  existing configurations 85
  libset 87
  libset name 89
  maintaining libraries 83
  makefiles 91
  managing configurations 86
  NoRTOS Target Base 90
  Summary 89
  target 87
  Target bases 88
  target name 88
  Work Order 89
TEMP 23
Tools
  Add External Java 20
Trace Tool 60
  associating a model to a project 68
  configuring RequisitePro for traceability 61
  customized menu commands 67
  RequisitePro
    adding customized menu commands 67
    DESIGN requirement 70
    requirement type 61
    RoseRTItemID 64
    RoseRTModelPath 64
    RoseRTType 64
    updating RequisitePro requirements 73
  retrieving data from an existing model element 71
  synchronizeing RequisitePro data with a Rose RealTime model 72
transient (Java) 39, 45
troubleshooting
  configuring search for missing dependencies 19

U

USE_THREADS 102

V

visibility 22
  aggregation 39
  operations 52
visibility (types) 44, 52

W

wizard
  component 75
wizards
  Aggregation Tool 29
  Attribute Tool 42
  Component 75
  Operation Tool 49