

Guide to Team Development

RATIONAL ROSE®

VERSION: 2002.05.00

PART NUMBER: 800-025096-000

WINDOWS/UNIX

IMPORTANT NOTICE

COPYRIGHT

Copyright ©1993-2001, Rational Software Corporation. All rights reserved.

Part Number: 800-025096-000

Version Number: 2002.05.00

PERMITTED USAGE

THIS DOCUMENT CONTAINS PROPRIETARY INFORMATION WHICH IS THE PROPERTY OF RATIONAL SOFTWARE CORPORATION (“RATIONAL”) AND IS FURNISHED FOR THE SOLE PURPOSE OF THE OPERATION AND THE MAINTENANCE OF PRODUCTS OF RATIONAL. NO PART OF THIS PUBLICATION IS TO BE USED FOR ANY OTHER PURPOSE, AND IS NOT TO BE REPRODUCED, COPIED, ADAPTED, DISCLOSED, DISTRIBUTED, TRANSMITTED, STORED IN A RETRIEVAL SYSTEM OR TRANSLATED INTO ANY HUMAN OR COMPUTER LANGUAGE, IN ANY FORM, BY ANY MEANS, IN WHOLE OR IN PART, WITHOUT THE PRIOR EXPRESS WRITTEN CONSENT OF RATIONAL.

TRADEMARKS

Rational, Rational Software Corporation, Rational the e-development company, ClearCase, ClearCase Attache, ClearCase MultiSite, ClearDDTS, ClearQuest, ClearQuest MultiSite, DDTS, Object Testing, Object-Oriented Recording, ObjecTime & Design, Objectory, PerformanceStudio, ProjectConsole, PureCoverage, PureDDTS, PureLink, Purify, Purify'd, Quantify, Rational, Rational Apex, Rational CRC, Rational Rose, Rational Suite, Rational Summit, Rational Visual Test, Requisite, RequisitePro, RUP, SiteCheck, SoDA, TestFactory, TestFoundation, TestMate, The Rational Watch, AnalystStudio, ClearGuide, ClearTrack, Connexis, e-Development Accelerators, ObjecTime, Rational Dashboard, Rational PerformanceArchitect, Rational Process Workbench, Rational Suite AnalystStudio, Rational Suite ContentStudio, Rational Suite Enterprise, Rational Suite ManagerStudio, Rational Unified Process, SiteLoad, TestStudio, VADS, among others, are either 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.

Microsoft, the Microsoft logo, Active Accessibility, Active Channel, Active Client, Active Desktop, Active Directory, ActiveMovie, Active Platform, ActiveStore, ActiveSync, ActiveX, Ask Maxwell, Authenticode, AutoSum, BackOffice, the BackOffice logo, BizTalk, Bookshelf, Chromeffects, Clearlead, ClearType, CodeView, Computing Central, DataTips, Developer Studio, Direct3D, DirectAnimation, DirectDraw, DirectInput, DirectMusic, DirectPlay, DirectShow, DirectSound, DirectX, DirectXJ, DoubleSpace, DriveSpace, FoxPro, FrontPage, Funstone, IntelliEye, the

IntelliEye logo, IntelliMirror, IntelliSense, J/Direct, JScript, LineShare, Liquid Motion, 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, Natural, NetMeeting, NetShow, the Office logo, One Thumb, OpenType, Outlook, PhotoDraw, PivotChart, PivotTable, PowerPoint, QuickAssembler, QuickShelf, Realimation, RelayOne, Rushmore, SourceSafe, TipWizard, TrueImage, TutorAssist, V-Chat, VideoFlash, Virtual Basic, the Virtual Basic logo, 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 trademarks or registered trademarks of Microsoft Corporation in the United States and other countries.

FLEXIm and GLOBEtrotter are trademarks or registered trademarks of GLOBEtrotter Software, Inc. Licensee shall not incorporate any GLOBEtrotter software (FLEXIm libraries and utilities) into any product or application the primary purpose of which is software license management.

Portions Copyright ©1992-20xx, Summit Software Company. All rights reserved.

PATENT

U.S. Patent Nos. 5,193,180 and 5,335,344 and 5,535,329 and 5,835,701. Additional patents pending.

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

GOVERNMENT RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in the applicable Rational Software Corporation license agreement and as provided in DFARS 277.7202-1(a) and 277.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii) (Oct. 1988), FAR 12.212(a) (1995), FAR 52.227-19, or FAR 227-14, as applicable.

WARRANTY DISCLAIMER

This document and its associated software may be used as stated in the underlying license agreement. 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 or fitness for a particular purpose or arising from a course of dealing, usage, or trade practice.

Contents

Preface	xv
Audience	xv
Using this Guide	xv
Other Resources	xvi
What to Read Next	xvi
Limitations of this Document	xvii
Contacting Rational Technical Publications	xvii
Contacting Rational Technical Support	xvii
1 Understanding Team Development	1
Contents	1
Planning for Team Development	1
Developing a Strategy	1
How Rational Rose Supports Team Development	2
2 Establishing a Model Architecture and Process for Team Development	5
Model Architecture and Process	5
Establishing Roles and Responsibilities	6
Model Architect	7
Model Manager	7
Modeler/Developer	8
Model Integrator	8
Source Control Administrators	9
Configuration Managers	10
Developing a Model Architecture	10
Understanding Subsystems	10
One Model Versus Multiple Models	12
Mapping the Architecture to Subsystems	12
Checking Package Dependencies for Completeness	13
Checking if a Subsystem is Self-Contained	14
Defining Subsystem Interfaces	15
Configuring Subsystem Components	15
Providing Support for Unit Testing	17
Using Property Sets for Build Settings	17
Creating Processors and Component Instances	17

Preparing and Releasing Subsystems	19
Splitting a Model into Subsystem Models	20
Splitting a Model Under Version Control	23
Managing/Administering a Model	26
Configuring Compatible Workspaces	26
Configuring a Version Control System and Repository	27
Partitioning the Model into Controlled Units	27
Save Model to Local Work Area	28
Adding the Model to Version Control	28
Defining Developer Work Areas	28
Creating Labels and Lineups	28
Manipulating the Version Control Repository	28
Developing/Implementing a Model	29
Setting up Version Control	29
Setting up Developer Work Areas	29
Getting a Specific Lineup of a Model	29
Opening a Model Under Version Control	29
Working under Version Control	30
Comparing and Merging Model Elements	30
Promoting Changes for Integration	30
Integrating Changes	30
Automating Model Validation	30
3 Best Practices	33
Contents	33
Goals of Team Development	33
Sharing Within a Team Environment	34
Protecting Configuration Items From Unintentional Changes	35
Overwriting a Modification	36
Adding Dependency Issues	38
Managing Relationships Between Configuration Items	40
Managing and Delivering Configuration Items	41
Improving Efficiency in Team Development	43
Model Architect Role	43
Recommendations	44
Source Control Fundamentals	44
Preempting Conflicts	46
Managing Dependencies	46

Labeling	47
When Merging is Necessary	48
Advanced Concepts and Heuristics	48
Moving Controlled Units	48
Parallel Development	50
Model Integrator	51
Using Rational ClearCase Multi-Site	52
Additional Heuristics for Team Development	52
4 Dividing a Model into Controlled Units	55
What is a Controlled Unit?	55
What Can be a Controlled Unit	56
How Controlled Units are Related and What They Contain	57
Working with Controlled Units	59
Creating Controlled Units	59
Loading, Reloading, and Unloading Controlled Units	59
Creating and Using Model Workspaces	61
Protecting Controlled Units	64
Splitting a Controlled Unit	65
Merging Controlled Units	66
Adding Controlled Units to a Model (Importing/Loading)	66
Uncontrolling Controlled Units	66
Creating Virtual Paths to Controlled Units	67
Understanding Virtual Path Maps	67
How Virtual Paths Work	68
Creating Virtual Path Maps	69
Defining a Path Map Relative to the Location of the Model File	70
Defining a New Path Map Using Another Path Map Symbol	70
Defining a Path Map with Wildcards	70
Using Virtual Paths for the Value of a Model Property	71
Using Path Maps for Other Artifacts	71
Where Virtual Path Maps are Stored	72
Checking References and Access Violations	72
Check Model	72
Show Access Violations	73
Organizing Controlled Units for Teams	74
Suggested Strategies	74

5 Comparing and Merging Models	77
Contents	77
About the Model Integrator	77
Model Integrator Interface	78
Contributors	80
Base Model	80
Comparing Models	80
Merging Models	80
Differences and Conflicts	81
Model Files and Model Integrator	82
Understanding Semantic Checking	84
Memory Requirements and Performance	85
Model Integrator and ClearCase	86
Merging Whole Models with Controlled Subunits	87
Starting Model Integrator in a ClearCase Integration	87
Comparing and Merging Models	87
Starting Model Integrator	87
Preparing Models for Merging	88
Selecting the Contributors	88
Loading or Unloading Controlled Units	89
Using Compare Mode	92
Using Merge Mode	92
Interpreting Compare and Merge Results	94
Navigating Through a Model	95
Accepting Changes from Contributors	99
Changing Nodes with Differences	100
Reversing Changes to Nodes	101
Using Subtree Mode	101
Using Semantic Checking	102
Checking Merged Model for Consistency	102
Correcting Merge Errors	103
Saving Results	104
Performing a Partial Merge	105
Merging Models Without a Base Model	106
Viewing a Single Model File	107
Using Model Integrator from the Command Line	107

6 Working with a Version Control System	109
Understanding Version Control	109
Types of Version Control Systems	110
Version Control Development Concepts	110
Versioning Strategies	112
Rational Rose Integration with Version Control Systems	114
Version Control Add-In	114
ClearCase Add-In	115
Choosing and Activating a Version Control Add-In	115
Using Rational ClearCase	116
About ClearCase	116
Versioned Object Bases (VOBs)	116
ClearCase Views	117
Configuring ClearCase for Rational Rose	118
Using Microsoft Visual SourceSafe	119
Configuring Microsoft Visual SourceSafe for Rational Rose	119
Using Version Control Features From Rational Rose	120
Using the Version Control Add-In on a Previously Controlled Model	120
Adding Controlled Units to Version Control	121
Checking in Controlled Units	122
Checking Out Controlled Units	123
Undoing the Check-Out of Controlled Units	124
Getting the Latest Version of Controlled Units	124
Removing Controlled Units from Version Control	125
Index	127

Figures

Figure 1	Architect Role in Team Development	7
Figure 2	Manager Role in Team Development	8
Figure 3	Modeler/Developer Role in Team Development	8
Figure 4	Integrator Role in Team Development	9
Figure 5	Subsystems	11
Figure 6	Component Diagram for a Sample Subsystem	16
Figure 7	Overwriting a modification	36
Figure 8	Check-out and Check-in Scenario	37
Figure 9	Checking Out an Artifact After it is Checked In	37
Figure 10	Merging Changes Prior to Check-In	38
Figure 11	Comparison Between Versions	38
Figure 12	Removing Required Dependencies	39
Figure 13	Comparing Dependency Reports	40
Figure 14	Labelling Configuration Items	41
Figure 15	Example of Labelling Items	42
Figure 16	Comparing Reports	43
Figure 17	Parallel Stream Versioning Strategy	45
Figure 18	Controlled Units	56
Figure 19	View from which you cannot create Controlled Units	56
Figure 20	Example of a Controlled Unit Hierarchy	57
Figure 21	Controlled Unit Hierarchies in the Rational Rose Browser	58
Figure 22	Controlled Unit File Name	60
Figure 23	Loaded and Unloaded Controlled Unit Icons	60
Figure 24	Adornments Indicating Controlled Units and Unresolved References	61
Figure 25	Example Model Workspace	63
Figure 26	Write-Protected Control Unit	64
Figure 27	Virtual Path Maps	68
Figure 28	Virtual Path Map Dialog Box	69
Figure 29	Model Integrator Graphical User Interface	78
Figure 30	Subunits Dialog Box	89
Figure 31	Property View of a TransView Object	98
Figure 32	Model Integrator Window	99
Figure 33	Parallel Stream Versioning	113
Figure 34	Version Controlled Object (VOB) Tree Structure	117

Tables

Table 1	Image Legend	36
Table 2	AutoMerge Rules for Merging Models	93
Table 3	Compare Status Icons	94
Table 4	Merge Status Icons	95
Table 5	Navigation Buttons for Viewing Conflicts and Differences	96

Preface

This book provides an overview of the basic team development concepts in Rational Rose, and shows how to set up and use Rational Rose in a team environment.

Audience

This manual is intended for:

- Users who work in or support teams of modelers/developers.
- Database developers and administrators.
- Software system architects.
- Software engineers and programmers.
- Anyone who makes design, architecture, configuration management, and testing decisions.

This manual assumes you are familiar with a high-level language and the life-cycle of a software development project.

Using this Guide

This guide provides an overview of the basic team development concepts in Rational Rose, as well as how to set up and use Rational Rose in a team environment.

The information in this book spans other product lines, including software from other vendors. Its primary goal is to help you develop and tailor your own guidelines.

While this book provides explanations of some features, you will need to rely on additional product libraries for information. For example, you may need to refer to the ClearCase documentation to configure ClearCase for your environment.

In addition to this guide, refer to the Rational web site (www.rational.com) for white papers, technical notes, and articles relating to team development.

Other Resources

- For more information on training opportunities, see the Rational University Web site at <http://www.rational.com/university>.
- The information in this guide spans numerous products, both from Rational and from other software vendors. To learn more about these products, consult the product's documentation. The Rational Rose Guide to Using Rose and the Rational Rose online Help provide detailed information about Rose models and the Model Integrator.

What to Read Next

All users of Rational Rose should familiarize themselves with how models are stored and how Rational Rose interacts with source control systems. See “Managing/Administering a Model” on page 26 and “Source Control Fundamentals” on page 44 for details on these areas.

Developers should also read “Working with a Version Control System” on page 109. Project leads and architects will want to review the material in “Developing a Model Architecture” on page 10.

Development environment infrastructure is discussed in “Working with a Version Control System” on page 109. That section should be read by source control administrators and anyone involved with preparing builds. As well, specific details regarding the creation of a build process and automating builds, as well as how to perform model integration, is covered in “Establishing a Model Architecture and Process for Team Development” on page 5.

Limitations of this Document

One of the primary goals of Rational Rose is to fit into your existing development and build processes. While this document tries to be thorough, you will encounter team development issues which are not covered here. As a general guideline, you should approach these situations with the same mindset you would have for 'more typical' C++, C, and Java development. In most cases, you can apply the same, or slightly modified practices to achieve your goals.

Contacting Rational Technical Publications

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

Contacting Rational Technical Support

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

Your Location	Telephone	Fax	E-mail
North America	(800) 433-5444 (toll free) (408) 863-4000 Cupertino, CA	(781) 676-2460 Lexington, MA	support@rational.com
Europe, Middle East, Africa	+31 (0) 20-4546-200 Netherlands	+31 (0) 20-4546-202 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 Technical Support, please be prepared to supply the following information:

- Your name, telephone number, and company name
- Your computer's make and model
- Your operating system and version number
- Product release number and serial number
- Your case ID number (if you are following up on a previously-reported problem)

Understanding Team Development

1

Contents

The chapter is organized as follows:

- *Planning for Team Development* on page 1
- *How Rational Rose Supports Team Development* on page 2

Planning for Team Development

Developing complex systems requires that groups of analysts, architects, and developers be able to see and access the “big picture” while working on their own portion of that picture. Successfully managing an environment where multiple team members have different kinds of access to the same model requires:

- Formulating a working strategy for managing team activity.
- Having the tools to support that strategy.

Developing a Strategy

When working in teams, you need to develop strategies for:

- Supporting current development.
- Maintaining and retrieving the reusable modeling artifacts that result from development.

Current Projects

When developing current projects, the tools that a team uses must be able to:

- Provide all team members with simultaneous access to the entire model.
- Control which team members can update different model elements.
- Introduce change in a controlled manner.
- Maintain multiple versions of a model.

Implementing a configuration management or version control system is essential for complex projects. A configuration management system can effectively support team development as long as it:

- Protects developers from unapproved model changes.
- Supports comparing and merging all changes made by multiple contributors.
- Supports distributed (geographically dispersed) development.

Developing for Reuse

When you develop a system, you develop valuable project artifacts that can be reused. Artifacts are typically maintained in some type of repository. To support reuse:

- Model artifacts should be architecturally significant units, such as patterns, frameworks, and components (not usually individual classes).
- All members of a team, no matter where they are located, should have access to reusable artifacts.
- It should be easy to catalog, find, and then apply these artifacts in a model.

A reuse repository can differ from your project's configuration management system as long as it supports versioning. Versioning is a process of tracking a file's history from the initial version to the current version.

The repository should also support cataloging artifacts at an appropriate level of granularity, for example, at the component level.

How Rational Rose Supports Team Development

To support teams of analysts, architects, and software developers, Rational Rose:

- Allows team development of a shared model by supporting decomposition of the model into versionable units, called controlled units.
- Permits model files and controlled units to be moved or copied among work areas by using the virtual path map mechanism.
- Enables teams to manage their model in concert with other project artifacts by integrating with standard source control systems.
- Provides a separate tool, called Model Integrator, to compare and merge controlled units.
- Enables teams to build their models in concert with other project artifacts by integrating with standard build environments.

Since managing parallel development is so crucial, Rational Rose provides integrations with Rational ClearCase and with SCC-compliant version control systems, such as Microsoft Visual SourceSafe. By integrating configuration

management systems, Rational Rose makes frequently used version control commands directly accessible from the Rational Rose menus, such as check in and check out functions.

Establishing a Model Architecture and Process for Team Development

2

The following topics are covered in this section:

- *Model Architecture and Process* on page 5
- *Establishing Roles and Responsibilities* on page 6
- *Developing a Model Architecture* on page 10
- *Managing/Administering a Model* on page 26
- *Developing/Implementing a Model* on page 29

Model Architecture and Process

Chapters 4, 5, and 6 of this guide describe fundamental concepts about models, how they are stored, and the tools that you use to manage them. As essential as this information is, it is probably even more important that you and your team develop and implement a sound architecture for layering and partitioning your Rational Rose models, as well as defining a process for managing your model and related artifacts throughout the development cycle.

This chapter provides:

- Guidelines for developing a model architecture
- A suggested breakdown of activities and roles associated with the architecture

Note: The Rational Unified Process (RUP) provides detailed information about the overall development process and should be one of your primary resources for implementing team development.

Establishing Roles and Responsibilities

This section provides an overview of the typical development roles played by team members in a software project. The organization of the remaining sections elaborate on the logical activities associated with these roles.

Typical Roles

A role is a named behavior of an entity participating in team development, and each role has assigned tasks to complete. There are typically seven roles to consider in your team environment:

- Model Architect
- Model Manager
- Modeller/Developer
- Model Integrator
- Administrator (for source control)
- Configuration Manager

Roles Vary Based on Team Size

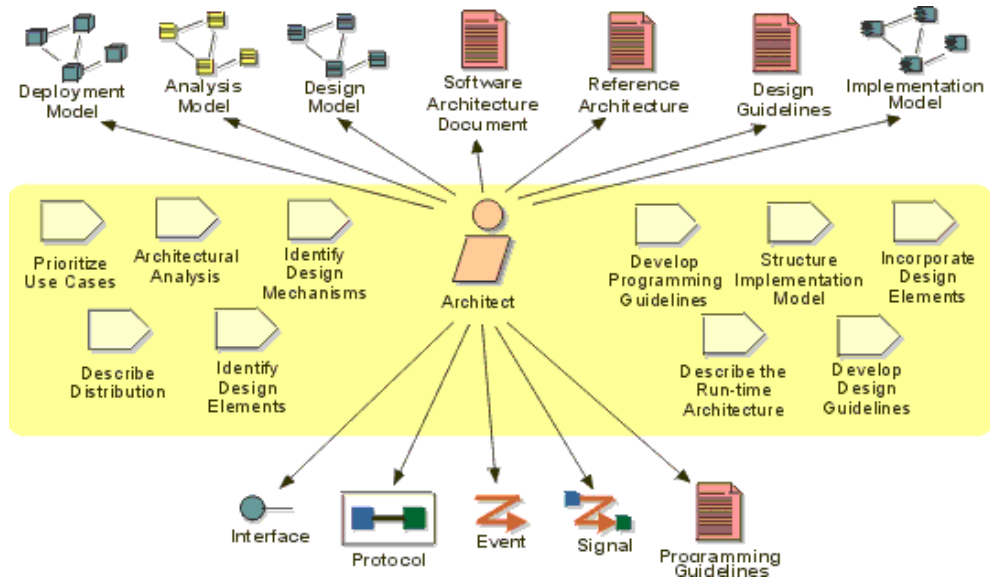
In a large team environment, several people can be responsible for different team tasks associated with the same role, whereas smaller projects can have only one person responsible for most or all of the tasks for a specific role.

A single person can play multiple roles. A user can perform Architect tasks while working on the initial architecture of the system. Later, they can perform Developer tasks when they are performing detailed implementation. After they make changes, the user can perform Integrator tasks to promote this change to the integration branch of their source control system.

Model Architect

The architect role leads and coordinates technical activities and artifacts throughout the project. The Architect establishes the overall structure for each architectural view, including the decomposition of the view, the grouping of elements, and the interfaces between these major groupings.

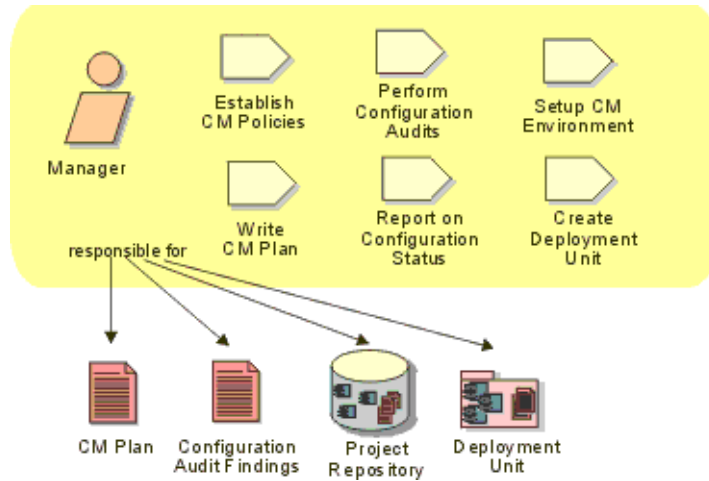
Figure 1 Architect Role in Team Development



Model Manager

The manager provides the overall version control infrastructure and environment to the product development team. The manager function supports the product development activity so that developers and integrators have appropriate workspaces to build and test their work, and so that all artifacts are available for inclusion in the deployment unit as required. The manager also has to ensure that the version control environment facilitates product review, and change and defect tracking activities.

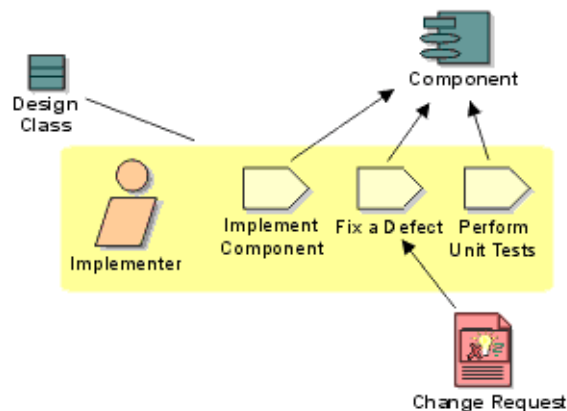
Figure 2 Manager Role in Team Development



Modeler/Developer

The modeler/developer is a collective name that represents people who view or modify Rational Rose models.

Figure 3 Modeler/Developer Role in Team Development

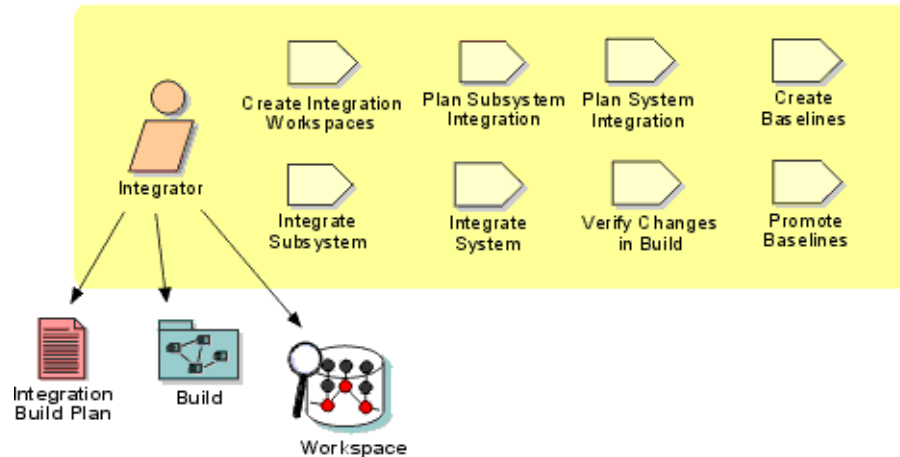


Model Integrator

Developers deliver their tested components into an integration workspace where integrators combine them to produce a build. An integrator is also responsible for planning the integration, which takes place at the subsystem and system levels, with each having a separate integration workspace. Tested components are delivered from

an implementer's private development workspace into a subsystem integration workspace, whereas integrated implementation subsystems are delivered from the subsystem integration workspace into the system integration workspace.

Figure 4 Integrator Role in Team Development



Source Control Administrators

The Source Control Administrator provides the overall source control infrastructure and environment for all required members of the team.

Source Control Administrator Tasks:

- Configuring the source control system for use with Rational Rose
- Placing a model under source control
- Creating a default workspace file
- Defining work areas
- Defining lineup policies
- Enforcing all other configuration management plan policies

Depending on your team organization, the Integrator role can perform one or more of these tasks.

Configuration Managers

The Configuration Manager provides the overall Configuration Management (CM) infrastructure and environment. The CM function supports the product development activity so that developers, integrators have appropriate workspaces to perform work.

The Configuration Manager must ensure that the CM environment facilitates product review, change, and defect tracking activities. The Configuration Manager is ultimately responsible for a comprehensive plan that identifies and deals with pitfalls to team development in the most efficient way for the project.

Developing a Model Architecture

One of the Architect's primary goals is to structure or organize a Rational Rose model so that it can be used effectively by a team.

Product development often starts with a small team working on a single model. As development progresses, the team (and the model) grow to a point where organizing the model appropriately becomes crucial to supporting multiple teams working in parallel.

An Architect also has a profound impact on developing for reuse. You can use Rational Rose to split a model into highly cohesive layers or frameworks that can be reused in multiple models.

The actual division of a model into packages and subsystems is something of an art form and this chapter attempts to describe guidelines to help you get started. Remember that after a model is partitioned into subsystems, you can either work with one model or split the model into separate models for each subsystem.

Understanding Subsystems

Packages group model elements. There are four types of packages in Rational Rose: use case, logical, and component packages. Each type of package can only group certain model elements. For example a logical package can group classes, whereas a component package groups component diagrams and components. Packages can also contain packages of the same kind, so it is possible to decompose your models hierarchically.

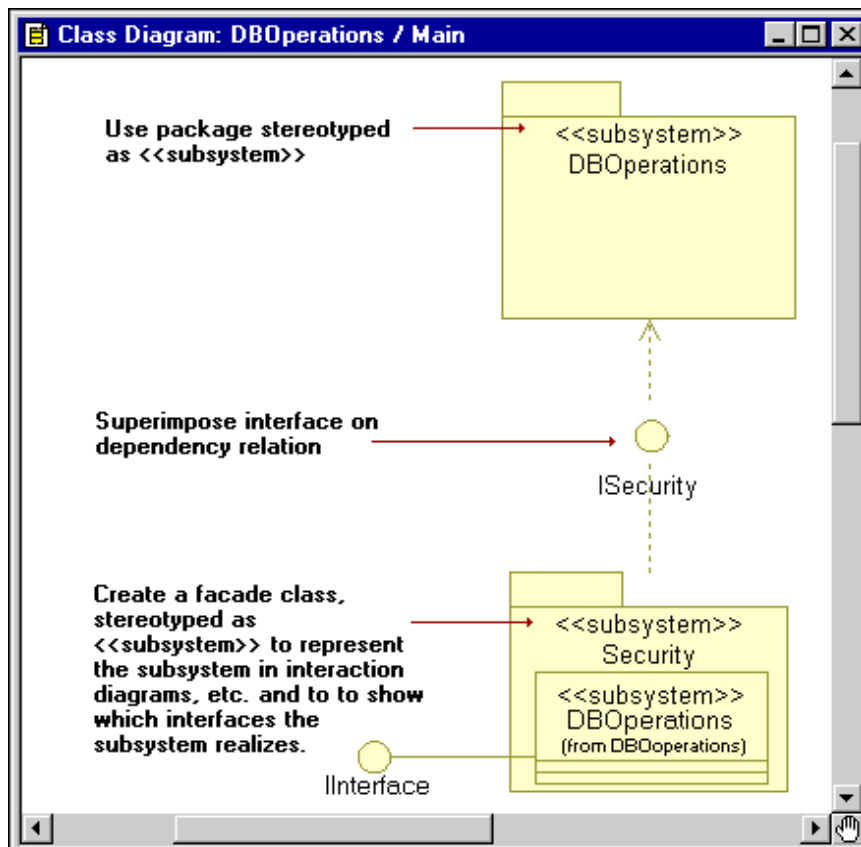
A model is composed of the four root packages: Use Case View, Logical View, Component View, and the Deployment View. The model is the top level model element which contains all sub elements.

A subsystem is a concept and not an explicit modeling element in Rational Rose. The term represents a set of related packages that can be developed, tested, and released together. Subsystems provide strong separation between major portions of your model and form the basis for reuse between models. In a layered development approach, the model for each layer will share in subsystems for the layers beneath it.

A subsystem will typically consist of one or more logical packages and one or more component packages. The logical packages contain the classes in the subsystem and the component packages contain the components that are used to build the subsystem.

By converting packages that provide discrete, well-defined services into subsystems, you are able to control dependencies better. Subsystems expose services only through an interface and subsystem internals should depend only on the interfaces that are offered by other systems.

Figure 5 Subsystems



One Model Versus Multiple Models

A large development project can result in a correspondingly large model for the complete application. If the model has a layered architecture, then it is possible to produce a set of smaller models that follow the layering of the larger model.

One of the goals of having a separate model for each layer/subsystem is to reduce the number of developers working on the same model. This technique helps to isolate development work and reduce parallel development issues.

To build the full project, one designer, typically called the builder, opens a model that references all the subsystems that make up the project, thus loading all the changes made to the packages in the subsystems, then build from that model.

Before splitting a model into a set of subsystem models, you should first consider the trade-offs.

Advantages of a model for each subsystem:

- Improves Rational Rose performance and memory footprint because the model is smaller.
- You can build, test, and release subsystems separately, reducing system complexity.
- Groups can share subsystems. Teams can share stable versions of subsystems.

Disadvantages:

- Can be more complicated to set up.
- Build process can be more involved.
- Might not be appropriate for small teams.

The following sections describe steps you should perform before splitting a model to ensure that your model is well partitioned.

Mapping the Architecture to Subsystems

You can decompose a model by grouping modeling elements into packages then assign a set of these packages to subsystems.

You should consider each subsystem as a distinct unit that you can build and test independently, whether or not you will split the model. You will also need to define and enforce the interfaces between subsystems.

Tasks for Decomposing a Model into Subsystems

- Checking Package Dependencies for Completeness
- Checking if a Subsystem is Self-contained
- Defining Subsystem Interfaces
- Configuring Subsystem Components
- Providing support for Unit Testing
- Using Property Sets for Build Settings
- Creating Processors and Component Instances
- Preparing and Releasing Subsystems

Tasks for Splitting a Model:

- Splitting a Model not in Version Control
- Splitting a Model Under Version Control

Checking Package Dependencies for Completeness

Developers can define class-level relationships that may violate dependencies between packages and subsystems. After you create packages and move the model elements into the packages (subsystems), ensure that the subsystems you created have the dependencies that you expect. If the dependencies between subsystems are too complex, it may be difficult to work in teams (changes are not isolated) and split the model.

Show Access Violations

Click **Report > Show Access Violations** to verify that the designed dependencies between packages (subsystems) are correct and complete.

The Architect should revisit the package dependencies periodically to check that the detailed implementation has not violated the intended architecture.

Click **Show Access Violations** to verify that there are no violations in the logical packages and component packages in the subsystem. You should also verify that every class and logical package referenced by the components in the subsystem are also part of the subsystem.

Determine the External Dependencies for a Package

The **Specification** dialog box for a package contains a **Relations** tab that shows the dependencies for this package. You can determine if a package has any dependencies, but it can be difficult to visualize the dependencies if you only look at this list. In order to properly visualize the package relationships, use a class diagram.

To create a class diagram showing the relationships for a specific package, follow these steps:

- 1 Open the class diagram.
- 2 If the package is not already on this diagram, then drag it from the browser onto the diagram.
- 3 Click **Query > Add Classes** to add all the classes from a package to a diagram.
- 4 Press CTRL + A to select all of the classes in the diagram, then click **Query > Expand Selected Elements**.
- 5 The resulting dialog allows you to add related elements to this diagram based on the chosen options. To see the direct dependencies for this package, set the options to expand one level of suppliers. Ensure that dependency relations are selected in the **Relations** dialog box.

By varying the options chosen in these dialogs you can produce a diagram showing the desired information. If many packages were added to the diagram, then you may want to use the automatic layout feature to produce an initial layout for the diagram.

By reviewing the relationships in this diagram, the Architect can detect any undesirable dependencies. Resolving an undesirable dependency can involve either modifying the class(es) that caused the violation and/or moving some of these classes to another package.

Checking if a Subsystem is Self-Contained

A self-contained subsystem is composed of packages that do not have any dependencies to packages outside of the subsystem. A self-contained subsystem can be shared without requiring any other subsystems.

Assuming the package dependencies are complete (see *Checking Package Dependencies for Completeness* on page 13), then checking whether a subsystem is self-contained means examining the dependencies for the packages in the subsystem to ensure that all of them are to other packages within the subsystem.

A subsystem does not need to be self-contained in order to be shared, as long as the sharing model contains all the other subsystems that are required.

Defining Subsystem Interfaces

By reducing the coupling between subsystems, you can lessen the chance of having integration problems caused by using subsystems that have complex dependencies.

It is important for the subsystem producer to pay close attention to which classes in a subsystem are public (visible and usable outside of the subsystem) and which are private. It is also recommended that the subsystem contain a set of class diagrams that illustrate the public classes.

Best practices include:

- Specify the visibility of each class (public or implementation).
- Include one or more class diagrams showing the public classes.

You may also use different visual clues (such as color) for the public classes in a class diagram.

Configuring Subsystem Components

Rational Rose can support general types of components such as:

- Executables
- Source code
- Binary code
- dlls

A small model may have a single executable component that is built to produce the application. A large model has an executable component and many library components, typically corresponding to the layering in the architecture.

In addition to the components used to build the complete application, it is useful to have components that build subsets of the model, for example for unit testing purposes.

Components in Subsystems

Ideally, each subsystem contains one or more external library components. These components are built as part of the build process of a subsystem and are referenced in models that use the subsystem.

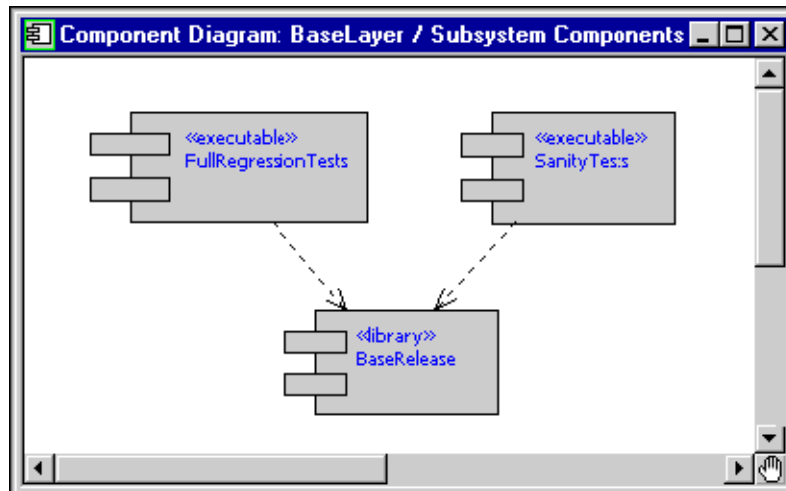
An external library component allows the sharing model to reuse the prebuilt library, which can dramatically reduce build times for a large model.

A subsystem often includes multiple variations of each component, such as a debug component and a release component. For ease of navigation and organization, group the subsystem components into packages (a Debug package and a Release package) containing the debug and the release components.

The subsystem model requires one or more executable components used to test the subsystem. Typically, the executable component only contains the testing classes and has a dependency on the library component for the subsystem.

The following component diagram shows three components for a sample subsystem. The BaseRelease component is a library that contains the subsystem. The SanityTests and FullRegressionTests components are executables that use the BaseRelease component.

Figure 6 Component Diagram for a Sample Subsystem



After you create the necessary components and the dependencies between them, you must determine which classes belong to which components. Typically, this follows from the architecture of the model, although there can be issues that arise during development. As new classes are created, they must be added to the appropriate component(s). If multiple developers create classes referenced by the same component, then the component can become a source of contention.

The contention for a component can sometimes be avoided, or at least reduced, when the component references logical packages instead of classes themselves. Referencing a package from a component is equivalent to referencing all the classes in that package.

The added benefit is that the component does not need to be updated when a new class is added to the package as long as that class belongs in that component. The risk is that a component may contain classes that it does not require.

Providing Support for Unit Testing

While working within a subsystem model, a developer may find it useful to create a component for use in unit testing changes. If this component has lasting value, then it should be created as part of the subsystem model so that it can be reused. To support the organized storage of unit testing components, an Architect may find it useful to create component packages that can be used for grouping these components.

If many developers create components in the same package, then this package can become a source of contention. If your development process requires the creation and version control of unit testing components, then you may wish to create several component packages used for this.

Using Property Sets for Build Settings

Using property sets for common build settings is a suggested method for maintaining and reusing project level configuration information for building components.

Tasks

- The builder or architect defines custom sets of component properties that are specific to a project. For example, you can have debug and release build settings. Custom properties are stored in the .prp file for this model.
- A component should be based on the appropriate property sets by modifying the Default set field in appropriate property tabs of the component Specification dialog. Any local overrides should also be added.

Creating Processors and Component Instances

Project Level Processors

For each project, there is usually a known set of processors where component instances execute. Since all the subsystems in the model are intended to execute on this set of processors, these project level processors should be defined in a package shared between the various subsystem models.

The builder should set up a package that contains these project level processors. For example, the builder could configure processors for the labs that are available for the development teams. These package(s) can then be shared in each subsystem model. Each package should be owned by one of the models so that modifications can be made to it in a controlled manner.

The processors in these project level packages will not typically contain any component instances. If they did contain a component instance, then sharing them would also require the corresponding component packages which contain the required components. In turn, these components would require the referenced classes and logical packages. Unless these elements are present in all subsystem models, these processors should only be used as templates in the subsystem models.

Subsystem Level Processors

A development team may choose to create additional processors for their own use, either by copying the project level processors or by creating new processors for platforms that are not shared with other teams.

The subsystem level processors can contain component instances based on the components present in the subsystem. Typically, this includes component instances for regression testing the subsystem and for unit testing major classes in the subsystem.

Component Instances

Component instances indicate the ability to run a specified executable component on a specified processor. A component instance is controlled with the processor. As mentioned above, project level processors usually won't have any component instances so they will be typically copied before they can be used to execute/test a component.

Subsystem level processors will often contain component instances that execute/test the entire subsystem. Developers working on the subsystem can use these component instances but they may find it easier to create specific unit testing components and corresponding component instances.

Tasks

- A set of packages can be created to hold processors that are available in-house for testing. The processors will contain IP addresses, host names, and other configuration information that can be reused by all developers.
- Subsystem processors can be created by copying project level processors and creating the component instances desired for executing/testing the subsystem.

Preparing and Releasing Subsystems

In a model composed of multiple subsystems, there should be policies in place that describe how new versions of the subsystems will be made available to other models.

Subsystem Supplier

When a team is ready to release a new version of a subsystem, they must ensure that the correct version of all the necessary elements of the subsystem are made available. This includes:

- Logical packages containing the classes in the subsystem
- Component packages containing the library components and/or external library components for the subsystem
- Any other required Rational Rose elements
- Any other required external (non-Rose) elements for external library components

The team releasing the subsystem will typically prepare the required elements using one of the following mechanisms:

- Label Subsystem Elements. If the model is under version control, then a label can be applied to the elements in the subsystem.
- Copy Subsystem Elements. The elements in the subsystem can be copied to a known location.

Subsystem Consumer

The architect for a model that requires this subsystem must then ensure that the model includes the new version of the subsystem. The mechanism for this depends on how the subsystem elements were made available.

If the subsystem elements have been copied to a known location, the architect must ensure that this location is referenced by the model. If the location is the same as the previous version of the subsystem, then no changes should be necessary. If the location has changed, then the architect may have to recreate the model by loading the shared packages from the new locations and adding in the packages that are owned by this model.

If the subsystem has been packaged using a version control label, then the architect must ensure that this label is used for getting the new lineup for the model.

If there are changes to the subsystem interface, then the architect of a model which uses this subsystem must ensure that the corresponding changes are made within their model.

Splitting a Model into Subsystem Models

Splitting a large model into smaller subsystem models can improve team development. A developer can then work on the appropriate model for his or her particular subsystem. Working on this smaller model should reduce the Rational Rose footprint and improve the performance of several operations such as opening a model.

It is possible to split a model before or after it has been placed under version control. If a model has not been controlled, we recommend that you split the model first, then add the resulting controlled units to version control.

Before a model is split into subsystem models, you must ensure that the dependencies between the subsystems will support this partitioning. Specifically, you must ensure that the subsystems form a layered architecture that allow each subsystem to exist in a model that does not contain any of the “higher level” subsystems. See *Checking Package Dependencies for Completeness* on page 13 for more information.

Should you Split the Model Before Adding to Version Control?

If your model is not currently in version control, split the model before adding it to version control. If your model is already in version control, you can also split the model into separate models, however, this process is different.

Splitting a Model not in Version Control

At present, we assume that you have a base model (in this example we call it Base) and that the model is not yet in version control. We also assume that you will create separate models for each of your subsystems.

Lastly, this description also assumes that you want to keep the controlled units for each subsystem model together so they can be moved into the subsystem directory tree. Moving the files is optional but it can make it much easier to manage the files that make up each model.

See *Dividing a Model into Controlled Units* on page 55 for more information on loading and importing controlled units.

Tasks

- 1 Ensure that the base model has defined the initial controlled units, at least at the package level corresponding to the subsystem partitioning.

Note: By default, Rational Rose does not put files in a directory hierarchy.

The base model (Base) directory hierarchy for the sample model looks like:

```
Base.mdl
<Base>
  UseCaseView.cat
  <UseCaseView>
<LogicalView>
  SubSystem1.cat
  <SubSystem1> SubSystem2.cat
  <SubSystem2>
ComponentView.sub
<ComponentView>
  SubSystem1.sub
  <SubSystem1>
  SubSystem2.sub
  <SubSystem2>
DeploymentView.prc
```

Click **File > Edit Path Map** to create a Virtual Path Map variable for each top level package in the model (for example, each subsystem package). In our example, we could create path map variables SubSystem1LogicalPkg, SubSystem1ComponentPkg, SubSystem2LogicalPackage, SubSystem2ComponentPkg, and so on.

- 2 Save the Base model units affected by the new path map variable.
- 3 If the Base model makes use of custom property sets, then these must be made available to the subsystem models. Click **Tools > Model Properties > Export** to create a file that can be imported to the subsystem models.
- 4 Create a new model by selecting **File > New**. This model will be used for the first subsystem. Ensure that the path map variables are defined correctly.
- 5 If the Base model makes use of custom property sets, then ensure that these are available in the subsystem model. Click **Tools > Model Properties > Replace** to import the file containing the property sets.
- 6 Control all the elements in the new model by clicking **File > Units > Control**.
- 7 Save the model (.mdl file) into an appropriate directory by clicking **File > Save As**. We recommend that you create a dedicated directory for each subsystem. For example, name the subsystem model SubSystem1 and store it in a directory called SubSystem1.

- 8 Next, you can optionally move the packages that make up your subsystem from the base model directory hierarchy into the subsystem model directory hierarchy that was created when you saved the new model.

For each package that will be part of the subsystem, move the package controlled units into the corresponding directory level in the new model, and then move the directories for each package to the corresponding location. The resulting directory hierarchy for the new model looks like:

```
SubSystem1.mdl
  <SubSystem1>
    UseCaseView.cat
    <UseCaseView>
  <LogicalView>
    SubSystem1.cat
    <SubSystem1>
  ComponentView.sub
  <ComponentView>
    SubSystem1.sub
    <SubSystem1>
  DeploymentView.prc
```

If you move the files, edit the associated path maps to reflect the new file locations.

- 9 Add the subsystem packages into the subsystem model by clicking **File > Units > Load**. These packages should be added in at the same location in the subsystem model hierarchy as they were in the base model. In our example, SubSystem1.cat should be added to the Logical View and SubSystem1.sub should be added to the Component View.
- 10 Save the subsystem model.

Repeat steps 5 - 11 for each remaining subsystem with the following addition:

Before adding the subsystem packages to the new subsystem model (Step 7), you must load the packages from the other subsystems that are required by this subsystem.

After splitting the original model, you will typically not use that model for any further development. You may choose to create an equivalent model that shares in all the subsystems. For example, create a new model called NewBase that shares in the packages in SubSystem1 and SubSystem2. This model cannot be used to edit any of the subsystems, but it might be useful for building and/or testing.

Splitting a Model Under Version Control

At present, we assume that you have a base model (in this example we call it Base) and that the model is under version control. We also assume that you create separate models for each of your subsystems.

Lastly, this description also assumes that you want to keep the controlled units for each subsystem model together and so they will be moved into the subsystem directory tree. Moving the files is optional but it can make it easier to manage the files that make up each model.

See *Dividing a Model into Controlled Units* on page 55 for background information that should be understood before proceeding with this task.

Tasks

- 1 Ensure that the base model has defined the initial controlled units, at least at the package level that corresponds to the subsystem partitioning.

The base model (Base) directory hierarchy for the sample model looks like:

```
Base.mdl
<Base>
  UseCaseView.cat
  <UseCaseView>
  <LogicalView>
    SubSystem1.cat
    <SubSystem1>
    SubSystem2.cat
    <SubSystem2>
  ComponentView.sub
  <ComponentView>
    SubSystem1.sub
    <SubSystem1>
    SubSystem2.sub
    <SubSystem2>
  DeploymentView.prc
```

- 2 Click **File > Edit Path Map** to create a Virtual Path Map for each top level package in the model (for example, each subsystem package). In our example, we could create path map variables SubSystem1LogicalPkg, SubSystem1ComponentPkg, SubSystem2LogicalPackage, SubSystem2ComponentPkg, and so on.
- 3 Check out the root packages in the Base model.
- 4 Explicitly save the Base model units affected by the new path map.

- 5 Check in the root packages in the Base model in order to save the modified file path information under version control.
- 6 If the Base model makes use of custom property sets, then these must be made available to the subsystem models. Click **Tools > Model Properties > Export** to create a file that can be imported to the subsystem models.
- 7 Create a new model by clicking **File > New**. This model will be used for the first subsystem. Enable version control for this model by enabling the SCC (Version Control) or the CC add-in using the Add-In Manager. Ensure that the path map variables are defined correctly.
- 8 If the Base model makes use of custom property sets, then ensure that these are available in the subsystem model. Click **Tools > Model Properties > Replace** to import the file containing the property sets.
- 9 Control all the elements in the new model by right-clicking the Model in the browser and clicking **File > Units > Control**.
- 10 Save the model in the appropriate local working directory for your version control system clicking **File > Save As** (for example, /vob/SubSystem1). We suggest that you create a dedicated directory for each subsystem.

For example, name the subsystem model SubSystem1 and store it in a directory called SubSystem1.

You may want to add the subsystem model to version control at this stage. For the SCC add-in, click **Tools > Version Control > Add to Version Control** to ensure that all the controllable units are added. For the CC add-in, click **Use Tools > ClearCase > Add to Version Control**.

- 11 You can optionally move the packages that make up your subsystem from the base model directory hierarchy into the subsystem model directory hierarchy created when you saved the new model.

The actual steps involved in moving the files and directories within version control depend on the version control tool used.

For each package that will be part of the subsystem, move the package controlled units into the corresponding directory level in the new model, and then move the directories for each package to the corresponding location. The resulting directory hierarchy for the new model should look like:

```
SubSystem1.mdl
<SubSystem1>
UseCaseView.cat
  <Use Case View>
<Logical View>
  SubSystem1.cat
  <SubSystem1>
  ComponentView.sub
  <Component View>
  SubSystem1.sub
  <SubSystem1>
  DeploymentView.prc
```

If you move the files, edit the associated path maps to reflect the new file locations.

- 12 Add the subsystem packages into the subsystem model by clicking **File > Units > Load**. These packages should be added in at the same location in the subsystem model hierarchy as they were in the base model.

If you previously added the subsystem model to version control, you must manually check out the root packages that are affected.

- 13 Save the subsystem model.
- 14 Use the SCC or CC add-in command **Add to Source Control** to enter the changes for this subsystem model into version control.
- 15 We recommend that you create a default workspace for each subsystem model.

After splitting the original model, you will typically not use that model for any further development. You may choose to create an equivalent model that shares in all the subsystems. For example, in our example we could create a new model called NewBase which shares in the packages in SubSystem1 and SubSystem2. This model cannot be used to edit any of the subsystems but it might be useful for building and/or testing.

Managing/Administering a Model

The Rational Rose manager or administrator is responsible for providing the overall version control infrastructure and environment for the development team.

Before starting team development work, the following tasks must be completed:

- Setting up Compatible Workspaces
- Setting up version control system and repository
- Partitioning the model into controlled units
- Adding the model to version control

After these steps are completed, development can start. However, consider these additional responsibilities:

- Defining developer work areas
- Creating labels and lineups
- Manipulating version control repository

Configuring Compatible Workspaces

To effectively work as a team, each team member should have a consistent workspace for working in a model. The starting point is the model structure created by the model Architect.

The tasks for managing a model include:

- **Defining Rational Rose model defaults.** All team members working in the same model should adhere to the same rules and should use the same model properties, including those settings that affect diagram layout, style, format, and so on.
- **Defining the physical storage structure for model elements.** In this task, you determine how the various model elements (specifically the controlled units) are organized.
- **Defining virtual path maps.** Defining the root of the hierarchy as a symbolic name is the first step in setting up a multiuser environment. (See *Understanding Virtual Path Maps* on page 67 for information about virtual path maps.) Each team member controls the definition of these symbolic names in his or her own workspace. Path maps are essential for working in a team since members often cannot work in the same directory on their local machines. By using path maps, you can distribute and relocated physical files.

Configuring a Version Control System and Repository

Before placing Rational Rose models under version control, there are steps that must be followed to configure the version control system to allow proper integration with Rational Rose. Most of these tasks are performed outside of Rational Rose and require knowledge of the version control tools you use. If you are unsure about the procedures, please see your version control documentation.

Before continuing, please review the tool-specific documentation in *Working with a Version Control System* on page 109.

After reviewing this material, ensure that a repository is properly set up for integration with Rational Rose.

Partitioning the Model into Controlled Units

Controlled units are the smallest Rational Rose model elements that you control via a version control system. Therefore, the packages that are controlled should be selected carefully. For example, it is not always correct to control all the packages. Packages that are controlled units may contain packages that are not controlled units and vice versa. Control the units that provide sufficiently low level of granularity to allow project members to do their work without preventing other project members doing theirs. Ownership of packages and controlled units is very important for effectively working in a team.

For complete details on Controlled Units, see *Dividing a Model into Controlled Units* on page 55.

Because controlled units correspond to files in your file system, only one team member should be allowed to work on a given controlled unit at any one time. While this works well in most cases, there are situations when it is necessary to allow multiple team members to work simultaneously on the same controlled unit. The following procedure can be used to that effect:

- 1 The current owner of the package of interest creates subpackages for each team member who needs to get involved in the work. These packages can even be named after the team members.
- 2 Within each package, relocate the elements of the parent package that you want to assign to the different team members.
- 3 Control the new packages and assign them to their intended owners.

When the work is complete, uncontrol the temporary packages, relocate all elements in them to the original package, and delete the temporary packages now empty. This is a tactical solution to a controlled unit access problem that can be used as required so that package structures and controlled units are not permanently created on an arbitrary or expedient basis, but for sound architectural reasons.

Save Model to Local Work Area

Before placing the model under version control, it must be saved to the local work area. Save the model to the directory you have associated with your version control repository.

Adding the Model to Version Control

The easiest method of adding all applicable units to version control when using the SCC add-in is to click **Tools > Version Control > Add to Source Control**. When using the ClearCase add-in, you can add units to source control by clicking **Tools > ClearCase > Add to Source Control**.

Defining Developer Work Areas

The model manager should think about how each worker (developer, integrator, and other team development roles) will work individually and access specific versions (lineups) of a model. This usually involves defining labeling policies.

The model manager should provide guidelines to the rest of the team as to how work areas should be created for each developer. In some cases the manager may need to actually create the work areas.

Defining work areas is tool dependent, and the steps required for setting up a work area for single stream and parallel stream development can be quite different. See *Working with a Version Control System* on page 109 for more information.

Creating Labels and Lineups

Labels, and the use of labels to create lineups, are crucial to any successful development strategy. There are many ways to use labels and lineups, though, and the specifics of each are highly specific to each organizations development environment and version control tools.

Manipulating the Version Control Repository

It may be necessary to move or rename files in the repository. This should only be performed by someone who is familiar with the version control tool being used. In many development environments, moving and renaming is always carried out by the version control administrator, who can carry out the task most effectively.

Developing/Implementing a Model

Developers work day-to-day with a subsystem model under version control. Therefore, each developer should be familiar with the material in chapter 6, *Working with a Version Control System* on page 109 .

Setting up Version Control

Before using Rational Rose with your version control system, you must perform any tool-specific configuration as described in *Working with a Version Control System* on page 109.

If you have customized Rational Rose to work with another version control tool, then you should ensure that tool is correctly installed on each developer workstation.

Setting up Developer Work Areas

Before working on a version controlled model, you first have to get a specific lineup of controlled units onto your local disk. From there you can start working on the model. Your Version Control Administrator or Integrator will know how to determine the specific label or configuration that should be used to create a local work area. Next, it is a matter of configuring a local work area before running Rational Rose.

Getting a Specific Lineup of a Model

When a developer begins a development task, they must start with the correct version of the model files. The steps involved vary depending on your team development process and the underlying version control tool. For Rational ClearCase, the developer should be using a config spec that defines the view to include the correct versions of the model elements. For Microsoft Visual SourceSafe, your team may be using labels to mark the correct versions and the developer should perform a Get based on that label by using the **Label** field available from the **Parameters...** button in the **Get** dialog box.

Opening a Model Under Version Control

Opening a model under version control is no different than opening a non-version controlled model. In either case, opening the model file or its associated workspace file is the recommended way to load the model into Rational Rose. Default property settings will typically be made available by the Version Control Administrator, see *Make default property set available to project members* later in this chapter.

Working under Version Control

After your model has been placed under version control, use the following procedures:

- Check Out Parent Package.

When a new controlled unit is added to a version controlled model, you will have to check out the package in which the new unit will be placed. If there is excessive contention for parent packages, then you may wish to partition the package into several smaller packages.

- Checking Controlled Units In and Out of Version Control.

After you have a model under version control, you should check out elements before you edit them. Depending on the version control settings, Rational Rose may force you to check out before editing.

- Undoing a Check Out.

After you check out a controlled unit, you may choose to undo the check out and not submit a new version.

Comparing and Merging Model Elements

See *Comparing and Merging Models* on page 77 for more information.

Promoting Changes for Integration

When working in a single stream development process, there is no explicit integration step. Instead, submitting changes to the version control repository effectively integrates them with the existing file versions.

Integrating Changes

Integrating changes is highly dependent on the development process being used. The primary goal of the Integrator is to produce an updated lineup of model elements to use as a basis for subsequent development activities. This often involves merging changes from multiple developers (using the Model Integrator) and performing local builds to verify sanity.

Automating Model Validation

Rational Rose provides an automated method to determine if a model is valid. These steps can be incorporated into an automated build process to determine if the code generation and compilation steps of the build should be performed.

Using the Rational Rose Extensibility Interface (REI), you can write a script that:

- 1 Opens a specified model (using the `Application.OpenModel` method).
- 2 Saves the log to a specified file (using the `Application.SaveLogAs` method).
- 3 Closes Rational Rose (using the `Application.Exit` method)

For more information on the REI, see the Extensibility Interface documentation. This script could be invoked as part of an automated build. The automated build script can then search (for example, `grep`) the log file to determine if any errors/warnings were encountered when the model was opened. If problems were encountered, then the build script can email the log file to the builder. If no problems were encountered, then the build script can continue with the code generation and compilation steps.

Contents

This chapter is organized as follows:

- *Goals of Team Development* on page 33
- *Sharing Within a Team Environment* on page 34
- *Protecting Configuration Items From Unintentional Changes* on page 35
- *Managing Relationships Between Configuration Items* on page 40
- *Managing and Delivering Configuration Items* on page 41
- *Improving Efficiency in Team Development* on page 43
- *Recommendations* on page 44
- *Advanced Concepts and Heuristics* on page 48

Goals of Team Development

Developing complex systems requires that groups of people, such as analysts, architects, developers, and testers, coordinate their efforts to produce the finished product. Consequently, they must ask themselves the following questions:

- What are we trying to accomplish in team development?
- What are the goals of team development?

How does Rational help implement strategies and best practices to meet those goals?

- What do I need to do to have efficient and effective team development?

The purpose of this book is to outline the goals of team development, and recommend some best practices when using Rational Rose to help ensure success.

Team development touches on development, testing, configuration management, project management, and other disciplines such as engineering, analysis, and design.

This overview of team development helps provide your team with an overview of the challenges associated with team development, while specifically outlining the tools and mechanisms Rational Rose supports to aid in implementing a team development strategy.

The Guide to Team Development provides an overview of the basic team development concepts in Rational Rose and specifies how to configure and use Rational Rose in a team environment.

The goals of team development are to:

- Allow team members to share their work with a team. See *Sharing Within a Team Environment* on page 34.
- Protect configuration items from unintentional change. See *Protecting Configuration Items From Unintentional Changes* on page 35.
- Manage the relationship between configuration items. See *Managing Relationships Between Configuration Items* on page 40.
- Deliver specific versions of configuration items to interested parties. See *Managing and Delivering Configuration Items* on page 41
- Reduce or eliminate disruptions to team activities. See *Improving Efficiency in Team Development* on page 43.

Sharing Within a Team Environment

After a developer completes an activity (work), they require a mechanism to share that work with others. Integration is the mechanism that permits the integration of changes made by a team member into what is currently being shared.

A version control system can facilitate the work flow of team members. A team member working on a shared artifact acquires some type of implicit or explicit permission to check-in their work by performing a check-out prior to working on the artifact.

The check-out status for the artifact indicates to other team members that work is currently being done to change the artifact. A configuration manager or configuration system can monitor these operations and enforce any policies. The mechanism can involve the use of a version control system, or it may be an unsophisticated implementation whereby the check-in is a simple copy, and the communication is verbal between developers. Regardless of the mechanism used, an awareness of a change at the appropriate levels must be achieved, and you must assess the implications of the change.

A check-in does not necessarily imply that the artifact is immediately available to team members. Typically, it is useful to work with older versions of shared artifacts until such time as the team is ready to access the latest version.

A version control system allows the team to return to previous versions of work, while providing an audit trail of changes. The desire to associate work with specific requirements is a type of policy the Integrator can enforce at integration time.

Work produced by a member of a team can affect other members of the team; therefore, those effects must be intentional. A copy of the work is made available to a team member in an environment isolated from other team members.

The environment is only isolated one-way. The work environment can see shared team artifacts, but other environments are not effected by the isolated environment. The benefits of this type of implementation are:

- Development team members can produce builds in their isolated environment in an iterative, non-intrusive way. It also allows team members to see a read-only version of shared work.
- Testing teams can perform a series of tests on a specific lineup of work in their own test environment. A lineup is a collection of specific versions of files from a version control repository.
- Production users can use a particular lineup of work that has met quality control criteria.

Protecting Configuration Items From Unintentional Changes

There are several ways revisions can cause unintentional changes:

- Direct conflicting change where one change overwrites another. See *Overwriting a Modification* on page 36.
- The source from one change conflicts with another change by removing a dependency that one of the changes relies on. See *Adding Dependency Issues* on page 38.

Table 1 shows the legend that explains some images found in Figure 7 through Figure 11.

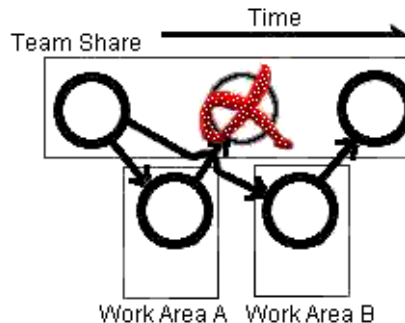
Table 1 Image Legend

Image	Description
○	Represents a unit of work or configuration item
↗	Represents movement
✘	Represents an unintended change

Overwriting a Modification

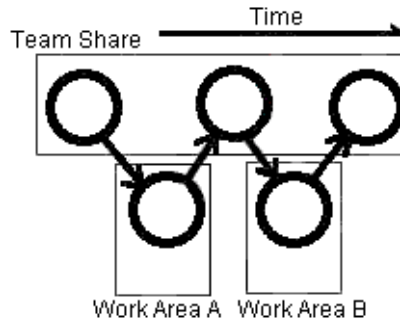
If a team member shares their work with the team, not realizing that someone else produced or edited some work with the same name, they may overwrite the changes of the other team member.

Figure 7 Overwriting a modification



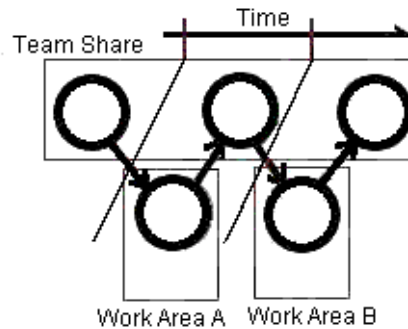
Most version control tools provide adequate protection from this type of unintentional change through a process of obtaining permission to make modifications, called a check-out. The version control tool grants implicit permission when there are no check-outs currently in place. When one team member has an artifact checked out, other team members are denied permission to check out that same artifact until it is no longer required by the first team member. Figure 8 shows a scenario where a check-out is followed by a check-in, allowing the sequence of events to iterate.

Figure 8 Check-out and Check-in Scenario



This type of scenario may cause contention that is unacceptable for high traffic work items. The diagonal lines in Figure 9 indicate that a check-out cannot occur until the previous check-in process completes.

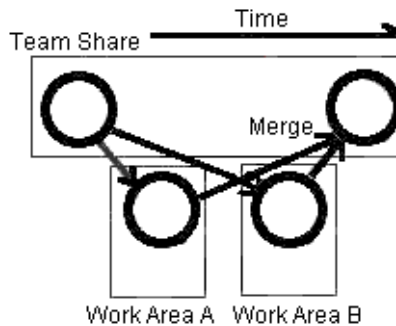
Figure 9 Checking Out an Artifact After it is Checked In



The problem illustrated in Figure 7 commonly occurs in strategies that do not use a version control system. Because previous versions of configuration items are always available to developers, the possibility of having this type of unintended change always exists. A developer may make changes to a private copy of an artifact without permission to do so. Subsequently, they may acquire the appropriate permission and check in the changes of the local copy that may not represent the latest version of the configuration item.

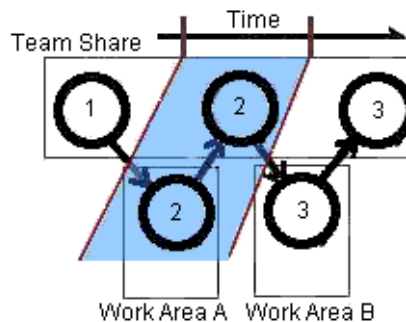
You can use a merge tool to apply a combined set of changes in situations when multiple team members have permissions to make changes to a single artifact. Figure 10 shows how you can merge two changes made to the same artifact.

Figure 10 Merging Changes Prior to Check-In



It may be difficult to remove a set of changes that occurred in a previous version of an artifact. The situation in Figure 11 shows us three versions of an artifact. If you want to remove all changes applied to the second version (the changes occurring between the two diagonal lines), you may encounter difficulties.

Figure 11 Comparison Between Versions



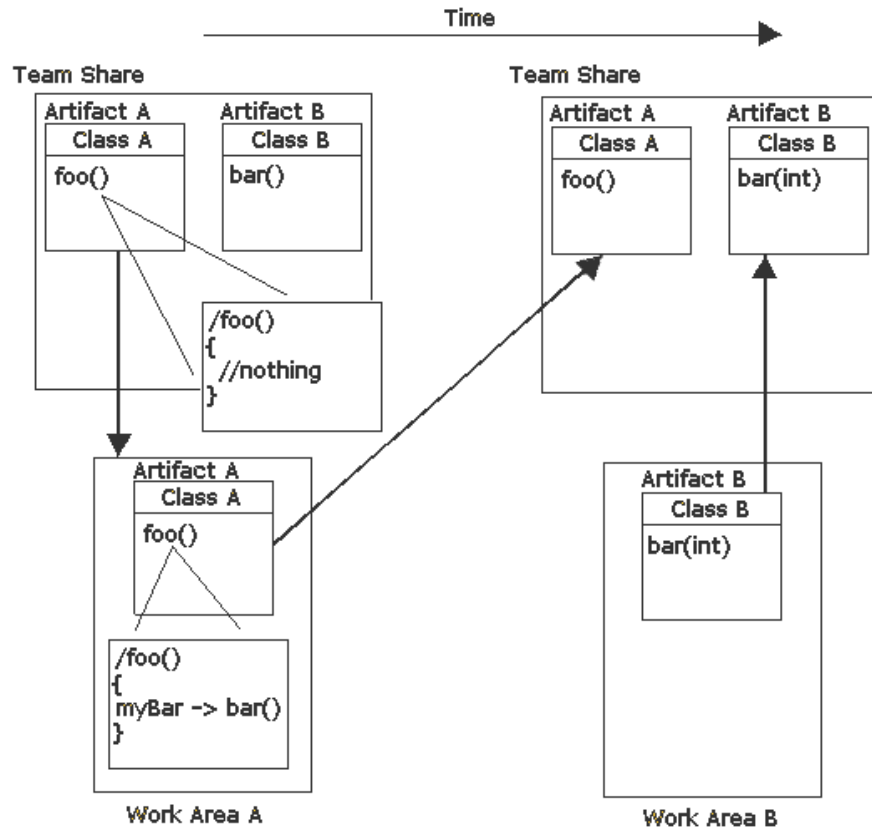
For example, the changes between version 1 and version 2 must be compared to the changes between version 1 and version 3.

Obtaining adequate permission to modify artifacts helps to ensure that unintentional changes do not occur. Configuration management can choose to implement and enforce this type of policy.

Adding Dependency Issues

Modifying an artifact may cause a conflict with another change if it removes a dependency that one or more other artifacts rely on. Figure 12 shows how this type of problem can occur.

Figure 12 Removing Required Dependencies



Developer A and B individually check out artifacts A and B respectively, and have access to the shared version of artifact A and B respectively.

Developer A creates a new dependency in `foo()` by adding `myBar-> bar()`.

Developer B makes changes to `bar()` in class A by changing the parameter signature to integer.

Changes to `bar` - from `bar()` to `bar(int)` - cause any references to this function to fail. The changes made by Developer B to artifact B that are referenced by `foo` from artifact A are not valid.

Note: Most merge tools are unable to identify a conflict here because they compare items of work individually, and not against all referenced work.

This type of change is common and may have serious implications. Often, when product maintenance is underway and feature development is concurrently managed, the maintenance person or developer may be unsure or unaware of all dependencies

involved in a proposed change. Rather than research all the dependencies associated with the artifact, they do not modify the original item. Instead, they create a new item with the proposed changes.

Changing Language Semantics

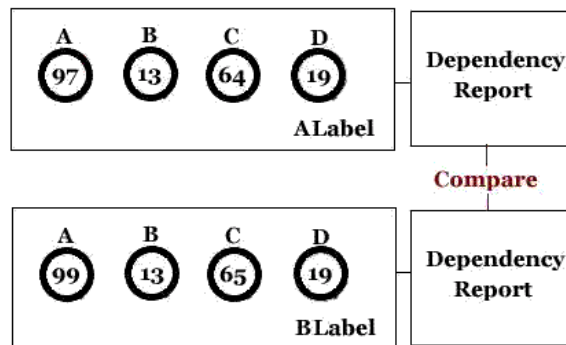
Managing Relationships Between Configuration Items

Team members must understand and use the dependencies between configuration items to reduce or prevent unintended changes in the system.

Because most configuration items do not work in isolation from other configuration items, a set of particular versions of configuration items has a set of dependencies. When a set of versions of configuration items changes, the possibility exists that the dependencies also change. It is useful to compare the set of dependencies from one set of versions to a previous set to ensure that dependency changes are intentional.

A set of versions of configuration items is also known as a lineup. Figure 13 shows a generated dependency report for a lineup identified by the label called ALabel. Later, a comparison is made between ALabel and another dependency report generated for the lineup identified by BLabel. Although the dependency reports themselves may be too large to be of any use, a good differencing tool can make it easy to see dependencies modified since a previous stable lineup of the project artifacts.

Figure 13 Comparing Dependency Reports



Specific to Rational Rose, there are several levels of dependencies that must be understood and managed:

- Dependencies between control units in a model.

The Rational Rose Toolset interprets what is loaded into memory as the entire model. When loaded from separate configuration items, the model elements stored on secondary storage must be loaded such that it creates a model where elements are consistent with any corresponding relationships.

- Model element relationships

Managing and Delivering Configuration Items

A specific set of configuration items in their appropriate version (a lineup) must be accessible and reproducible. Protection of these version sets is important. Like most one-to-many relationships, a label is often stored many times; once with each configuration item.

Figure 14 Labelling Configuration Items

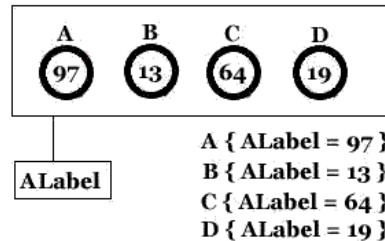


Figure 14 shows the following:

- The full set of configuration items are not all labelled at the same time.

Note: If the label is applied while the lineup changes, this may create an inconsistent state.

- A configuration item may be overlooked or may not be associated with the label. Sometimes, it is better if the configuration item is not associated with the label. The label associated with a previous version of the configuration item would make the problem difficult to find.

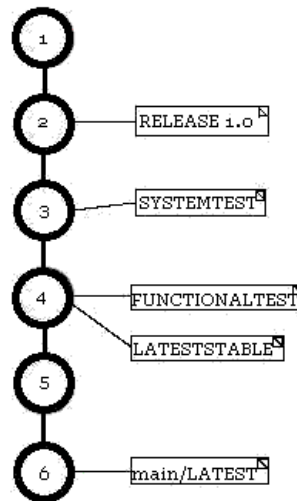
A fixed label is the first primary use of a label, forever identifying a version of a configuration item with a specific label. It is also useful to include in your naming convention details, such as the date and time in a label name.

The two types of floating labels (logical and explicit) become associated with different versions of a configuration item.

Over time, a logical floating label is arbitrarily associated with the latest version of a configuration item on a particular branch or stream. For example, “LatestDevelopment” or “JanesLatest”.

An explicit floating label is explicitly assigned to different versions over time, and it is almost always based on the associations of another label and not with the latest versions on a branch or development stream. This means that it is not necessary to “freeze” the configuration items to associate a label with versions already assigned to another label; only the state of the base model must be frozen. For example, Figure 15 shows that the SYSTEMTEST label is associated with version 3 of this particular configuration item.

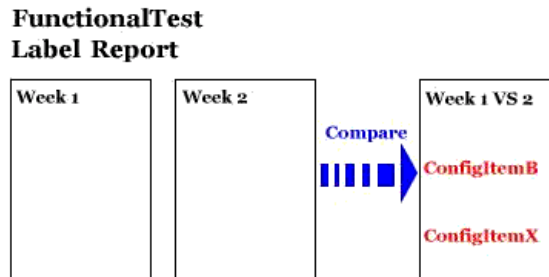
Figure 15 Example of Labelling Items



When the test team for the system is ready, they can associate the label with all the versions associated with FUNCTIONALTEST. No changes should occur to the FUNCTIONALTEST label until the SYSTEMTEST label change is complete. However, assigning LATESTSTABLE with the current versions of all the files on the main branch of development requires that no new main branch versions are added to any of the configuration items until the LATESTSTABLE label change has completed the operation. Since labels can be moved, it is good practice to produce and keep a dated report on the versions associated with important labels for milestones.

Creating and comparing label reports of different dates on a regular basis can reveal trends and areas that require additional testing to ensure quality of volatile areas of the system. Figure 16 shows label reports for two consecutive weeks.

Figure 16 Comparing Reports



Teams looking at a particular lineup of configuration items should retrieve artifacts solely on the selection of configuration items associated with a specific label. Testing in this type of environment quickly identifies overlooked configuration items because of a missing association. It also ensures that all necessary configuration items are included as they are made available to other teams.

Improving Efficiency in Team Development

The implementation of some team development practices can hinder the implementation of other team development goals. Planned activities may be part of the strategy to deal with implementation issues in a team environment.

You can reduce unplanned activities by using an effective strategy that promotes handling conflicts up front. Your configuration management plan should implement a strategy that promotes team development goals with as little impact to team activities as possible. See *Goals of Team Development* on page 33 for more information about specific team development goals.

The stakeholders of the configuration management plan are almost everyone, and their needs vary significantly. The description of the roles and tasks in this document is general and must be customized to suit your particular development organization.

Model Architect Role

The Model Architect establishes the overall structure of the model: the grouping of elements into packages, the separation of models into subsystems, and the interfaces between these major groupings. The Model Architect adapts the structure of the model to reflect the organization of the team.

Recommendations

Protection of configuration items and the ability to deliver a consistent set of configuration items are the main priorities of the configuration management plan. An implementation of a plan to achieve the other goals should support this ideal.

Use the source control operations supported through Rational Rose to facilitate the implementation of a greater configuration management plan. For complex projects, a large part of the configuration management strategy that deals with Rational Rose models, may be strict ownership of shared packages.

You may think of shared packages as the building blocks of the system. One Rational Rose model brings all the building blocks together in a coherent system. Many working models are used with the sole purpose of creating and testing those building blocks.

Source Control Fundamentals

In chapter 5, called *Working with a Version Control System*, specifies the source control operations supported from Rational Rose . It outlines some of the differences in view-based and file-based source control systems. There is also a discussion on versioning strategies.

The ability to associate labels and create a lineup exists in both types of source control systems. Using a parallel stream versioning strategy while maintaining a single stream versioning policy, provides the safety inherent in single stream versioning strategies, and also the ability to control parallel development of the same artifacts among different teams.

Any source control tool that allows branching is capable of supporting a parallel stream versioning strategy. An example of appropriate streams of development are:

- Development streams, where developers make changes to the configuration items.
- Integration stream (implementing requirements and features) managed by Integrators.
- Product version maintenance streams (providing fixes for bugs/defects identified after release date) also managed by Integrators.

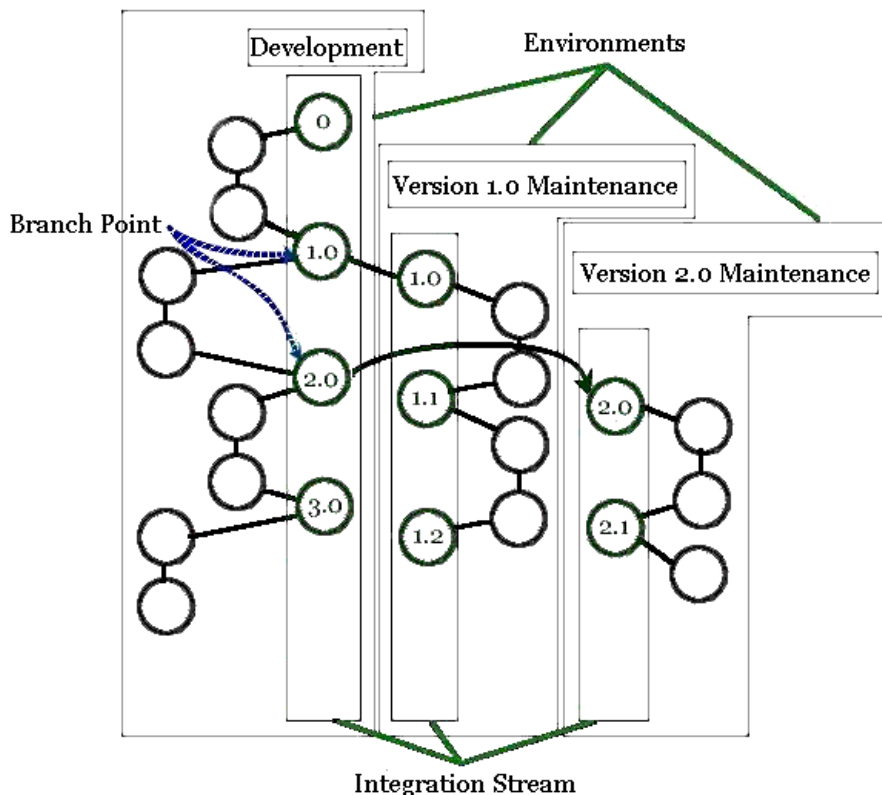
Include a maintenance stream for every product version currently supported by your organization. When support for the specific product version is concluded, these streams should end.

Note: *You can use merge tools, such as Model Integrator, for merging simple, non-conflicting changes. However, because of their limited semantic support, we do **not** recommend that you use automated merge tools when there are many conflicting changes.*

Bugs and defects reported against a version of the product should be evaluated against the product under continued development in the new development stream. Other versions of the product that may be affected by the bug/defect are under continued support. Apply corrections to all affected versions through a manual merge, or through focused merges.

If you implement a parallel stream versioning strategy, maintain virtual single streams within the parallel streams. For example, Figure 17 shows a version tree history for a configuration item. A branching of development effort occurs at version 1.0, and version 2.0 of the configuration item

Figure 17 Parallel Stream Versioning Strategy.



Only one side of the branch is checked back into that integration stream. The Integrator uses the main streams of development and may be unaware of the details of individual changes. Therefore, from the perspective of these streams, they are a single stream of development only receiving updates from one source that has

permission to modify the next version in the stream. If you require merging, perform it outside of these integration streams, and sanity test it before integrating it as a new version.

Do not associate product verification labels and packaging or deployment labels with versions outside these main integration streams of development. When working with files such as test scripts that are version controlled, consider these files as if they were in a separate project.

You may have separate streams for the development and maintenance of these scripts as well, but this should be thought of as a different project than the one it supports from a version control perspective. That supporting system may have logical ties or parallels with the product under development.

Preempting Conflicts

You want to minimize more than one concurrent check-out of a configuration item. If this strategy results in unacceptable contention for a configuration item, or a dead-lock occurs, put overrides in place to deal with the contention.

A dead-lock occurs when team member A requires a configuration item checked out to team member B to finish work, and team member B requires the configuration item that team member A currently has checked out. Because this is done up front, there is an awareness that changes are being concurrently made to the same configuration item, and these changes can be managed to minimize the likelihood of unintended change.

This type of concurrent work must occur outside the main development streams. When it occurs, resolve this type of situation as quickly as possible and provide adequate testing of the configuration item following the period of concurrent change, to ensure no unintended changes occurred as a result.

The Rational Rose shared package capability can implement an ownership strategy to limit the scope of implicit permissions to change configuration items.

Managing Dependencies

To effectively manage changes to the dependencies in your system, you must create and enforce your own team processes. For Rational Rose projects, you must identify the following dependencies:

- Dependencies between control units in a model.
- Model element relationships

If you do not have a formal reporting mechanism that automatically identifies these dependencies, every change must be addressed to ensure that dependencies are researched and assessed as a result of the change.

Labeling

When considering labelling, we recommend the following:

- Establish an environment for each group or individual that will work with a specific set of configuration items in isolation from other changes for *any* length of time. For most version control tools, this is established with directories containing a copy of the appropriate version of the configuration item identified through a movable label. The team member performs work on artifacts in these directories, and this set of directories is also called the sandbox. ClearCase users have the capability of achieving virtual directories through the configuration specifications of dynamic views.
- When using file-based version control tools in Unix systems, developers can configure a directory that references the shared work through soft links. When team members modify the reference in the directory, the link is broken and it is replaced in the sandbox by the modified file.
- Create dated reports for each floating label on a regular basis, listing all configuration items associated with the label and the associated version. We recommend that you add the report to your version control system. You can use the data from the report to identify how the set of configuration items changed over time, and to help you identify volatile and stable elements of the system. Fixed labels do not require this type of report. For a label associated with a set of configuration items that do not change often, you can reduce the frequency to some appropriate interval, or on an ad-hoc basis.
- Define your labeling strategy as much as possible before you begin. Use a naming convention so that everyone can understand the labels.
- Identify labels that may require protection from modification, and those labels that may require restricted access.

When Merging is Necessary

Merging is necessary when an awareness exists that concurrent development may result in conflicting changes. Perform the merge as often as possible. Each developer involved in a concurrent change must regularly work with a merged version of the ongoing work to identify adverse or unintended change.

The intention is to reduce the amount of lost work that can occur when conflicts arise. A conflict identified early reduces the amount of re-work necessary. This kind of concurrent work on the same artifacts must be done in isolation from other work.

The way ClearCase facilitates integration branches, it is wise to choose a special integration stream for the concurrent changes to a configuration item. This isolates the remaining artifacts in your system (which uses mutual exclusion) from these changes until the configuration item can go through extensive quality verification.

With other sandbox type systems, one developer merges other developer's work, and then provides the merged version to the other developer.

After every merge, assess changes to semantic relationships and other dependencies.

Advanced Concepts and Heuristics

This section includes additional information about advanced concepts and heuristics in the following areas:

- *Moving Controlled Units* on page 48
- *Parallel Development* on page 50
- *Model Integrator* on page 51
- *Using Rational ClearCase Multi-Site* on page 52
- *Additional Heuristics for Team Development* on page 52

Moving Controlled Units

When a model element moves from one package to another, and the element is under configuration management (CM) control, Rational ClearCase does not move the file corresponding to the model element into its new directory.

Some CM systems do not support moving history when moving a file from one directory to another. Consequently, if the file is not moved to its corresponding directory as the element in the model is, the operations that involve labeling will not be done correctly.

When a UML package is assigned a Rational ClearCase label, ClearCase performs an operation on the directory and all its contents. However, if the controlled moved, its corresponding element will not be labeled correctly.

When the name of a package, diagram (one that can be a separate control unit), or classifier changes, and that element corresponds to a controlled unit, the source control element in Rational ClearCase should also change.

What are some use-cases that relate to moving controlled units?

- Control to package level granularity: move an element from one package to another.
- Control to package level granularity: move an element from one package to a scratch-pad package.
- Control to package level granularity: move an element from a scratch-pad package to package under CM
- Delete a package from the model and the file exists in the CM repository. The script created to move a controlled unit could identify these files as well.
- A file is under CM, but an element with that name already exists. The tool currently generates a unique file name, and provides a warning.
- Changing from a controlled unit, to uncontrolled unit, and then back to a controlled unit.

Some solutions for these use-cases may include:

- Writing a script to move a controlled unit. Ensure that the script detects the controlled units whose locations do not match the model element, and then repairs the locations.
- Writing a script to rename the controlled unit, when required. Ensure that the script detects any name differences, and then repairs the names.
- Since Rational creates the CM scripts for ClearCase, and ClearCase supports moving history with a file, ensure that the scripts are fixed to address this issue.
- Ensure that the ClearCase move script can handle a move between VOBs.

Considerations

In ClearCase, the relationship between a file element and directory elements is such that an element may be in multiple directories at the same time, possibly even in the same view. This does not necessarily complicate things for the toolset, but requires careful consideration.

A Rational Rose model element may be saved as two distinctly named Rational ClearCase elements.

Heuristics

Use package-level granularity rather than class-level granularity. Class-level granularity helps reduce issues when moving classes, and issues with package ownership.

Parallel Development

Parallel Development is a term which sets high expectations regarding collaborative development, where there is a need for multiple users to work together on a common set of artifacts to achieve the same goals.

When collaborating on a common set of artifacts, consider the following approaches to collaborative development:

- When more than one user needs to make changes to the same artifact, they must share the artifact; the changes are made serially, one after the other. Although this is the most reliable approach, it is perceived by most users as not being most efficient. This approach can be managed using the check-in and check-out features of most CM systems.
- When more than one user needs to make changes to the same artifact, they can make the changes at the same time. The changes are merged back into one artifact at a later date. The benefit of this approach is that work goes on in parallel, and it saves time. The problem is that arbitrary and uncoordinated changes on the copies of the same artifact can be difficult to resolve during the merge process. And in fact, may never be resolved and the changes from only one contributor are accepted, and the other discarded.

The development process and tool chain can have a significant impact on the opportunity to use and the effectiveness of the second approach. The second approach is characterized as Parallel Development. For the purpose of this discussion, the term Parallel Development refers specifically to this second approach to collaborative development.

It is unrealistic to expect employ parallel development without any constraint or guidance. Too often, this technique is used without coordination or planning. Sophisticated tools, such as Rational ClearCase, may not be properly used and can lead to this misperception. The design artifacts at the center of collaborative development have complex interrelationships within them, and between them. These higher level abstractions and concepts are not easily, and cannot arbitrarily be, merged without some experience. Fortunately, when team members are working

within a well-defined process, and there is a clear definition of roles and responsibilities, most changes made in parallel are done in a complementary manner. A certain amount of conflicting changes are inevitable. You can resolve the changes by choosing one or the other. These conflicting changes must be expected and their frequency should be minimized. If they are unexpected, it may be counterproductive and time is being wasted by changes that will not be discarded.

The following guidelines will help maximize the efficiency and productivity of a process that employs parallel development:

- Scrutinize and minimize the occurrence of every conflicting change in the merge.
- A well documented and communicated development plan helps ensure that every developer knows how they are contributing and what they will implement. This helps minimize duplication of effort, even at the lowest level of detail.
- Establish clear ownership of design artifacts, and use source control to enforce it.
- Invest time into understanding what the Rational Rose Model Integrator will and will not do during a merge.
- Follow all guidance specific to the Rational Rose Model Integrator regarding the types of changes that it can reliably merge.
- Resolve all issues relating to merging parallel changes prior to integration.

Model Integrator

The Rational Rose Model Integrator is a powerful tool that manages the merging and differencing of models at the Rational Rose meta-model level. It is not a visual model or UML semantic-level merge tool, therefore it lacks a number of features that can make the merging of models more efficient and more accurate.

For every use-case of Model Integrator that fails to do what you may expect, there are many other use-cases that do add value or do what is expected, and will save time. When using Model Integrator, you must understand what it can do efficiently and properly, and what should be avoided.

When you plan for a graphical change (a layout change) to a diagram within a model, only one person should make this change. This ensures that during the merge process, all of the graphical changes are accepted by one contributor and merging at a lower level of detail is not allowed.

You can change most information associated with a model element as long as it is information not related to its identity. For example, the Action property or Documentation field.

Using Rational ClearCase Multi-Site

When a team follows best practices, for example, being careful about artifact ownership, they can use Rational ClearCase Multi-Site to work on separate branches.

Rational ClearCase Multi-Site is a powerful tool that can help you with the challenges of a distributed team development. When using Rational ClearCase Multi-Site, you must consider the following:

- Rational ClearCase Multi-Site has a restriction that a branch is owned by a site.
- Only developers on that specific site can check out to that particular branch.

Additional Heuristics for Team Development

- Begin with high level of granularity for controlled units when area of a model is immature. As the area of a model becomes more mature, then it's level of granularity can be lowered.
- During the architecture phase, the granularity is coarse. When the architecture is released to the designers, decrease the granularity to manageable pieces for efficient team development.
- Use a layered architecture, where the coupling between layers is minimal and well-defined. This kind of architecture is also called loosely coupled.
- Define the interfaces between layers of the architecture early and minimize changes to these interface elements.
- The interfaces and associated components at a layer boundary are released separately and have their own test and release schedule. There are one or more separately released components in each layer.
- Every controlled unit should have only one owner.
- Plan for conflicting merges and attempt to minimize them throughout the development life-cycle.
- Only merge controlled units with primary edits are back into the integration stream.
- If the system is sufficiently complex, divide each layer into subsystems.
- Ensure that subsystems have a well-defined and minimal interface to other subsystems.

- Subsystems are not necessarily confined to one layer. Interfaces at lower and higher levels of abstractions should coincide with one of the architectural layers. Subsystems may encapsulate their own set of layers which satisfy particular objectives.
- Employ at least three streams of development: release stream, integration, and developer.
- You can place a new part of a model under source control after it has had some (minimal) testing.
- Do not make frequent or large changes to a superclass.
- A process that employs parallel development should insure that
- Subsystem interfaces may need to be modified by both users, but changes should be planned, controlled, and authorized by owner (or group).
- Appoint one responsible person for each interface. This person is the only one that can change the interface. For example, all requests for changes must be sent to this single team member for them to make the required change.

Dividing a Model into Controlled Units

4

This chapter is organized as follows:

- *What is a Controlled Unit?* on page 55
- *Working with Controlled Units* on page 59
- *Creating Virtual Paths to Controlled Units* on page 67
- *Checking References and Access Violations* on page 72
- *Organizing Controlled Units for Teams* on page 74

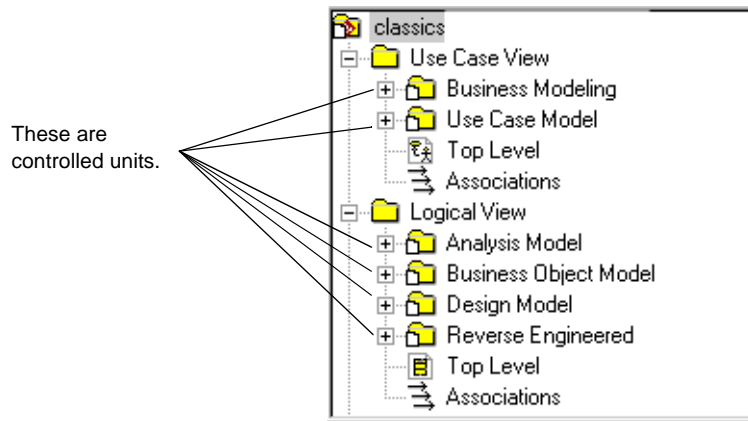
What is a Controlled Unit?

By default, Rational Rose saves a complete model as a single model (.mdl) file. However, when many users are working on a model at the same time, you can reduce contention and enable parallel development by dividing the model into a series of individual files called *controlled units*.

Controlled units are the configuration elements that a team places under version control. When using controlled units, each team or team member is responsible for maintaining or updating a specific unit.

The lowest level of granularity for a controlled unit is a package in the use case, logical and component views of a model since packages are considered the smallest architecturally significant elements.

Figure 18 Controlled Units



What Can be a Controlled Unit

You can create controlled units for packages in your Use Case, Logical, Component, and Deployment Views, as well as your model properties.

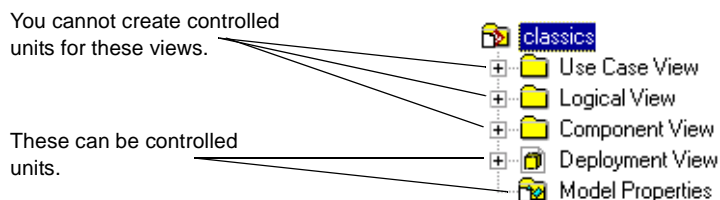
When you create controlled units, you name each new file and use one of these four extensions for the particular type of controlled unit you create:

- Logical packages and use-case packages are stored in .cat files.
- Component packages are stored in .sub files.
- A deployment view is stored in a .prc file.
- Model properties are stored in a .prp file.

You can have an unlimited number of .cat and .sub files associated with a Rational Rose model. Because a model supports only one deployment diagram, there is only one .prc file. Similarly, there is a single set of model properties and only one .prp file.

You cannot create controlled units for three of the top-most views, namely the Use Case, Logical, and Component Views.

Figure 19 View from which you cannot create Controlled Units



How Controlled Units are Related and What They Contain

When you create a controlled unit from a package, the contents of the package are moved from the model file or enclosing package, and stored in the new unit file. The new controlled unit file contains:

- All model elements that are in the package.
- All packages that are in the package, or a reference to those packages if they are also controlled units.
- All diagrams that belong to the package.

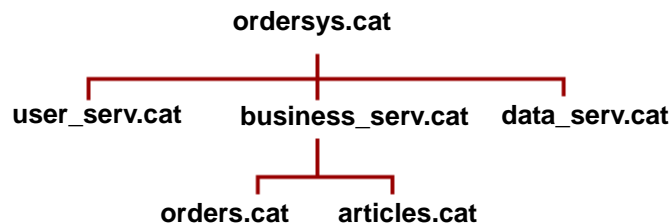
The original file no longer holds the contents of the package. Instead, the original file now only *references* the new controlled unit file.

The model file only references the first level of controlled units. Thus, a controlled unit that holds another controlled unit also holds the reference to that unit.

Packages own modeling elements such as other packages. Ownership implies a one-to-one relationship. Therefore, every package is owned by exactly one other package in the model.

When you work on a controlled unit, you can change its contents without affecting the controlled unit it might belong to, or the controlled units it encloses.

Figure 20 Example of a Controlled Unit Hierarchy



In Figure 20, developers can modify the file `articles.cat` without affecting `business_serv.cat`, and `business_serv.cat` can be modified without affecting `ordersys.mdl`.

You can create a virtually unlimited hierarchy of controlled units where top-level controlled units consist of references to other controlled units.

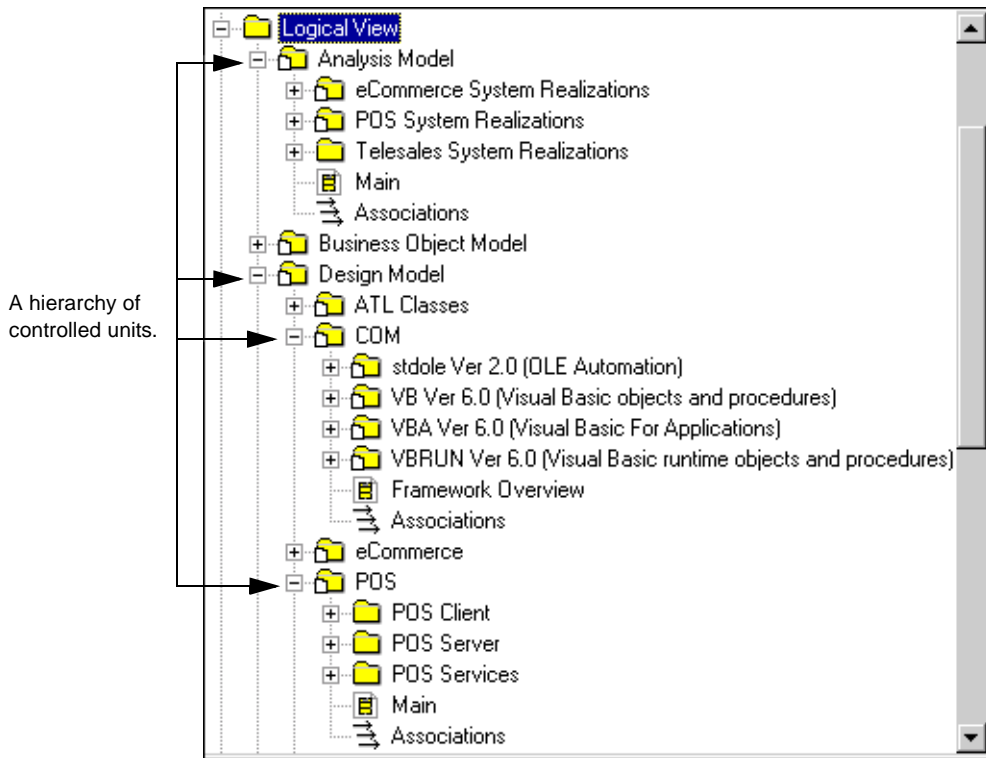
For example, you could make all packages controlled units with top-level packages that are pointers to nested packages. This enables two developers to check out packages that belong to the same higher level package.

Packages can also be shared. By creating controlled units, multiple models can share the same packages, allowing you to effectively reuse model elements.

How you partition a model and the type of hierarchy you implement will depend on how team members will operate, both physically (who works on which packages) as well as logically (how best to partition the model and preserve its design).

Figure 21 illustrates how controlled unit hierarchies appear in the Rational Rose browser.

Figure 21 Controlled Unit Hierarchies in the Rational Rose Browser



The format that Rational Rose uses for the model (.mdl) and controlled unit files is called petal format. Petal is a text-based format that allows you to open and view the model and controlled unit files in any text editor. The petal format is the same on Windows and Unix platforms, thus enabling teams of developers on different platforms to share models.

Working with Controlled Units

Creating Controlled Units

To designate a package as a controlled unit, you select the package in the browser or diagram then click **File > Units > Control**. Rational Rose prompts you to provide a location and a filename with the appropriate extension, then it moves the contents of the selected package from the model file (or enclosing controlled unit) into the specified file.

Note: After you create one or more controlled units, you *must* save your Rose model in order to save the new references in any enclosing controlled units or in the model (.mdl) file.

Because the model file (or the enclosing controlled unit, if the new unit is going to be created inside another unit) is changed when a new controlled unit is created, ensure that the enclosing file is write-enabled. If the file is under version control, you must first check out the file.

Carefully consider the file structure you implement when creating and saving controlled units. If the controlled unit will be under version control, the file structure you use may need to correspond to the version control structure.

Also, in the early stages of analysis and design, the architecture of your model can change, sometimes drastically. During these early stages, modeling elements are often created, moved, and deleted. When a controlled unit is moved in the model, the corresponding controlled unit file is not automatically moved in the directory structure. If there is significant change, the directory structure can become fragmented, resulting in situations where controlled units that are logically grouped in the model will not be physically located in the same directory hierarchy.

Loading, Reloading, and Unloading Controlled Units

There are three ways to load controlled units:

- Open a model

If a model has controlled units, Rational Rose prompts you to determine whether to load all the subunits as the model opens. Clicking **Yes** loads all the controlled units associated with the model.

- Manually load units

Use **File > Units > Load** (or **Reload**) to individually load each controlled unit file that you require. When you point to a package in the browser, Rational Rose displays the controlled unit file name in the Status Bar, regardless if the controlled unit is not currently loaded.

Figure 22 Controlled Unit File Name

D:\RationalRoseModels\MyModels\classics\ecommerce.cat (write enabled)

If you double-click an unloaded package in the browser, Rational Rose loads that package.

Use **File > Units > Load** or **File > Import** to add controlled units from another source to your model.

- Loading a model workspace

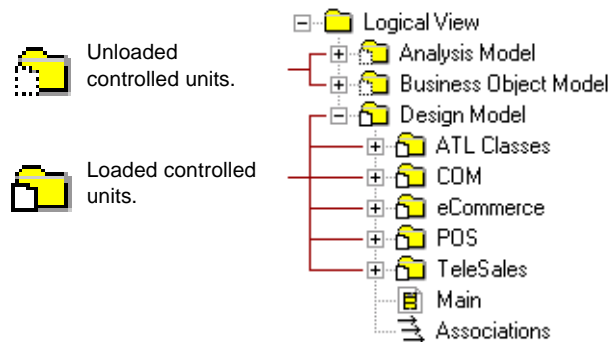
Rational Rose allows you to save your working environment; that is, the set of controlled units that you have loaded. For complete details about model workspaces, see *Creating and Using Model Workspaces* on page 61.

If your model is large, or you are planning to work on a few specific units, you can greatly reduce latency and resource consumption by manually loading individual controlled units, or by creating and then loading a model workspace.

To view or update a unit modified by another developer since you last loaded it, you must reload the unit by using **File > Units > Reload**.

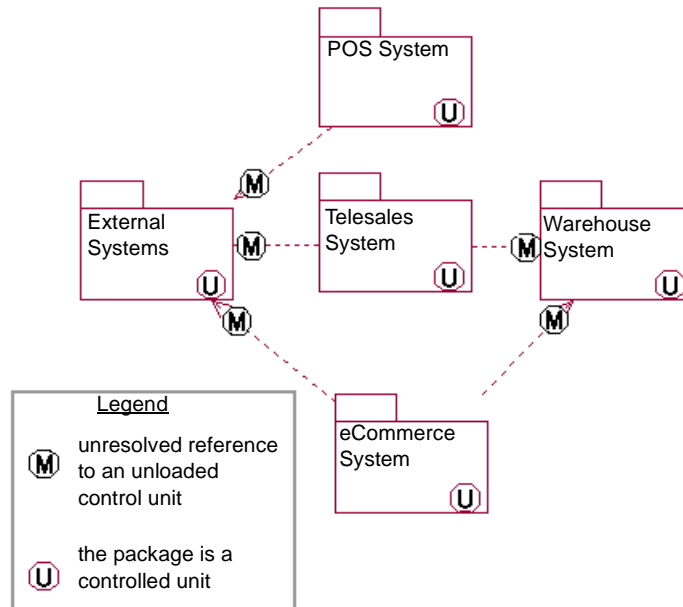
In Figure 23 Rational Rose uses the icons in its browser to distinguish between loaded and unloaded units.

Figure 23 Loaded and Unloaded Controlled Unit Icons



In diagrams, Rational Rose uses adornments to indicate which model elements are controlled units and whether there are unresolved references to unloaded units. You can enable or suppress adornments by clicking **Tools > Options > Diagram > Display**.

Figure 24 Adornments Indicating Controlled Units and Unresolved References



To unload a controlled unit, select the controlled unit in a diagram, then click **File > Units > Unload**.

Creating and Using Model Workspaces

Understanding Workspaces

A workspace is a snapshot of all currently loaded units and open diagrams. By defining one or more workspaces, you can set up your working environment in Rational Rose and return to that environment each time you are ready to work. When you load the workspace, Rational Rose restores the snapshot by loading the specified controlled units and opening the correct diagrams.

If you work with large models that are divided into many controlled units, you will notice even greater productivity gains by using workspaces to load predefined units and diagrams.

How a Saved Model Differs from a Model Workspace

A saved Rational Rose model contains the diagrams, elements, and controlled units that make up the complete model. A model workspace contains the actual state of open diagrams and controlled units for a specific saved model at a given point in time.

It is possible to have multiple workspaces that correspond to only one model. For example, during analysis and design, you can define one model workspace that displays the most important analysis diagrams and controlled units, and another model workspace for important design diagrams and controlled units. Each workspace is different but uses the same model.

Saving a model workspace does not affect how the model is loaded on another computer. If another team member loads a model using a model workspace that you defined on your computer, they must have a copy of the model and the model workspace in the same folder on their computer.

By default, Rational Rose names the workspace *<model name>-<Operating System User Name>.wsp*. For example, the name of a saved model workspace may be *MyModelName-SomeUser.wsp*.

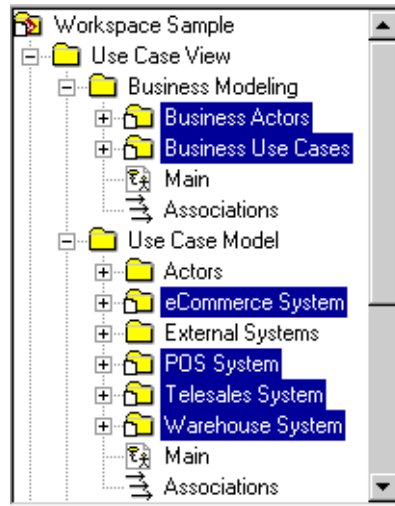
Note: Rational Rose stores all workspace files (*.wsp) in the workspaces folder of the Rational Rose installation directory.

Workspace Scenario

The following scenario shows how using model workspaces can benefit a team working on a large model.

A new software developer joins a distributed team working on a very large model containing over two hundred controlled units. Through the course of the next several months, the new developer will model several systems in the Use Case Model and modify the Business Actors and Use Cases (as shown in Figure 25). To help the new developer, the team's project manager creates a model workspace that loads all of the units the software developer is responsible for, as well as some of the more important diagrams.

Figure 25 Example Model Workspace



When the developer loads the model workspace, the Business Actors, Business Use Cases, eCommerce System, POS System, Telesales System, and Warehouse System controlled units load. The workspace configuration also displays some important class and activity diagrams in the diagram window.

The model workspace helps the new developer by:

- Automatically loading the controlled units that the developer is responsible for.
- Displaying some of the more important diagrams the developer should examine first.
- Saving the developer time because Rational Rose only has to load a subset (six) of the controlled units.
- Eliminating confusion by limiting the scope of information the developer sees.

After working in the model, the developer can easily customize the model workspace the project manager created, or create additional model workspaces for greater efficiency.

Creating and Saving a Model Workspace

To create a model workspace:

- 1 Load all controlled units that you will want to restore when loading this workspace.
- 2 Open all diagrams to restore when loading this workspace.
- 3 Click **File > Save Workspace**.

Name the model workspace file in the **Save As** dialog box. The default model workspace name is *<model name><Operating System Use Name>.wsp*.

Rational Rose stores all model workspace files (*.wsp) in the Rational Rose workspaces folder.

Loading a Model Workspace

To load a model workspace, click **File > Load Model Workspace**. Select the name of workspace file (*.wsp) to load.

Protecting Controlled Units

When loading a controlled unit into a model, Rational Rose makes the unit write-protected or write-enabled depending on the current status of the file in the file system. A controlled unit, which is read only in the file system, is write-protected in the model and Rational Rose prevents you from modifying it. This means that the toolbox is dimmed on all of its diagrams and you cannot update the Specifications of the contained model elements.

Note: If a write-protected controlled unit contains other controlled units, the write-protection is not extended to the contained controlled units.

A unit's write protection status displays in the Rational Rose Status Bar when selecting the controlled unit in the browser. For example, the status of the controlled units in Figure 26 is write-enabled.

Figure 26 Write-Protected Control Unit

A screenshot of the Rational Rose Status Bar. It shows a text box containing the file path "D:\ordersystem\units\user_serv.cat" followed by "(write enabled)" in parentheses. The text is enclosed in a thin black border.

If you use a version control add-in, Rational Rose handles the write-protection of the controlled unit automatically. This means that a checked-in unit is automatically write-protected.

There may be situations when you want to write-protect or write-enable a controlled unit manually from within Rational Rose (even if the controlled unit is under version control). For example, if:

- You want to load a checked-in unit, modify it, and save the result to a new file. To do this, you must manually write-enable the unit after loading.
- You have loaded a checked-out unit with the intention of browsing rather than modifying the unit. Manually write-protecting the unit assures that you do not inadvertently change the unit. After you reload the model, write-protection no longer applies, and you can edit the file.

You can change a unit's write-protection manually from within Rational Rose by using **File > Units > Write Protect/Write Enable**.

Write-Protecting a Controlled Unit

There are three ways to write-protect a controlled unit:

- If you use a version control system, place the controlled unit under version control and check it in.
- Make the file read-only by setting the file protection of the .cat file to read-only in the underlying file system.
- Right-click on the controlled unit in the browser, and click **Write-Protect** on the **Units** menu.

Write-Enabling a Controlled Unit

To write-enable a controlled unit under version control, click **Tools > Version Control > Check Out**. This will check out the unit and allow you to edit it.

Splitting a Controlled Unit

If a controlled unit becomes too large or if several team members often need to update a unit at the same time, you can split the unit. There are two ways to split a controlled unit:

- Remove the unit and split it into two different units.
- Keep the unit, but divide its contents into two new sub-units.

To split a controlled unit into two units, ensure that the controlled unit is write-enabled. If it is under version control, you must check out the file. If you want the original unit to constitute one of the new units, create one new package at the same level as the controlled unit. Otherwise, create two new packages.

Move the contents that belong to the new package (including the associations) from the unit into the new package by using drag-and-drop in the browser. Or, move the contents by selecting an element in a diagram, copying the element to the Clipboard, pasting it into a diagram that is owned by the new package, and then clicking **Relocate** on the **Edit** menu. Designate the new package(s) as a controlled unit.

Note: If you move classes from one package to another, the dependencies and generalizations move; the associations do not move. You must move the associations manually.

To divide the contents of a controlled unit into two sub-units, ensure that the controlled unit is write-enabled. If it is under version control, you must check out the file.

Create two new packages in the package that corresponds to the controlled unit, then move the contents from the unit into the two packages by using drag-and-drop in the browser. You must manually move any associations as well. Designate the new packages as controlled units.

Merging Controlled Units

You can display the differences between, and merge two versions of a controlled unit by using Model Integrator on the **Tools** menu.

For more information, see Chapter 5, *Comparing and Merging Models* on page 77.

Adding Controlled Units to a Model (Importing/Loading)

The controlled units you create can be imported or loaded into other models. Importing and loading adds a reference to a controlled unit in the model; it does not make a copy of the controlled unit. The imported controlled units can be edited in both models; changes made in one model are visible in the other model. As with any file, the controlled unit cannot be open simultaneously in two different models.

Rational Rose imports controlled units to diagrams that currently have focus control; therefore, ensure you are in the diagram to which you want to import the controlled units. The import will fail if you import to the wrong type of diagram. For example, you cannot import .cat files to a deployment diagram.

When you import a controlled unit, Rational Rose attempts to resolve any references. If an element has a reference that Rational Rose cannot resolve, the problem is logged.

Before you import a controlled unit, ensure that the destination model file or enclosing controlled unit is write-enabled. If under version control, you must check out these files.

To import a controlled unit, click **File > Import**. In the dialog box, select *.cat or *.sub, or *.* as the file type. Do not import a .ptl file. These are exported model files. When you import them, they replace the contents of your model.

Uncontrolling Controlled Units

Uncontrolling a controlled unit incorporates the contents into the model file or into the enclosing controlled unit if the unit to uncontrol is contained within another unit. After uncontrolling a unit, the enclosing file will no longer reference that unit's file. Instead, the content of the uncontrolled unit's file is inserted into the enclosing file.

Before you uncontrol a unit, ensure that the model file (or the enclosing controlled unit) is write-enabled. If it is under version control, check out the file.

If you use a version control add-in, select the model element and click **Tools > Version Control > Remove from Version Control**. The contents of the unit are now incorporated into the corresponding package in the model, and the file is removed from version control.

If you do not use version control, right-click on the package in the browser and click **Uncontrol** on the **Units** menu. The contents of the unit are incorporated into the corresponding package in the model, but the file continues to exist in the file system.

Save the model (or enclosing controlled unit), which now holds the contents of the unit file.

Creating Virtual Paths to Controlled Units

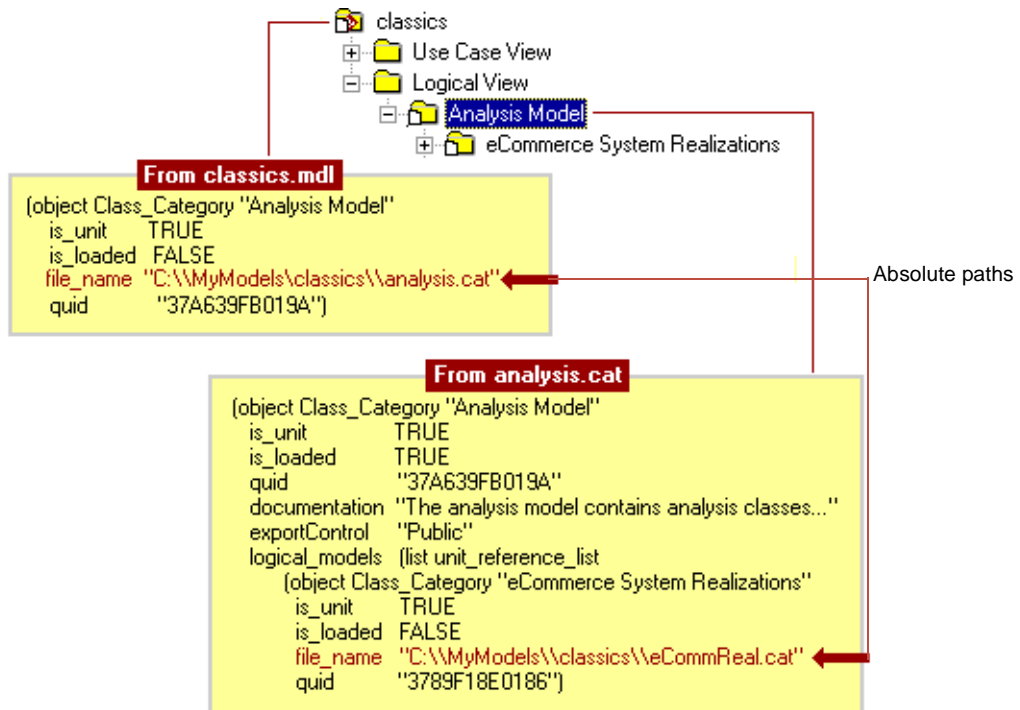
Understanding Virtual Path Maps

Rational Rose allows you to define symbolic names for file paths. Each user can control how these symbolic or virtual names are defined in their own workspace.

Path maps are essential for working in teams, especially where all users cannot work in exactly the same directory on their local machines. Using path maps allows model files to be distributed and relocated.

When you create controlled units, the enclosing model file or controlled unit stores the reference to the new unit as a file path. In the sample illustration that follows, the `classics.mdl` file contains the path to `analysis.cat`, the Analysis Model controlled unit file. In the illustration, the text is actually the contents of the `.mdl` and `.cat` files when opened in a text editor. The `analysis.cat` file contains a path to `eCommReal.cat`, the controlled unit for the eCommerce System Realizations package which Analysis Model encloses.

Figure 27 Virtual Path Maps



By defining virtual path maps, you substitute absolute paths with virtual paths. This allows you to move models and controlled units between different folder structures, and to update them from different workspaces.

How Virtual Paths Work

When Rational Rose reads from or writes to a model, it attempts to substitute every absolute path with a virtual path. When Rational Rose opens a controlled unit, or uses a path specified in a model property, it converts each virtual path to an absolute path.

For example, if a user has defined a virtual path,

```
$MYPATH=Z:\MyModels\classics
```

and saves a package as

```
Z:\MyModels\classics\analysis.cat
```

the model file will refer to the package as

```
$MYPATH\analysis.cat
```

When another user, who has defined \$MYPATH as

```
$MYPATH=X:\MyModels\classics
```

opens the same model from their “X” drive, Rational Rose resolves the internal reference to the controlled unit and loads the following file:

```
X:\MyModels\classics\analysis.cat
```

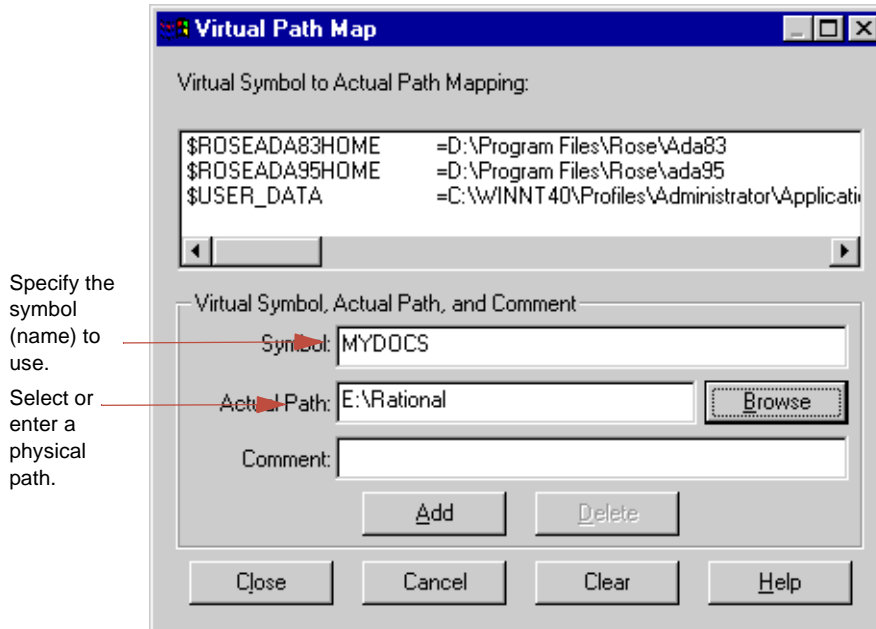
After you create virtual path maps, when anyone on the team opens or saves a model, Rational Rose attempts to match the longest possible file path to the symbols in the path map, and will continue until you have concatenated path map symbols.

Note: Each user working on a particular model must define the same path map symbols before opening the model. For example, a user with the private workspace called Y:\MyModels, must define \$MYPATH=Y:\MyModels. We recommended that you do not use path maps to point to network drives or shared files.

Creating Virtual Path Maps

You use **File > Edit Path Map** to open the **Virtual Path Map** dialog box.

Figure 28 Virtual Path Map Dialog Box



Do not enter the **\$** before the name in the **Symbol** box; it is automatically added to the name. When you click **Add**, the new path map is added to the list of existing path maps at the top of the dialog box.

Defining a Path Map Relative to the Location of the Model File

A leading “&” on a path name indicates that the path is relative to the model file or the enclosing controlled unit (if any). For example, if you create a model:

```
X:\MyModels\classics.mdl
```

and a controlled unit:

```
X:\MyModels\units\analysis.cat.
```

To allow different users to open the model and load the unit in different locations, each user can create a path map:

```
$CURDIR=&.
```

When the model saves, the reference from the model file to the package is stored as:

```
$CURDIR\units\data_serv.cat
```

When the model opens in another location, \$CURDIR is expanded to the physical path to the model in that specific workspace, for example:

```
Z:\ordersystem.
```

Note: The “&” requires that the controlled units be located in the same directory as the model file or in a subdirectory of the model file.

Defining a New Path Map Using Another Path Map Symbol

The actual path in a path map definition can contain existing path map symbols. For example, if there is a path map, \$ROOT=X:\model_vob, you can define a path map for the path X:\model_vob\MyModels by adding the path map \$MYPATH=\$ROOT\MyModels.

Defining a Path Map with Wildcards

You can use a wildcard character (*) in the path map to parameterize a virtual path. For example, if the following virtual path is defined:

```
$SUBSYSTEM=\\server\models\project\*\fred
```

and each user working on “project” has their own set of model files within each subsystem, then a controlled unit belonging to the display subsystem can have the following path:

```
\\server\models\project\display\fred\diagrams.cat
```


The model file refers to the unit as:

```
$SUBSYSTEM(display)/diagrams.cat
```

When the model is opened by user “suzanne,” who has the following virtual path definition:

```
$SUBSYSTEM=\\server\models\project\*\suzanne
```

the virtual path reference to the unit is converted back to the actual path:

```
\\server\models\project\display\suzanne\diagrams.cat
```

This allows different users to work on the same files with the same contents but in different folders without having to define a virtual path symbol for each such folder.

The slashes you use to define a path map are not literal, meaning that Rational Rose substitutes the correct format for Windows or Unix platforms.

Using Virtual Paths for the Value of a Model Property

Rational Rose does not convert actual paths in model properties to virtual paths. To use a virtual path in the value of a model property, you must manually enter the virtual path map symbol, including the “\$” sign (for example, \$CURDIR) into the value of the model property.

Using Path Maps for Other Artifacts

In addition to using path maps for model and controlled unit files, you can use them for any artifacts attached to your model, such as documents, code, and URLs.

It is strongly recommended that you maintain one path map for all model artifacts, including model and controlled unit files. However, if that is not possible, create separate path maps for each directory structure.

The easiest way to use path maps for artifacts other than model and controlled unit files is to create the path map *before* you attach the artifact. When you do this, Rational Rose automatically converts the absolute path to a virtual path when you attach the artifact and save it.

For example, suppose you created the path map:

```
$MYDOCS =E:\Rational
```

When you attach the file test.doc that resides in your E:\Rational directory to the Analysis Model package in a Rose model, the following virtual path is added to the analysis.cat file (the controlled unit for the Analysis Model package):

```
external_docs (list external_doc_list
  (object external_doc
    external_doc_path "$MYDOCS\\test.doc"))
```

If you attach an artifact before you create a path map, Rational Rose does not automatically convert the absolute path to a virtual path when you save your model. However, it does automatically do the conversion for controlled units and .mdl files.

Alternatively, you can move the artifact to another part of your model, immediately move it back to its appropriate location, then save the model. There is no need to save the model when the artifact is in its “temporary” location.

Similarly, if you delete or change a path map, you need to perform these same actions for Rational Rose to register the change.

Where Virtual Path Maps are Stored

Virtual path maps are stored in two locations in your system registry: the users area and the system area. A user can typically see and access only the virtual path maps in their specific area of the registry.

There are also system virtual path maps that are in HKEY_LOCAL_MACHINE. You need to be an administrator to edit these virtual path maps on a specific computer.

Checking References and Access Violations

After you create controlled units and unit ownership becomes distributed, it becomes increasingly important to check the integrity of your model. There are two ways to do this:

- By using Check Model.
- By using Show Access Violations.

Check Model

Check Model (**Tools > Check Model**) scans the entire model looking for unresolved references and places the results into the log. You can use this feature when you save your model to multiple controlled units, to ensure that all the units are consistent with one another. This is especially useful where parallel development occurs in multiple controlled units, since it is possible for different units to get out of synch with one another.

In a model where one item holds a reference to another item, it is possible that a reference exists, no item in the model of the right kind or with the right name. In that instance, the reference is unresolved.

Check Model checks the reference:

- To the supplier, such as any kind of dependency, generalization, association, realizes, instantiation.
- From a view on a diagram to an item in the model.
- From a logical package to its assigned component package, and from a module to its assigned class.
- From an object to its class.
- From a message on an object diagram to an operation in a class.
- From dynamic semantics in an operation to a scenario diagram.

Show Access Violations

Show Access Violations (**Report > Show Access Violations**) provides a list of all access violations between packages in a model.

As projects grow larger, access violations become more important

Show Access Violations is the primary tool for verifying that a large project is maintaining its design architecture.

An access violation occurs when a class in one package references a class in another package without an import relationship between the two packages. The import relationship is a dependency between the two packages. The direction of the dependency must be the same as the direction of the relationship between the classes or interfaces.

An access violation will also occur when a package references a class from another package whose export control is not set to Public. In this case, the presence of an import relationship between the two packages has no bearing. All references to non-public classes from different packages are cited as violations.

Import (dependency between packages) is not transitive, so if package A imports package B, which imports package C, then package A is not importing package C. A would have to have import package C separately.

Also, a package that has a nested package automatically gets visibility to the nested package. The inner package does not have visibility to its parent. Any package that imports the parent does not get visibility to the nested packages.

Violations are displayed in a dialog box. You can locate the diagram and element where the violation occurs by selecting the violation from the dialog box and clicking **Browse**.

Organizing Controlled Units for Teams

When sharing models among teams of developers, it is essential that the model be partitioned so that it can evolve in a controlled manner. To successfully share a model, you need to manage the dependencies between different portions of a model.

Ultimately, how many packages and controlled units to create becomes a question for the project leader or model architect, and the person responsible for configuration management in your project. The level of version control you use may define what becomes a controlled unit. For example, all packages can be controlled units, including nested packages. Doing so, provides the capability for two developers to check out packages that belong to the same higher level package.

Suggested Strategies

The following are strategies to consider when partitioning a model into controlled units:

- The model should be a shell with nothing but controlled units under the use-case, logical, and component views.
- Create design model, analysis model, and business model controlled units under the logical view.
- Create an implementation model controlled unit under component view.
- Consider separating actors and use-case controlled units.
- Also consider separate controlled units for each use-case.
- Prevent your use-case controlled units from including any diagrams that describe internal system operations or structure, such as class or interaction diagrams.
- Under the design model and analysis model packages, provide a use case realizations controlled unit and provide a separate controlled unit for each realization.
- Class and interaction diagrams that describe system internals should go with the use case realizations.
- Describe the system structure using a series of nested packages that become controlled units.

- Layers and global packages should be at the top level of nesting.
- Maintain interfaces in separate controlled units.
- Describe each significant mechanism in its own controlled unit.
- Control dependencies. Create UML subsystems by using packages that provide discrete, well-defined services.
- Subsystems should expose services only via UML interfaces - they provide strong separation between major portions of the model.
- Subsystem internals should depend only on the interfaces that are offered by other subsystems.
- Developers sometimes define class-level relations that violate dependencies between packages and subsystems. To detect this in a model, click **Report > Show Access Violations**.

For more details about model architecture, see *Establishing a Model Architecture and Process for Team Development*.

Contents

This chapter is organized as follows:

- *About the Model Integrator* on page 77
- *Model Integrator and ClearCase* on page 86
- *Comparing and Merging Models* on page 87
- *Performing a Partial Merge* on page 105
- *Merging Models Without a Base Model* on page 106
- *Viewing a Single Model File* on page 107
- *Using Model Integrator from the Command Line* on page 107

About the Model Integrator

Model Integrator is a tool for comparing and merging Rational Rose models. Model Integrator lets you compare model elements from up to seven contributor files, discover their differences, and merge them into a recipient model.

For example, two developers may need to modify a shared model at the same time. They can each copy the model, modify it separately, and then use Model Integrator to merge their changes back into a single shared copy of the model. Or they can use Model Integrator to compare their models and identify the differences between them.

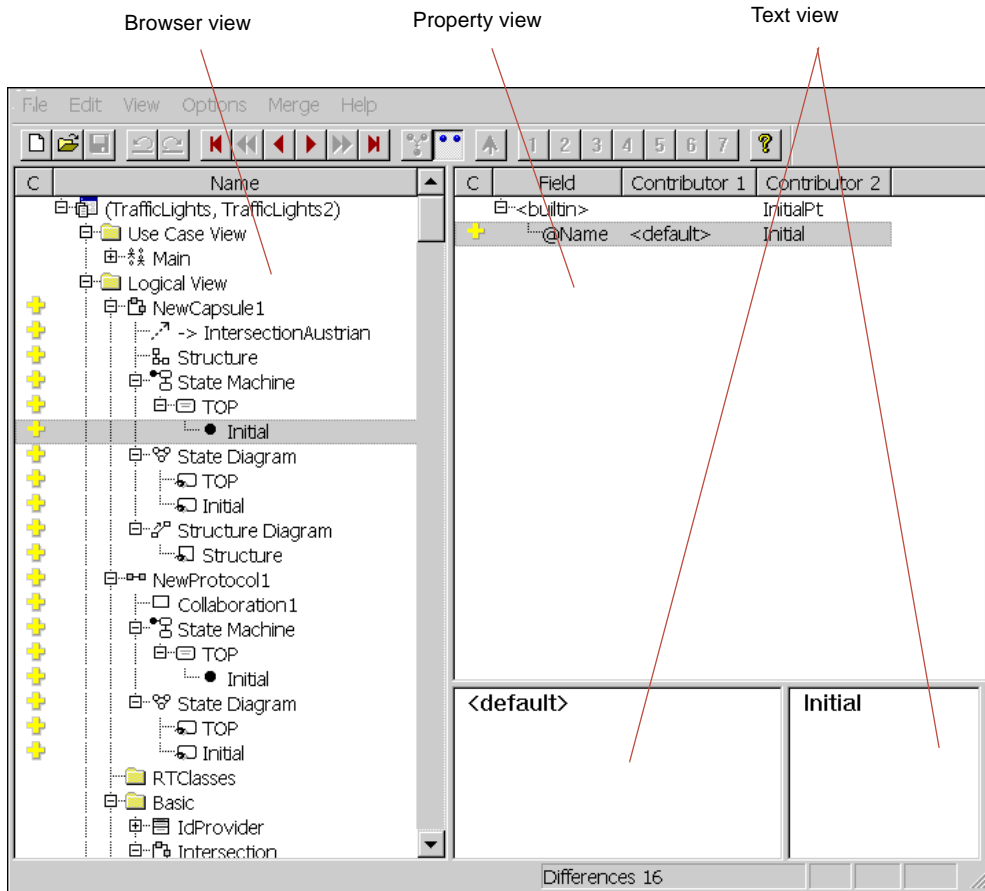
You can also use Model Integrator to view the contents of a single model file. Model Integrator provides a method of looking at a model that is different from the view provided by Rational Rose. Model Integrator provides a low-level textual view of all model elements and their properties. This method of examining a model allows you to view all property settings in use.

Model Integrator runs in two modes: Compare mode and Merge mode. As described later in this chapter, you can switch between modes.

Model Integrator Interface

Model Integrator runs outside of Rational Rose and provides its own interface as shown in Figure 29.

Figure 29 Model Integrator Graphical User Interface



There are three major components to the interface: browser view, property view, and text view.

Browser View

The left window pane is called the browser view. In this window, the primary objects that make up the model are displayed in a hierarchical tree structure similar to that used in Rational Rose. However, the objects displayed in the browser view are not

identical to those displayed in Rational Rose. Model Integrator displays some objects that Rational Rose hides from your view. See *Model Files and Model Integrator* on page 82 for a brief discussion of the objects Model Integrator displays.

The browser view displays only a single view of the model hierarchy, even though there are several models loaded. The browser view shows all of the objects from all of the contributing models, but it attempts to partner objects that are the same across all the models. If all of the contributors have the same model element in the same location, the browser only displays a single entry for that node of the model.

If different contributors have the same model element located in different places in the model, there is a node in the browser view for each location where the model element exists in the merged model. However, only one of these locations will be written to the final merged output model file (you will decide which one when you resolve the conflict at that node).

On the left side of the browser window are icons that display the results of comparing and, in merge mode, of merging the models. The meaning of these icons is discussed in *Interpreting Compare and Merge Results* on page 94.

Property View

The upper right window pane is called the property view. This window displays the set of properties that belong to the currently selected object in the browser view. In this view, there is a column for each contributor and a column for the recipient model (in merge mode). There is also a column of icons to help you see the comparison state of the properties provided by the different contributors. These icons are the same as the comparison icons mentioned above.

Text View

These windows along the lower right side of the main window display the values from each contributor for the property currently selected in the property view. In merge mode, the left-most text view displays the value for the recipient model, with the other contributors following it to the right in numerical order. These windows are for viewing purposes only. You cannot change the values displayed there.

Other Interface Features

The toolbar makes some commonly used functions available as buttons. All of these functions are also available in the menus. When you position the cursor over the icons in the browser view, they display a message explaining the compare or merge state. At the bottom of the screen, a status bar displays the merge status of the node currently selected in the browser view.

Contributors

Contributors are the models that form the input to Model Integrator. Model Integrator accepts up to seven contributor models for merging. All contributors must be of the same type; you cannot compare a .mdl to a .sub file, for example. A contributor can be any of the following:

- A model file, with or without its associated controlled units (subunits). If you specify a model file (*.mdl) as the contributor, and the model has subunits, Model Integrator prompts you to load its subunits.
- A controlled unit of a model.

You can specify a single controlled unit (a .cat, .sub, .pty, .prc, or .prp file) as the contributor. Controlled units are also referred to as subunits.

The first contributor, called Contributor 1, has special significance to Model Integrator; it is the base model used for comparing the differences between the other models.

Base Model

The base model is the model that is the ancestor to all of the other contributor models being merged. That is, the base model is the version of the model that existed before any changes were made. The base model must always be specified as Contributor 1.

Comparing Models

Compare mode in Model Integrator highlights the differences and conflicts between two or more models. You can switch back and forth between Compare mode and Merge mode; you can begin a work session in Compare mode and then switch to Merge mode if you decide to merge the models. In Compare mode, you cannot make any changes to the model, and the **Merge** menu and toolbar functions appear dimmed.

Merging Models

Merge mode incorporates all of the features of Compare mode, along with additional information to support the decisions you need to make to successfully merge model files. Model Integrator supports two types of merge functionality:

- **Automatic Merge:** Model Integrator merges all changes that do not produce conflicts.
- **Selective Merge:** Allows the user to optionally choose the contributor for each difference identified by Model Integrator between the models to merge.

Automatic merge takes effect when Model Integrator first enters Merge mode. It creates a recipient model and automatically merges all unchanged or trivially changed nodes into the recipient model for you. (A node is another name for an object in the model hierarchy. Examples of nodes are classes, use cases, objects, operations, components, and diagrams.) If the merged model has nodes that have conflicts, Model Integrator displays an icon at the location of the conflict in the browser window. As you make choices to resolve these conflicts, Model Integrator shows you the results of your merge.

The selective merge feature lets you change the contributor at nodes that have differences as well as conflicts. This can be useful when you do not want to accept all the changes that a contributor makes to your model. It is also useful when you need to correct more complicated errors such as those discovered by the semantic checking functions.

Model Integrator merges models that have a common ancestor (the base model). This is necessary when you keep your model under version control, and when two or more people modify the model at the same time. However, Model Integrator also supports merging models that do not have a base model.

Differences and Conflicts

Model Integrator uses the base model, called Contributor 1, to identify the types of changes made to the models being compared or merged. Each contributor is first compared to the base model. Model Integrator displays additions, changes, and deletions between a contributor and its base model as differences. Symbols identify the types of differences found. (These symbols are displayed in the C column of both the browser view and the property view.)

In compare mode, Model Integrator only displays differences; but in merge mode, Model Integrator also displays conflicts. A conflict occurs when there are two or more differences at the same node of the model. When Model Integrator finds a conflict, it cannot tell which of the different contributors to incorporate into the recipient model. (Conflicts are displayed in the M column of the browser window, along with other status information about the merge.)

In Merge mode, Model Integrator automatically incorporates differences into the recipient model. However, you must resolve conflicts by selecting the contributor from which to accept changes.

Model Integrator also supports comparing and merging models without using a base model as a reference point. However, in this mode, every node of the model displays as a difference. Conflicts continue to have the same meaning in this mode.

Model Files and Model Integrator

A Rational Rose model consists of a set of objects (also called model elements, items, or nodes), where each object has its own set of properties that define attributes of the object. Model Integrator exposes to your view all of the objects and properties defined in the models you are merging. This way of looking at the model is considerably different from the normal graphical presentation of the model in Rational Rose. The following is a brief introduction to the kinds of objects that Rational Rose models contain.

Basic Objects

The objects you are most concerned with when you create the model are those that represent elements in your application such as actors and classes.

Diagram Objects

Each diagram in your model is an object. The diagrams display differently in Model Integrator than in Rational Rose. The diagram titles are the same in the browser window, but the diagrams are not shown as pictures. They are shown as lists of their component objects. Some of these components you are already familiar with, such as Labels. Others are new because Rational Rose does not typically display them. These objects include view objects.

View Objects

Each basic object that appears in a diagram is represented by a view object when it appears in a diagram. For example, when a class appears on a diagram, the diagram object will have as a child a ClassView object for that class, and so on for every kind of basic object. Other view objects exist for items that are part of a mechanism.

Mechanism

A Mechanism is hidden component of a model that contains a set of objects used internally to implement parts of the model you created. A mechanism will contain more objects as children.

Quids

A quid is a unique identifying number that distinguishes the object it is attached to regardless of the object's name. A quid property is generated by Rational Rose for each object created in the model. quids are unique, so that they can be used to identify an object when the name of the object changes, or the object is moved in the model. Model Integrator uses quids extensively to determine whether objects are the same; if the quid is the same, then those objects have a common ancestry.

References

Much of the power of a Rational Rose model comes from the relationships that exist between objects. These relationships are identified by reference properties (or just references), based on quids, that enable one object to point to another one. A given object in a model may have no references at all or it may have many. Reference properties have names; common names are client, supplier, and quidu. Model Integrator provides the command **View > Referenced Nodes** that allows you to follow these references to view the model element that lies at the other end of the reference.

It is essential to maintain valid references between the objects in the model after a merge completes. When objects are deleted or moved, Model Integrator verifies that references from other objects remain valid. This semantic checking function is performed before the model saves.

Unnamed Objects

Virtually every object in a Rational Rose model has a unique name. You are not required to name every object that you create. For objects that you do not name, Rational Rose creates a name of the form \$UNNAMED\$*nn*, where *nn* is a number.

Often a model will contain many unnamed objects that you are not aware of because Rational Rose does not display the \$UNNAMED\$ string. Model Integrator displays the actual name of every object, including unnamed objects. You can determine an object's type by its corresponding icon in the browser view, its properties (the object type is at the top of the property view), and, in some cases, by looking at the children of the object.

Rational Rose Model File Versions

Each new version of Rational Rose contains new or improved features that must be represented in the model files produced by that version. This leads to model files having their own versions. You can see the model file version information listed in the property view of the first node of the model in the browser view, under the @Petal property.

It is good practice to only merge models that have the same model file version numbers. This avoids problems encountered when creating merged models that declare themselves to be one version, but contain model elements (accepted from contributors) that may be incomplete or different from the expected version. Model Integrator itself is independent of model file versions, and does not know how to bring old model files up to date.

Understanding Semantic Checking

Semantic checking is a merge mode feature that helps to ensure that the merge choices you make are valid. There are two forms of semantic checking available in Model Integrator. The first is performed by the **Check Merge** function.

This function is called automatically before a merged model is saved. It cross-references all of the nodes of the recipient model to ensure that the final result is complete.

The second form of semantic checking is an optional, real-time version of the **Check Merge** function. This function checks references on the nodes as you access them, and disables merge choices that may introduce errors into the model.

For example, your base model contains class A. Contributors 2 and 3 make changes to one member of this class, while contributor 4 deletes the class. If you already accepted changes from contributor 4 to delete the class, it does not make sense to allow you to accept the changes that contributors 2 or 3 made to the class. However, with semantic checking turned off, Model Integrator allows you to make these contradictory changes. Model Integrator does not discover the problem until either you save the recipient, or you use the Check Merge function to verify the model.

This example is very simple and probably would not be a problem in practice. But in a large, complicated model, it can be difficult to determine which contributors present valid choices at a particular node of the model. This problem can also arise when Model Integrator makes automatic merge choices at nodes that do not have conflicts. If a node is deleted automatically, you may not be aware of that fact when you are viewing a conflict at one of its dependent nodes. Semantic checking helps you avoid these problems by making your choices clearer at each step of the merge process.

When semantic checking is activated, and the user moves the current selection to a new node of the model tree, the checker determines which choice of contributor (if any) would result in an invalid model if chosen by the user. These choices are then disabled in the interface by dimming the appropriate menu items and Toolbar buttons.

When working with a very large model, you may not want the overhead of semantic checking. Or, you may want to make the change now and fix it later. In this case, semantic checking can be disabled. Merge choices can then be made in the normal manner. After making the merge choices, the user can then select the **Merge > Check Merge** menu function to check and repair the model. The model is always checked for validity before it is saved.

Limitations of Semantic Checking

For performance reasons, semantic checking only checks the nodes of the model you are currently viewing, and only when you view them. Consequently, it is necessary to perform a check of the entire model before saving, and this check may reveal errors that need to be corrected.

For both types of semantic checking, references to subunits that are not loaded into the merge session are not checked.

Memory Requirements and Performance

For a typical merge operation, Model Integrator must load the models and then compile additional information from the loaded models to compare and merge them. This requires an amount of memory proportionate to both the number and the size of the contributors.

The exact proportion varies, but a good estimate of the maximum amount required is to take the sum of the sizes of the model files you are merging and multiply that number by 5 to get the amount of memory Model Integrator will need to complete the merge operation. This memory is in addition to that used by your operating system and other programs you may be using.

If your models are small, memory is not a problem. If you have large models to merge, such a 30 megabyte (MB) set of models, this may be a strain on your system resources. A typical sign of a serious memory deficiency is that loading the models is extremely slow (Model Integrator may also appear to be frozen) while the disk drive is constantly busy. This condition is known as thrashing. It occurs because Model Integrator requires access to the entire data set for all the models you want to merge, but the physical memory shortage results in much of this data being stored in virtual memory on your hard disk (in your computer's pagefile or swap file, depending on which operating system you use). The computer devotes much of its resources on reading and writing to the disk, without completing the merge. If your virtual memory configuration is also insufficient, your computer may need to be rebooted in order to recover.

Here are some tips on how to improve the performance of Model Integrator:

- Configure your computer with enough RAM to meet or exceed the 5x requirement stated above. For example, if you have 30 MB of models to merge, you should have at least 150 MB of RAM in your computer. Anything less will compromise performance, as Model Integrator has to store its data on the disk.
- If there is not enough physical memory to meet the requirements for Model Integrator, ensure that you allocate enough virtual memory. Consult your operating system documentation or ask a system administrator to adjust the available virtual memory.
- Close other programs to free up memory. If you have a lot of RAM and virtual memory in your computer, other programs may claim large portions of it. In some extreme cases, applications may load system components that are not unloaded when the application exits. If you continue to have problems, you may want to try running Model Integrator after rebooting your computer and before running other applications.
- Use the tools that come with your operating system to measure and report memory usage. For example, in the Windows 2000 environment, you can use the Task Manager and its Performance page to report on you system's memory usage.

Model Integrator and ClearCase

Model Integrator is designed to work with Rational ClearCase to allow you to compare and merge individual model files from within the ClearCase environment. You can use the standard ClearCase tools, such as the Version Tree Browser or the ClearCase context menus in Windows Explorer, to compare model file versions and merge branched versions of models.

For example, you can right-click a model file version displayed in the ClearCase Version Tree Browser window and select **Compare > with Previous Version** from the shortcut menu. ClearCase will invoke Model Integrator to display the differences. Or, from Windows Explorer you can right-click a model file in a ClearCase view and select **ClearCase > Compare with Previous Version** to accomplish the same thing.

If you select one of the above compare commands and you do not see the models displayed within Model Integrator, it is likely that the ClearCase integration with Rational Rose has not been set up. See Chapter 6 for instructions.

Merging Whole Models with Controlled Subunits

ClearCase and Model Integrator only support comparing and merging individual model files or controlled units directly from ClearCase. Often this works well in a team environment because modelers are only working on individual component files of the model.

For example, use cases can be divided into categories so developers only have to check out the .cat file that contains their use cases. These files can be privately branched and subsequently merged back into the main development branch without having to merge the entire model.

However, it can be desirable to merge the entire model, because semantic checking works best when the whole model is loaded into Model Integrator. This is done by constructing a separate ClearCase view for each full contributor to the merge session. Each view is constructed to make the correct version of the model files for that contributor visible within the view. The model files are checked out for writing in the view that will receive the merge result.

Starting Model Integrator in a ClearCase Integration

Model Integrator is started, not from a ClearCase menu, but from the Rational Rose **Tools** menu or by the standard method for the system you are using, such as the **Start** menu in Windows. The merge session proceeds in the same way it would if ClearCase were not involved. When completed, the merged model files are saved and checked back into ClearCase.

Comparing and Merging Models

Starting Model Integrator

To start Model Integrator, you can do one of the following:

- From within Rational Rose, select **Tools > Model Integrator**.
- From a Unix shell process or a Windows console process, type **modelint file.mdl** and press **Return**.

For more information about the command line interface, see *Using Model Integrator from the Command Line* on page 107. To run Model Integrator from the command line, the directory containing the Model Integrator executable must be in your path variable.

- You can also start Model Integrator from Rational ClearCase as part of a ClearCase compare or merge operation. See *Model Integrator and ClearCase* on page 86.

Preparing Models for Merging

Before merging models, it is good practice to check each model with the Rational Rose **Tools > Check Model**. If errors are reported, correct the errors before performing a merge with Model Integrator.

Selecting the Contributors

An easy method of specifying contributor files is to drag and drop the files from Windows Explorer onto the Model Integrator window (Windows platform only). If the **Contributors** dialog box is not open, Model Integrator opens it for you. If it is already open, then you must drop the files onto the dialog box, not the main window.

Alternatively, select **File > Contributors** to display the **Contributors** dialog box. Then, follow these steps to specify the files to compare or merge:

- 1 Do one of the following to specify the first .mdl, .cat, .sub, .pty, .prc, or .prp file in the **Files** list.
 - Enter the fully qualified file name in the blank area of the **Files** list.
 - Click **Browse** at the top of the **Files** list control and use the file browser to find a file to add to the list.
- 2 Click **Enter** to confirm the file name.
- 3 Click **New** to create a new file input field.
- 4 Repeat steps 1 through 3 until all files are specified.
- 5 Click **Compare** or **Merge**.

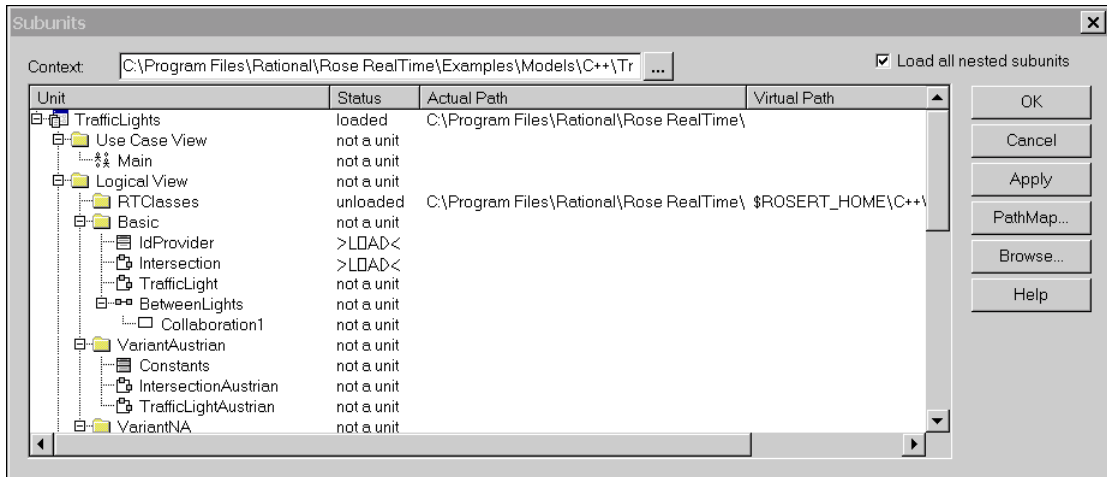
Note: If the **Compare/Merge Against Base Model** check box is selected, then the first specified file must be the base model. If the first file listed is not the base model, you can use the **arrow** buttons to change the order of filenames listed in the **Files** area so that the base model is listed first. Select one of the file names by clicking it, then click the **arrow** button to move it in the appropriate direction.

Model Integrator can provide a base model for you if you do not have one to use. See *Merging Models Without a Base Model* on page 106.

Loading or Unloading Controlled Units

If one or more of the contributor files you specify have controlled units, Model Integrator displays the **Subunits** dialog box. This dialog box allows you to load or not load (unload) those units before comparing or merging your files, and to save them again when you save the merged model.

Figure 30 Subunits Dialog Box



Subunit Status

The **Status** column displays the subunit status for each potential subunit in the model you load or save. The Status column can display four different values when loading subunits, or two values when saving.

Status	Description
loaded	Subunit was loaded successfully.
not a unit	Entry is not currently a separate subunit. This model section is part of the main .mdl file.
LOAD	Model Integrator loads the entry when you click OK or Apply .
SAVE	Model Integrator saves the entry to a separate file when you click OK .
unloaded	Subunit will not be loaded.

Loading Subunits

Subunits for each contributor are loaded separately, so a separate **Subunits** dialog box appears for each contributor .mdl file that has subunits. You can toggle the **Status** value between **LOAD** and **unloaded** by clicking the value with your left mouse button. By default, Model Integrator will try to load all subunits for a model. If there are units you do not want to load, click the **Status** value to change the status to "unload", and the subunit is skipped. If you do not want to load any subunits, click **Cancel**.

When you complete one dialog box and click **OK**, Model Integrator tries to load the subunits that have the **LOAD** Status value. If there is an error and some of the subunits cannot be loaded, the **Subunits** dialog box displays again.

Note: Model Integrator cannot perform reference checking for subunits that are not loaded.

Each contributor with subunits opens a corresponding **Subunits** dialog box. When you complete the final **Subunits** dialog box, Model Integrator immediately begins the Compare or Merge session.

Saving Subunits

When you save a model using **File > Save** or the **Save** button on the Toolbar, Model Integrator saves your subunits to the same place relative to the main .mdl file's location. In this case, the **Subunit** dialog box does not display. If you want to change the subunit configuration of your model, use **File > Save As**. When you save the merged model using this function, the **Subunits** dialog box displays. It allows you to:

- Save your existing subunits configuration by clicking **OK** in the dialog box.
- Create new subunits by clicking the **Status** column for the subunit you want to create. Model elements eligible to become subunits are displayed in the **Subunits** dialog box with the "not a unit" **Status** value. Click this value to change it to **SAVE**; when you click **OK** or **Apply**, a new subunit is created.
- Eliminate subunits by clicking the Status field and changing the **SAVE** value to "not a unit". When you click **OK**, this part of the model is saved in the main .mdl file instead of a separate subunit file.

Whether you use the **Subunits** dialog box or not, if you save subunits to a directory that already contains copies of the same subunits, Model Integrator warns you that you are overwriting the subunits and prompts you to continue. This is in addition to asking you if you want to overwrite the main model file. You can click **Yes**, **No**, or **Yes to All** to save your entire set of subunits with no more questions.

Subunit File and Path Names

The **Subunits** dialog box displays two columns of path-related information about the subunits in this model. The **Virtual Path** column shows the value of the path stored in the parent model. This value may be an absolute path or it may contain a virtual path map. The **Actual Path** column displays the path that Model Integrator is using to load the subunit.

If path map variables appear in the **Actual Path** column, click **PathMap** to set a value for the path map variable.

Model Integrator shares path map variables with Rational Rose and uses the same values transparently. However, Model Integrator may require you to enter a value for a path map variable if that variable has not previously been defined on the machine you are using. This situation is evident when you see a virtual path map variable listed in the **Actual Path** column of the **Subunits** dialog box.

You can left-click the value listed in the **Actual Path** column and directly edit the path name that Model Integrator uses to find the subunit.

When saving a subunit, we recommend that you define a path map variable (in the **PathMap** dialog box) and set it equal to the value “&”. This prevents absolute path names from being stored in the .mdl file for the subunits, which makes it easier to move the files to new storage locations in the future.

Resolving Subunit Loading Errors

If you instruct Model Integrator to load a subunit and the load process fails, Model Integrator displays the **Subunits** dialog box to allow you to correct the problem. The **Status** column of the dialog box shows you the current status of each subunit. You may need to scroll through the dialog box to find a subunit that was not loaded. It will continue to display the **LOAD** status.

Currently, you have several options to resolve the problem:

- You can directly edit the **Actual Path** field to change the path for that particular subunit as mentioned above.
- If the subunit uses a path map variable, you can change the value of the path map variable by clicking **PathMap** and modifying the variable in the **Pathmap** dialog box.
- You can first select the subunit in the list and then click **Browse** to open a directory browser window to find the file. Select a file and click **OK**. The filename appears in the **Subunit** dialog box.

- You can change the current directory for path map variables that take the value “&” by changing **Context** located at the top of the **Subunit** dialog box.
- You can elect to not load the subunit. Click the **Status** value for the subunit. The status changes to “unloaded”, and the subunit is not included in the merge.

Setting a New Context for Subunits

The **Context** box at the top of the **Subunits** dialog box shows the default path that Model Integrator uses to substitute for the “&” path map symbol (See *Understanding Virtual Path Maps* on page 67 for a description of how to use path map symbols).

If you created your models using a virtual path map, you can define the value of the symbol to be “&”, when Model Integrator encounters the “&” symbol in the definition of a path map, it replaces the symbol with the actual path specified in the **Context** box.

By default, the value of the **Context** box is the path where the main model file (*.mdl) is located. However, if you moved the files to a new location, you can change the **Context** value and Model Integrator will load the files from the new context.

You can select a new **Context** path by either entering a new value directly into the **Context** box, or by clicking **Browse** to the right of the field to locate the desired drive and folder.

Using Compare Mode

Use compare mode to scroll through the model and observe the differences between the contributors. If you want to merge the models, change the mode to merge mode or exit the program when you are done.

Using Merge Mode

In merge mode, Model Integrator has already tried to automatically merge the models for you. Your next step depends on the results of the automatic merge.

AutoMerge

When Model Integrator first enters merge mode, it applies the AutoMerge procedure to the entire set of contributors. The AutoMerge procedure follows the rules illustrated in Table 2 for a typical case of three contributors (not shown is a move operation, but it behaves like a change).

Table 2 AutoMerge Rules for Merging Models

AutoMerge State	Contributor 1 (Base)	Contributor 2	Contributor 3	Result
No change	A	A	A	A
Added	--	A	-	A
Changed	A	A	B	B
Deleted	A	A	-	-
Conflict	A	B	C	?
Conflict	A	B	-	?
Conflict	-	B	C	?

A, B, C are model elements. “-” means not present.

Note: Only the role of the base model is fixed in the AutoMerge procedure. The order of the other contributors does not matter. Swapping Contributor 2 and Contributor 3 does not affect the results.

If a contributor that is not the base model introduces a change (adds, modifies, moves, or deletes an object), that change is copied to the merged output instead of the original object. However, if two or more contributors change the same thing, then the AutoMerge procedure does not know how to decide which one to choose. Instead, it generates a conflict.

By default Model Integrator uses automatic merge to merge all changes that do not produce conflicts into your merged model. You can also use the **Merge > AutoMerge** command to reapply automatic merging to nodes of the model you have previously reverted using the **Merge > Revert** command.

In the bottom right corner of the main window, a message appears in the status bar saying “Unresolved items *nn*” where *nn* is a number. If the number of unresolved items is greater than zero, you must resolve these items before the merge can be completed. Use the **Forward** toolbar button to find the first conflict. Examine the contributors for this model element and accept your choice to resolve the conflict.

Interpreting Compare and Merge Results



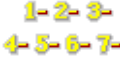
Model Integrator shows the results of comparing the contributing models by displaying an icon to the left of each node in the browser view in the C column. Table 3 depicts the Compare status icons and their meanings.

Table 3 Compare Status Icons

Symbol	Description
space	Common item (same values in all contributors).
	New item added by a single contributor. (difference)
	Item deleted in a single contributor. (difference)
	Item changed in a single contributor. (difference)
	Item moved to a new location in a single contributor. (difference)
	Item added by multiple contributors, each has different property values. (conflict)
	Item deleted in some contributor; item changed by another contributor. (conflict)
	Item changed in multiple contributors. (conflict)
	Item moved in some contributor; item changed by another contributor. (conflict)
	Item moved in some contributor; item deleted by another contributor. (conflict)
	Item moved to different locations by multiple contributors. (conflict)

Symbols indicating the status of a merge operation appear in the **M** column. Table 4 depicts the Merge status icons and their meanings.

Table 4 Merge Status Icons

Symbol	Description
space	Common item and recipient is set to the common property values.
	Recipient item is not set. This occurs for an unresolved conflict or after applying the Merge > Revert command.
	Recipient item is set with values from contributor <i>n</i> , where <i>n</i> is a number between 1 and 7.
	Recipient item is set to be deleted by contributor <i>n</i> , where <i>n</i> is a number between 1 and 7 and where, as indicated by the minus sign, this contributor has no values set for the selected item.

Note: Merge results do not appear in Compare mode.

Navigating Through a Model

Searching for a Model Element

To search for a particular node by its name in the browser window, select **Edit > Search**. Enter the search string, select the direction to search in, and click **Find**.

The search starts at your current location in the browser window and proceeds through all the nodes in the model that are displayed in the browser window. (Use **Edit > Expand All** to display every model object in the browser.) If the string is found, the browser window scrolls to display the desired node, and its properties are displayed in the property view. If the node found is not the desired one, click **Find Next** to continue the search from the current point. When the search reaches the last (first) node of the model, it will wrap back to the beginning (end) and continue searching. After the desired node is found, click **Cancel** to dismiss the search window.







You do not have to specify the entire name you want to find. Model Integrator performs the search by matching the string you enter against any part of the model element name. The search is not case-sensitive.

Viewing Conflicts and Differences

The **View** menu contains a number of options for navigating through conflicts and differences. These same commands also appear as buttons on the toolbar. Use these commands to speed your way through the merged model, ensure that you visit all the conflicts and differences. These commands automatically expand the browser tree to make the next conflict visible.

Some of these commands, as noted in Table 5, operate in the same mode as the setting for Auto Advance mode. The text displayed in the menus and tool tips changes to reflect this.

Table 5 Navigation Buttons for Viewing Conflicts and Differences

Button	Description
	First Difference Depends on the Auto Advance mode setting: <ul style="list-style-type: none">- Conflict: goes to the first conflict- Difference: goes to the first difference- None: goes to the first node of the model
	Previous Conflict Moves back to the previous conflict
	Previous Difference Depends on the Auto Advance mode setting: <ul style="list-style-type: none">- Difference: goes to the previous difference- None: goes to the previous node of the model
	Next Difference Depends on the Auto Advance mode setting: <ul style="list-style-type: none">- Difference: goes to the next difference- None: goes to the next conflict
	Next Conflict Moves to the next conflict
	Last Difference Depends on the Auto Advance mode setting: <ul style="list-style-type: none">- Conflict: goes to the last conflict- Difference: goes to the last difference- None: goes to the last node of the model

Viewing Conflicts and Differences with Auto Advance

The Auto Advance function automatically moves the current selection in the browser window after you have accepted a change. The function has three modes of operation:

- **Conflict:** advances to the next conflict
- **Differences:** advances to the next difference
- **None:** does not auto advance

You can change the Auto Advance setting by selecting your choice from the **Options > Auto Advance** menu.

The Auto Advance setting also affects the functioning of the commands for viewing conflicts and differences.

The Auto Advance function is set automatically when you load a set of models. If the model has conflicts, then the **Conflict** mode is set. If the model has no conflicts, but has differences, the **Differences** mode is set. If there are no conflicts or differences, the **None** mode is set.

Viewing Model Elements that have Moved

Model Integrator can detect when you move items from one place to another within your model (for example, by using drag and drop editing or by using the Clipboard within Rational Rose). When you merge models with elements that have been moved, Model Integrator displays all the locations where the model elements could be placed by the different contributors. However, you can only keep one of these locations in the merged file.

When you see one of the status icons indicating that an item has been moved, you can navigate between the different locations by clicking **View > Other Locations** menu item. Each time you select this function, it will cycle to the next location where a contributor has placed the model element you are viewing. If the model element has only one location, this function is dimmed.

You can click **View > Previous Location** to return to the node you were originally viewing.

Viewing the Parent of a Node

Every node in the model, except for the first one, has a parent node. Usually there is an important relationship between a node and its parent. For example, the parent of a State node is a State Machine.

While merging models you may need to view the parent of a node currently displayed, but if the model is large, the parent node may not be visible on the screen. Click **View > Parent** to bring the parent node into view. Click **View > Previous Location** to return to the node you were viewing previously.

View Nodes Referenced by this Node

It is not uncommon for a particular node of a Rational Rose model to reference other nodes in the model. To ensure consistency in your merged model, you may want to view these referenced nodes while making a decision about which contributor to select to resolve a given conflict. Also, if semantic checking is turned on and a choice of contributor has been disabled, viewing the referenced nodes can often reveal why this is so. Use **View > Referenced Node** for those nodes that have one or more of the three common types of references: client, supplier, and quidu.

Note: These reference types are used internally within the Rational Rose model and their meaning changes depending on the node viewed. The only significance they have in Model Integrator is that they link two different objects in the model together.

Nodes that have these references displays them in the property view. Figure 31 shows an excerpt from the property view of a TransView object that has all three types of references. To the right of the reference name is the name of the referenced node. You could scroll through the browser trying to find this node, or click **View > Referenced Node**.

Figure 31 Property View of a TransView Object

C	Field	Recipient
	[-]<builtin>	TransView
	@Name	
	[-]label	SegLabel
	stereotype	TRUE
	line_color	3342489
	quidu	-> Error State
	client	Verify Passenger Items
	supplier	Error State
	line_style	0
	x_offset	FALSE

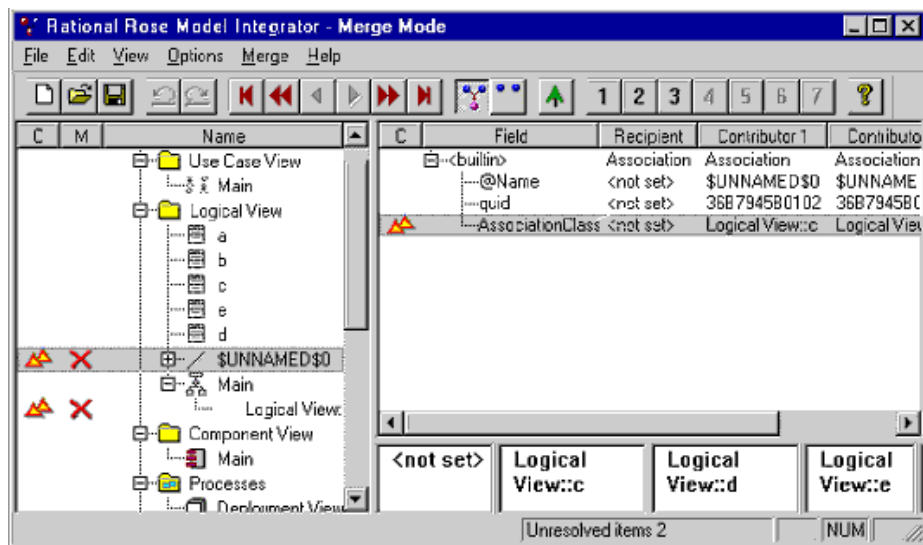
When a node in the model contains any of these references, **View > Referenced Node** is active for the type of reference (client, supplier, or quidu). The pop-up menu for each type of reference contains an entry for the recipient and each contributor, since the referenced nodes may be in different places in different models (one of the contributors may have moved them). If you or Model Integrator have already

accepted a change for the referenced node, **View > Recipient** becomes active. Typically, you view the recipient because it is saved in the merged model. If **Recipient** is not active, the referenced node is an unresolved item.

Accepting Changes from Contributors

The results of the merge are in the main Model Integrator window, as shown in Figure 32.

Figure 32 Model Integrator Window



The X indicates a node that must be resolved before the merge can be completed. To resolve the conflict, you must specify which of the contributors to accept.

Deciding Which Contributor to Select

The crucial issue in performing a merge is to decide which changes you want to keep. There are a few rules you can follow to make this job easier:

- Merge often while you are familiar with the changes.
- Partition the work and the model so that developers can work on different parts of it at the same time. This will reduce the number of conflicts you have to resolve.
- Know the models you are merging. Know in advance which contributors you want to select for major components of the model, such as classes and diagrams. This will help guide the choices you must make.

- You may encounter internal parts of the model that you do not understand. For these items, choose the same contributor that you chose for related items that you do understand.

For example, you have a use case with an associated interaction diagram, and you are selecting contributor 3 for this diagram because it has the most recent set of changes. If conflicts arise among the hidden objects, such as the mechanism or one of its components that are also part of this use case, you should select contributor 3 for those objects as well. This maintains consistency in the final merged model.

Two Ways to Accept Changes

There are two functions that let you accept changes from a contributor. You can:

- Resolve all the remaining conflicts with **Merge > Resolve All Conflicts Using**. This command lets you choose a single contributor to resolve all the remaining unresolved items. It operates over the entire merged model, regardless of where you are when you select it. However, it only operates on unresolved conflicts. Nodes that you have previously accepted changes for, or that are displaying only differences, will not be affected.
- Resolve an individual conflict or difference by selecting its node and then using **Merge > Resolve Selected Nodes Using**. This command copies one of the available contributor choices to the recipient. Unlike **Resolve All Conflicts Using**, it operates on any node that displays either a conflict or a difference, and previous choices are overwritten.

This command can also be used with the Subtree mode to resolve an entire subtree of model nodes at one time, or with a set of nodes selected using the mouse and SHIFT+CTRL keys. It affects all nodes displaying either conflicts or differences, and it changes values that have been set previously.

Warning: Subtree mode is very powerful. Use it with caution.

When semantic checking is enabled, Model Integrator disables choices of contributors that may produce errors in the recipient model. To make the change, you must turn off **Merge > Semantic Checking**.

You can use **Edit > Undo** to undo any merge choices you make.

Changing Nodes with Differences

You can accept changes from nodes that do not have conflicts, but do have differences. In this case, Model Integrator's AutoMerge procedure has already made the choice. The choice of contributor is not displayed in the M column of the browser window,

but does appear in the property view. The Recipient column displays the values for the chosen contributor. The AutoMerge choice is the contributor that is different from the others.

You can override the Auto Merge choice by selecting the node in the browser and clicking **Merge > Resolve Selected Nodes Using** to select a different contributor. The change is not accepted because you choose a contributor that does not change the model. This is useful when, for example, you do not want to delete a model element that is deleted in one of the contributors. When you apply this command to a node with a difference, the M column will show the contributor you've chosen for the result.

Reversing Changes to Nodes

If you change a node, you can always click **Edit > Undo** to restore it to its original state. If you previously made changes in your current merge session and you do not want to undo other work you completed, there is another method of reversing the changes. Click **Merge > Revert Selection** to restore a node to the unmerged state.

This command makes the node unresolved, regardless of whether or not it is a conflict. The M column for this node changes to display the “X” icon. For conflict nodes, this command removes your choice of a contributor to resolve the conflict. For difference nodes, this command removes the AutoMerge choice made by Model Integrator.

The **Merge > AutoMerge Selection** command is only applied to reverted nodes. Applying the AutoMerge command to reverted nodes restores them to the state they were in when the merge session started.

Using Subtree Mode

Subtree Mode allows you to apply merge mode commands to the current node and to all of its children. Click **Merge > Subtree Mode** to toggle to Subtree mode.

With Subtree mode not set, you can visit each subtree node and make independent choices of contributor for each node. With Subtree mode set, Model Integrator automatically applies the selected command to all of the children of the current node.

Merge mode commands affected by Subtree mode are:

- **Merge > Resolve Selected Nodes Using**
- **Merge > Revert Selection**
- **Merge > AutoMerge Selection**

Subtree mode is useful when you want to accept a group of related objects from a particular contributor. For example, you can accept an entire diagram from a contributor by selecting the top level node of the diagram, enabling Subtree mode, and then selecting **Merge > Resolve Selected Nodes Using** (or use the toolbar buttons).

Subtree mode is very powerful. Use it with caution, and turn off Subtree mode when you are done; but, you can always click **Edit > Undo** to undo any unwanted changes.

Using Semantic Checking

Semantic checks are performed by the **Check Merge** command before you save the merged model, but they are also available while you work. Clicking **Merge > Semantic Checking** enables Model Integrator to perform reference checking when you select a new node in the browser. Model Integrator disables merge choices that result in merge errors later in the session.

Enable this command when you want to avoid accepting changes that may produce errors. However, the checking performed by this function is not complete because it may take too long to check the entire model every time you select a different node. Consequently, Model Integrator must use the Check Merge function, and it may continue to find errors when semantic checking is enabled.

Disabling a Contributor Using Semantic Checking

When a contributor is disabled using semantic checking, you have two options:

- Track down the reason the choice is disabled by looking at the model elements referenced by this node (**View > Referenced Node**), or the parents of this node, or its referenced nodes (**View > Parent**). You will find that one of these nodes is already being deleted by another contributor. Choose a new contributor that does not delete the node, and then use **View > Previous Location** to return to the original node and make your desired choice.
- Turn off Semantic Checking, and make your desired choices. Rely on the Check Merge command to find errors when you finish your merge.

Checking Merged Model for Consistency

Click **Merge > Check Merge** to check your merged model for internal consistency. Inconsistency can occur during a merge operation when, for example, one of the contributor models being merged deletes model elements that are currently in use by another contributor to your merged model. This can occur because of:

- Decisions you make when you resolve conflicts between contributors.
- Decisions made by the automatic merging feature in Model Integrator.

You can use the **Check Merge** command to check your model while you are using Model Integrator to create a merged model. If there are errors, the **Check Merge** dialog box opens.

Before saving a merged model, Model Integrator automatically uses Check Merge to verify the model for consistency. If a merged model fails the Check Merge consistency check, Model Integrator does not allow you to save it.

Correcting Merge Errors

The **Merge Errors** dialog box floats above the main Model Integrator window. It provides a set of tools that can help you find and correct errors that Model Integrator detected in the merged model.

The first step in repairing an error is to select an error message in the list of errors in the **Merge Errors** dialog box. To correct a merge error, you must select a different contributor for some node of the model (See *Accepting Changes from Contributors* on page 99). The **Merge Errors** dialog box has buttons to help you find the node you need to change:

View Error: Takes you to the node of the model where the error was discovered - the error node.

View Definition: Takes you to the node of the model that defines the reference made by the error node.

View Parent: Takes you to the parent of the currently selected node in the browser. Use this command to search for the parent of a definition node. Click this button when you have a node whose parent is deleted.

View Other Locations: If the node you are viewing moved to different locations by a separate contributors, this command takes you to one of the other locations where the node exists. Only one of these locations actually exists in the merged output model; the other nodes are marked for deletion. Use this button when you have a forward reference error.

Refresh List: This command clears the error list and recomputes the Check Merge command. If new errors are encountered, they are listed. Use this command after fixing all the errors because there may be other errors that were hidden by the first set of errors. The same node of the model could have several errors, but only one is reported at a time. You may fix one error, but it may fix other errors. **Refresh List** always displays the current set of errors (if any).

Check Merge detects three types of merge errors. The error messages generated by Check Merge are:

- This node references a deleted node.

The error node references another node in the model that was deleted in the merged model. Click **View Definition** to display the location where the deletion occurs. This error must be corrected because the error node requires the other node to exist. To correct this error, choose either

- A contributor at the definition node that does not delete the node.
- A contributor that deletes the error node (if one is available).

Generally, choosing the same contributor at both locations is the preferred solution.

- This node references a node whose parent is deleted.

Here the problem is essentially the same as for the first message above, except that one of the parent nodes of the defining node is being deleted, rather than the defining node itself. When the parent node is deleted, all of its children are also deleted; you must change the parent node so that it is not deleted. Click **View Definition** to go to the defining node in the model. Click **View Parent** to move up the model tree until you find the parent node being deleted. Choose a contributor for this node that contains a definition of the node rather than deleting it.

- This node has a forward reference.

The error node references another node in the model moved backward in the merged model (for example, the definition previously occurred before the reference in the original model, but now it occurs after the node that references it). Certain forms of Rational Rose model references are only allowed to nodes defined first in the model. Click **View Definition** to display the original location of the node. Click **View Other Locations** to find where the referenced node was moved in the other contributors. To correct this problem, you must choose a contributor that will restore the definition node to its original place in the model. To move the nodes to the new location, you must do it in Rational Rose after the merge completes. This allows all references and definitions to be updated properly.

Saving Results

When the number of unresolved items is zero, you can save the model. Click **Save** to initiate the save operation. Model Integrator checks the model for errors. If it encounters errors, you must correct them before Model Integrator can save the model.

The **Merge Errors** dialog box has tools and help topics to help you correct these problems. After you finish correcting problems, close the **Merge Errors** dialog box and save the model again.

Model Integrator prompts you to specify the location to save the main model file. Select a file name and a directory for the output.

If your model contains loaded subunits, the **Subunits** dialog box prompts you to save the subunits. Click **OK** to continue the save operation. After the **Subunit** dialog box closes, the merge is complete and saved.

Performing a Partial Merge

You may have confined your editing in Rational Rose to a specific portion of the model, but when you load the model into Model Integrator, differences appear in other areas of the model that you may not expect. This is not an error on the part of Rational Rose or Model Integrator; it simply reflects the fact that the model is complicated, and not necessarily organized in the way you might expect.

However, if you want to restrict your merge session to a portion of the model, you can use a base model to provide the output for the parts of the model you do not want to modify. Follow this procedure to do a partial merge of a model:

- 1 Enter Merge mode, either from the **Contributors** dialog box or by clicking **Options > Merge Mode**.
- 2 Set Subtree mode by clicking **Merge > Subtree Mode**.
- 3 Select the root node of the model tree. This is the first node in the browser window.
- 4 Click **Merge > Resolve Selected Nodes Using > Contributor 1**. This causes the base model to be selected for all conflicts and differences in the entire model. The M column for the entire model changes to the **1** icon (nodes that were added by other contributors change to the **1-** icon).
- 5 Select the part of the model to actively merge. You can use Subtree mode if the area you want to merge consists of one or more subtrees. Otherwise, you can select portions of the model by holding down the **SHIFT** or **CTRL** keys while clicking nodes you want to select with the mouse. Ensure that you expand the model tree so that all nodes are visible or click **View > Expand All**.
- 6 Click **Merge > Revert Selection** to this part of the model to display an **X** icon opposite each node.
- 7 Click **Merge > AutoMerge Selection** to the same part of the model.

You have successfully restricted the AutoMerge command to a specific portion of the model. There may be conflicts in the part of the model you have reverted and automerged. Complete your merge of this part of the model and save the model.

Note: The **Check Merge** command may find errors due to references to the parts of the model you have excluded from the merge with this procedure. If this occurs, you must resolve the reference errors even when that means making changes outside of the area you have chosen to merge. You cannot save a merged model that has reference errors.

Merging Models Without a Base Model

To merge two files that do not have a common base model as an ancestor:

- 1 Click **File > Contributors** dialog box.
- 2 Before clicking **Compare** or **Merge** to load the models, clear **Compare/Merge Against Base Model** in the **Contributors** dialog box.
- 3 Load the models.

Model Integrator automatically creates an empty base model. The base model occupies the slot for Contributor 1, but it is not displayed and you cannot accept changes from it in the merge.

Note: Previous versions of Model Integrator required you to supply your own empty base model. Using this new feature, a separate empty base model is no longer required. Because a base model is not required in this mode, Model Integrator allows you to specify a merge session with as few as two files when **Compare/Merge Against Base Model** is cleared.

When merging models using this feature, all nodes in the contributors that do not conflict with each other appear with a plus “+” sign, indicating that they are being added to the merged model.

Viewing a Single Model File

Model Integrator supports a view mode for viewing the contents of a single model file. Do the following to view a single file:

- 1 Click **File > Contributors**.
- 2 Click a single file in the **Contributors** dialog box, and click **View**.

Note: If two files display, or if you enter a second filename, the button changes from **View** to **Compare**. When the button displays **Compare** and you have only entered a single file name, clicking the **Compare** button changes to View mode.

Using Model Integrator from the Command Line

Model Integrator supports a simplified command line interface used from the DOS and UNIX command lines.

Command	Description
modelint file.mdl	Starts Model Integrator with file.mdl in the View mode.
modelint file1.mdl file2.mdl	Starts Model Integrator in Compare mode for both files.
modelint file1.mdl file2.mdl file3.mdl	Starts Model Integrator in Merge mode with the first file named on the command line selected as the base contributor.

Additionally, you can use the following command line options; use either the slash character (/) or the minus sign (-) to begin each option:

Command	Description
/xcompare	Starts Model Integrator in Compare mode for the files named on the command line. This is the default mode for two files, but must be specified when comparing more than two files.
/xmerge	Starts Model Integrator in Merge mode for the files named on the command line. This is the default mode for three or more files.

Command	Description
/compare	<p>Starts Model Integrator in Compare mode but does not display the results in graphical mode. This mode performs the compare operation and then exits to the operating system with an exit code indicating the result of the compare operation:</p> <p>0 for identical models 1 for models with differences</p>
/merge	<p>Starts Model Integrator in Merge mode but does not display the results in graphical mode. If the merge algorithm detects conflicts, the merge is aborted and the program returns an exit code of 1. If the merge can be completed without conflicts, the merged file is written to disk to the file named by the /out parameter. If no /out parameter is specified, the Save dialog will be displayed. The Subunits dialog will also be displayed unless a subunit policy choice is made.</p>
/out <i>filename</i>	<p>Specifies the name of the file to write the merged output file to. You must specify an absolute or relative <i>pathname</i> for the file. Either of the following are valid:</p> <p>/out c:\models\test.mdl /out .\test.mdl</p> <p>but this is not valid</p> <p>/out test.mdl</p>
/ask /all /none	<p>Subunit policy options:</p> <p>The /ask option is the default in the graphical mode of Model Integrator. By default when reading and writing models, Model Integrator will display a subunit dialog that allows you to specify whether they want to load/save subunits.</p> <p>The /all option loads or saves all subunits without prompting the user with subunit dialogs.</p> <p>The /none option suppresses the loading and saving of subunits.</p>

This chapter is organized as follows:

- *Understanding Version Control* on page 109
- *Rational Rose Integration with Version Control Systems* on page 114
- *Using Rational ClearCase* on page 116
- *Using Microsoft Visual SourceSafe* on page 119
- *Using Version Control Features From Rational Rose* on page 120

Understanding Version Control

Successful team development requires versioning tools that meet certain minimum requirements, including:

- The ability to access artifacts in a controlled manner even when team members work from different geographic locations.
- An access mechanism that provides versioning of Rational Rose models and related artifacts.
- The ability for developers to concurrently access and modify different versions of an artifact.
- The ability to evaluate and merge changes that are introduced during concurrent development.
- The Ability to define configurations of related artifacts then checkpointing and retrieving them at any time.

Version control systems help make team development possible. At a minimum, they are repositories that store successive versions of files. A version control repository may contain thousands of files, but each version control user typically has a local working area for storing only a copy of the files in the repository that they needs to access.

Types of Version Control Systems

There are two types of version control systems, file-based and view-based. Each type of system has different features and methods for supporting the version control process. Consequently, there are features of each type that are not supported by the other.

File-Based Version Control Systems

Version control systems in this category include Microsoft Visual SourceSafe, Rational ClearCase with snapshot views, Revision Control System (RCS), and Source Code Control System (SCCS).

File-based version control systems require each user to have a copy of the files in a local folder, and use the file system's read-only attribute to control writing to files.

View-Based Version Control Systems

In view based version control systems, all versions of a file are stored in a versioned file system.

Users do not work with the contents of the versioned file system directly. Instead, they use a work area called a *view* that provides access to a set of files in the versioned file system. Moreover, a view provides access to an appropriate set of versions of those files by specifying how to choose the version of each file seen in the view.

Rational ClearCase is a view-based version control system.

Version Control Development Concepts

The following concepts are helpful when designing a development process for working with Rational Rose.

Development Activity

A development activity is comprised of changes to several elements. Each activity should encompass a unit of work, such as fixing a bug or defect, or adding a new feature. When changes for an activity are submitted to the repository, the model evolves to a consistent new state.

Integration

Integration is the process of making changes available for use by other developers. Typically, a single person performs the integration activities, but developers can also play this role.

Lineup

A lineup is a collection of specific versions of files from the version control repository. Examples of lineups are:

- Version 4 of every file involved in a project.
- The latest version of each file in the project dated before midnight, May 12.
- The version labeled “Build 6.1.112” of each file in the project.

Lineups represent significant combinations of files. In most development environments, the files that go into any nightly or production build form a lineup. Lineups are also valuable for reproducing specific builds of the system. The term baseline is also used to refer to a formal lineup.

Working in Isolation

It is essential that a developer’s work be isolated from the work of other developers. This is important for a number of reasons:

- To ensure that each developer can work without being influenced by other developers’ editing, compiling, testing and debugging.
- To ensure that each developer can access the appropriate material to perform his or her role. This usually requires using some sort of lineup process.
- To ensure that each developer does not expose work to other team members until it is ready for integration.

To support these basic team development requirements, developers need to have a private work area for implementing and testing code in accordance with the project’s adopted standards, and in relative isolation from other developers.

In addition to providing access to source versions, a work area needs to provide private (isolated) storage for files generated during software development, including:

- Working (checked-out) versions of source files
- Executables
- Other work area private objects and source code, test subdirectories, and test data files.

A work area private storage is typically located within a developer’s home directory on a workstation.

Versioning Strategies

Single Stream Versioning

Single stream versioning refers to having a single series of version numbers for each file. In effect, the version history for a file is a linear sequence of revisions.

While developing a project using single stream versioning, each developer always works with the most recent version of files in the repository. To edit a file, a reserved check out is performed on the latest version of the file. After developers make changes, they are submitted. This immediately makes the new version visible to other users, and becomes the latest version for others to base their changes on. This also means that only one person can work on each file at any one time since they must have the most recent version checked out in order to perform work.

Single stream versioning is not ideally suited for fixing defects or bugs for an existing release while doing new development for a future release.

You can use file-based and view-based version control systems for small projects without the need for branching or multiple stream development.

Benefits:

- Simple to set up.
- Work area configurations do not need to be modified.
- Users can browse any lineup stored in the repository.

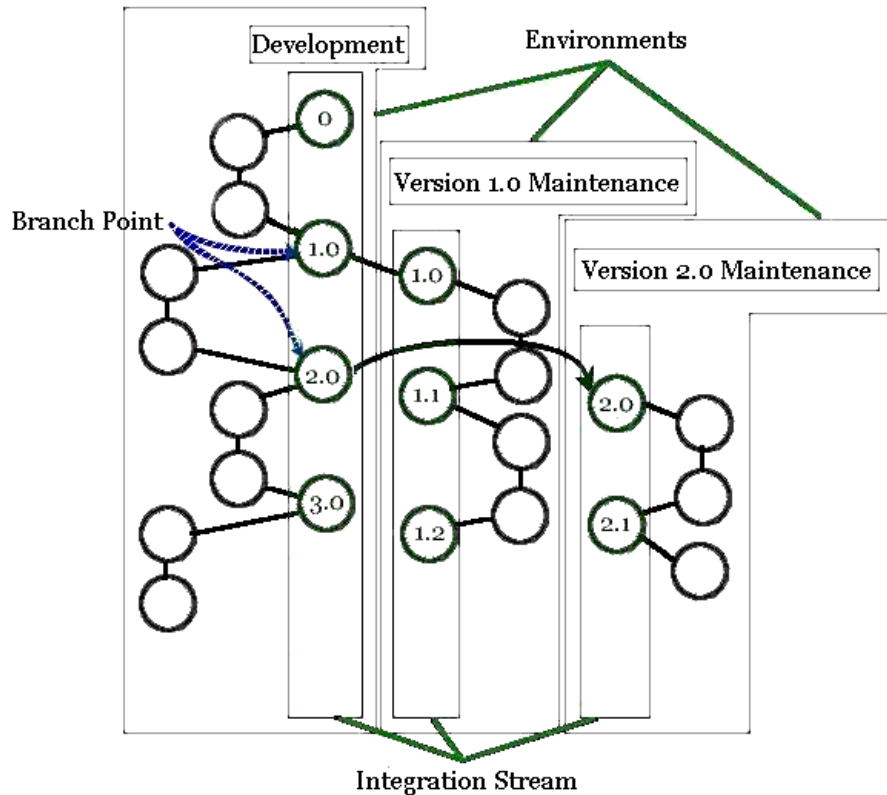
Drawbacks:

- Work is always based on latest version of elements in your version control system.
- You cannot work on arbitrary lineups - only with most recent version.
- If a developer checks in changes that are incompatible with the latest lineup in the version control, integration and build problems can occur.

Parallel Stream Versioning

Parallel stream versioning permits each file to have a branching tree of versions. This allows many versions of the same file to be active at the same time. Figure 33 shows the version tree for a typical file in a parallel development project.

Figure 33 Parallel Stream Versioning



Most parallel development environments involve nominating a branch in the version control system as the integration branch. The integration branch contains all changes to the project. Testing, release builds, and new development are based on the contents of the integration branch.

All labeled lineups should consist of file versions from the integration branch. Create a labeled lineup as the basis for builds, testing, or further development. You can also create and build a temporary lineup. If the build completes successfully and passes basic sanity tests, you can make the lineup available as a baseline. This process is usually automated, and should be done on a nightly or weekly basis, depending on your team development requirements.

The lineup of file versions in the baseline is used for subsequent development. Development activities should not be performed on the integration branch, but separate from it. When a development activity is finished, the changes for that activity

can be merged by an integrator back onto the integration branch. This ensures that the integration branch is strongly controlled and that only correctly working models are used to base further development on.

Benefits:

- Baselines are controlled.
- Baselines allow for the reuse of build results.
- Provides better control over exposing changes to the development team.

Drawbacks:

- Requires more sophisticated version control system knowledge.
- There is a separate integration step involved.
- Work area configurations must be regularly updated.

Rational Rose Integration with Version Control Systems

Version Control Add-In

Rational Rose provides version control facilities such as versioning and controlled access to model files by integrating with any SCC¹-compliant version control system.

Through its Version Control add-in, Rational Rose makes the most frequently used version control commands directly accessible from its **Tools** and shortcut menus.

For example, you can use the Version Control add-in to:

- Add packages to version control, which you must do before you can check out or check in the packages.
- Check out and check in packages.
- Start your SCC-compliant version control system.

Version Control logs its actions in the Rational Rose log window, as well as in the log file that you specify on the Log tab of the **Version Control Options** dialog box (**Tools**'> **Version Control** > **Version Control Options**).

1. SCC (Source Code Control) is the Microsoft standard API for version control systems.

The Version Control Add-In automatically determines which version control system you installed. To see which version control system and SCC API version the Version Control Add-In uses, see the **Version Control Options** dialog box.

Note: For the Version Control Add-In to work with your version control system, the version control system has to be configured for your environment.

ClearCase Add-In

The ClearCase Add-In provides a tight integration between Rational Rose and Rational ClearCase. In addition to the generic commands that the Version Control Add-In provides, the ClearCase Add-In provides:

- Reserved and unreserved checkout.
- Additional ClearCase query and browse commands.
- Support for managing files generated by the C++ and Ada Add-Ins.
- ClearCase-specific log reporting, including the Cleartool commands issued and complete ClearCase output messages for each command.

Note: For the ClearCase Add-In to work with ClearCase, ClearCase has to be appropriately configured for your environment.

Choosing and Activating a Version Control Add-In

When you install Rational Rose, the installation program detects whether ClearCase or an SCC-compliant version control system is installed on your system. Based on the version control system you installed, either the Version Control Add-in or the ClearCase Add-In is activated on your system.

If you install or change your version control system after you install Rational Rose, you must ensure that the appropriate add-in is activated. In addition, ensure that only one of the version control add-ins is active at a time. Because the Version Control and ClearCase add-ins use many of the same commands, you may receive error messages or unpredictable results if both are activated.

To activate or deactivate a version control add-in, click **Add-In Manager** on the **Add-Ins** menu, and click the add-in you want to activate or deactivate.

Note: If you use Rational ClearCase for version control, we recommend that you activate the ClearCase Add-In, even though you can also use the Version Control Add-In in the Windows NT or Windows 2000 environment. The ClearCase Add-In provides a much tighter integration and provides you with direct access to many ClearCase commands from within Rational Rose.

Using Rational ClearCase

About ClearCase

ClearCase is a comprehensive software configuration management system. It manages multiple variants of evolving systems, tracks which versions were used in software builds, performs builds of individual programs or entire releases according to user-defined version specifications, and enforces site-specific development policies.

These capabilities enable ClearCase to address the critical requirements of organizations that produce and release software, namely:

- **Effective development.** ClearCase enables users to work efficiently, allowing them to fine-tune the balance between sharing each other's work and isolating themselves from destabilizing changes. ClearCase automatically manages the sharing of both source files and the files produced by software builds.
- **Effective management.** ClearCase tracks the software build process so that users can determine what was built and how it was built. Further, ClearCase can instantly recreate the source base from which a software system was built, allowing it to be rebuilt, debugged, and updated -- all without interfering with other programming work.
- **Enforcement of development policies.** ClearCase enables project administrators to define development policies and procedures, and to automate their enforcement.

At its core, ClearCase has a secure data repository. It contains data that is shared by all users and includes current and historical versions of source files, along with derived objects built from the sources by compilers, linkers, etc.

In addition, the repository stores detailed accounting data on the development process itself, such as who created a particular version, what versions of source went into a particular build, and other relevant information.

Conceptually, the data repository is a globally accessible, central resource. The implementation, however, is modular. Each source (sub)tree can be a separate **versioned object base (VOB)**. VOBs can be distributed throughout a local area network, accessed independently, or linked into single logical tree.

Versioned Object Bases (VOBs)

ClearCase development data is organized into any number of versioned object bases (VOBs). Each VOB provides permanent storage for all the historical versions of all the source objects in a particular tree -- the right versions of the development objects appear, and all other versions are hidden.

A version-controlled object in a VOB is called an element. Its versions are organized into a version tree structure with branches and subbranches.

Figure 34 Version Controlled Object (VOB) Tree Structure

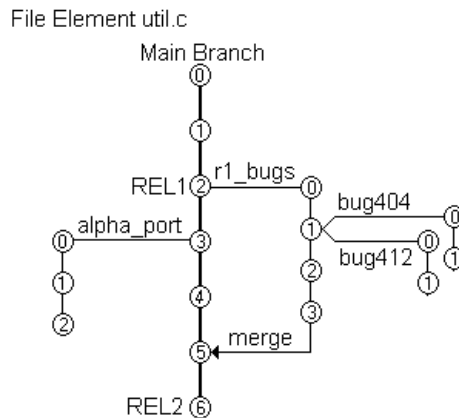


Figure 34 shows branches that have user-defined names to indicate their role in the development process. All versions have integer ID numbers. You can assign version labels to important versions to indicate development milestones, such as a product release.

ClearCase Views

Users access the ClearCase repository through views. A view is an isolated virtual work area that provides dynamic access to the entire data repository. The changes being made to a source file in a particular view are invisible to other views. Software builds performed in a view do not disturb the work taking place in other views.

Working in views, ClearCase users access version-controlled data using standard path names and their accustomed commands and programs. The view accesses the appropriate data automatically and transparently.

A view's isolation does not render it inaccessible. A view can be accessed from any host in the local area network. A view can be shared by several users working on a single host or on multiple hosts.

Configuring ClearCase for Rational Rose

By using a view model combined with a virtual file system, ClearCase allows users to specify the lineup of file versions with which they want to work. (A configuration specification, or config spec, controls the lineup used for a particular view.) Rational Rose can see the files in the current view as if they were stored on a regular (non-ClearCase) file system.

Rational Rose specifies the set of files that make up the model, and ClearCase provides the versions of these files based on the view's config spec. A config spec is a set of rules that determines which files are in a view.

To add files to version control, save the model to a view directory that is not view-private.

ClearCase allows you define a new element type, including specifying the merge and differencing tool that should be used on files of the new type. Rational Rose uses this feature to define an element type that applies to all Rational Rose files under version control. With this element type defined, all new Rational Rose files placed into a VOB are associated with the file type, and use Model Integrator as their default merge and differencing tool. (For more information about Model Integrator, see Chapter 5.)

Steps for Setup

To configure your ClearCase environment to work with Rational Rose:

- 1 Create and mount a ClearCase VOB (for example, ProjectRose).
Note: Create VOBs and views directly in ClearCase (outside of Rational Rose).
- 2 Create a ClearCase view to provide access to the VOB you created. If you use ClearCase on Windows, you must map the view to an appropriate Windows NT or Windows 2000 drive (for example, z:\)

- 3 Create all necessary views, prepare the model for team development, and organize the model in to controlled units.
- 4 To create the `rose_unit` element type in the VOB(s) where you store model files, do one of the following:

If you are using Rational Rose in the Windows NT or Windows 2000 environment:

- In a ClearCase command prompt window, change the directory to point to a drive and path representing a view and the VOB where your model files are located.
- Create the element type in this VOB, by typing:

```
cleartool mkeltype -supertype text_file -manager _rose -c "Model files" rose_unit
```

If you use Rational Rose in a Unix environment:

In Rational Rose, click **Tools > ClearCase > Setup VOB for Rose Units** to add the `rose_unit` element type and type manager to the VOB.

Using Microsoft Visual SourceSafe

Microsoft Visual SourceSafe (VSS) stores and retrieves files on your local disk. Each VSS project has a working folder specified for it. Rational Rose saves model elements to, and loads elements from this working folder. VSS then checks those local files in to and out of its repository.

Configuring Microsoft Visual SourceSafe for Rational Rose

Steps for Setup

- 1 Ensure that there is a Microsoft Visual SourceSafe database available to store the model. Create the database directly in Visual SourceSafe.
- 2 Since the Version Control Add-In uses the current user name to identify the SourceSafe user, the system administrator must add you as a user before you can use the integration. To identify the current user name, click **Version Control** on the **Tools** menu in Rational Rose, and click **Version Control Options**.
- 3 In Microsoft Visual SourceSafe, open the database where you want to store the model and create a project for the model. (You can start Visual SourceSafe from within Rational Rose by clicking **Version Control** on the **Tools** menu in Rational Rose, and then click **Start Version Control Explorer**.)

- 4 Right-click the project and click **Set Working Folder**. Select an existing working folder where you want to version control your model or create a new folder.

Note: If you attempt to control two files of the same name in the same Visual SourceSafe project, but you specify different working folders for the files (for example, c:\NewPackage.cat and c:\temp\ NewPackage.cat), Visual SourceSafe controls the first file in the project (c:\ NewPackage.cat); it does not control the second file (c:\temp\ NewPackage.cat). No error message informs you that the second file was not controlled. Consequently, we strongly recommend that you save all the files from a single project in the same working folder. Otherwise, you may think you have controlled a file, when you actually have not done so.

- 5 Click **Options** from the **Tools** menu. On the **Command Line Options** tab, select the **Assume Project Based on Working Folder** check box.

Note: If the Version Control commands on the **Tools** menu in Rational Rose do not work, the SourceSafe Integration component in Visual SourceSafe may not be installed. To install that component, start the Microsoft Visual SourceSafe setup program. Click **Add/Remove** and select **Enable SourceSafe Integration** on the **Maintenance Mode** page.

Using Version Control Features From Rational Rose

Using the Version Control Add-In on a Previously Controlled Model

If a model is currently under version control, but not through the Version Control Add-In, you must add the controlled units to version control by using the **Add to Version Control** command. If you do not add the model using this method, the Version Control Add-In will be unaware of any of the previously controlled units. for that model.

Note: This information applies only to the Version Control Add-In. If you use the ClearCase Add-In, you do not have to do anything because it will be aware of any previously controlled units.

Follow these steps to prepare a previously controlled model for use with the Version Control Add-In:

- 1 Ensure that the model is not currently open in Rational Rose.
- 2 In your version control system, check out all the controlled units that belong to the model. (This cannot be done through the Version Control Add-In because it does not know that the model is under version control.)

- 3 In Rational Rose, open the model, load all units, and click **Add to Version Control** on the **Tools** menu's **Version Control** menu.
- 4 If you want to check-in the files after this operation, clear **Keep Checked Out**.
- 5 If you use Microsoft Visual SourceSafe, click **Browse**, search for the project, then click **OK**. If all files are located in the same project, click **Select All**, then click **OK**. Otherwise, click the check box next to each file located in the selected project, click **OK**, and then repeat steps 3-5 for each set of files located in a different project.

The model is already under version control, so the Version Control Add-In only updates the controlled units with some additional Version Control information. Now, you can use the Version Control commands to check out and check in the units.

Adding Controlled Units to Version Control

The following procedure describes how to save a package to a file, and how to control it in ClearCase or an SCC-compliant version control system such as Microsoft Visual SourceSafe. You can use the same procedure to control the model file, the deployment diagram, or the model properties.

- 1 Ensure that the unit that the package belongs - that is, the model file or the enclosing package - is checked out.
- 2 Right-click the package in the browser or the diagram, then click **Add to Version Control**.
- 3 A list shows all model elements that can be added to version control. The specified packages are selected by default in the list. Ensure that you select all packages to add to version control from this list.
- 4 If you use Microsoft Visual SourceSafe, ensure that the **SourceSafe Project** box refers to the project that represents the working folder where the selected file units are (or will be) located. If the box does not refer to the appropriate project, click **Browse** and select the appropriate project.

Note: We recommend that you create new projects in the dialog box displayed when clicking **Browse**. However, if you do, the working folder for the new project becomes the same as the folder where you save the controlled units you are adding to version control.

- 5 To keep the files checked in after this operation, clear the **Keep Checked Out** check box.
- 6 You can also write a comment in the **Comment** box. Your version control system inserts the comment as a description of the new unit. This step is optional.

- 7 You can also click **Advanced** to display a dialog box with additional options. This step is optional.

This button is not available for all version control systems.

- 8 Click **OK**.
- 9 For each selected unit not yet saved to a file, a **Save As** dialog box displays. Specify the name and storage location of the new unit (for example, `x:\ordersystem\units\user_serv.cat`). You must save the file in the appropriate working folder in Visual SourceSafe or in the appropriate ClearCase view.
- 10 Click **Save**.

Rational Rose creates a file unit from each selected package, if needed, and adds each file to the version control system.

Note: If you use Microsoft Visual SourceSafe, the **Add to Version Control** command can only handle files located in the same Visual SourceSafe project and working folder. To add files that belong to different projects, you must repeat the **Add to Version Control** command for each project.

Checking in Controlled Units

To check in a loaded controlled unit into ClearCase or an SCC-compliant version control system, such as Microsoft Visual SourceSafe, use the following steps:

- 1 Right-click the unit in the browser or the diagram, and click **Check In**.
A list with all loaded, checked out controlled units in the current model displays. Any units currently selected in the browser or in a diagram are selected by default in the list.
- 2 Select the appropriate units.
- 3 Optionally, write an explanation of the check-in in the **Comment** box. The text you type is stored by your version control system as history for the current check-in.
- 4 Optionally, click **Advanced** to display a dialog box with additional options.

For example, to check in an unchanged unit in ClearCase, select the **Check in even if identical** option in the **Advanced** dialog box. If you do not select this option when checking in an unchanged unit, you receive an error message. (The **Advanced** button is not available for all version control systems.)

5 Click **OK**.

Rational Rose checks in the units and makes the corresponding model elements read-only in the model.

Note: If you use the Version Control add-in, to be able to check in a controlled unit from within Rational Rose, the unit must have been previously added to version control from within Rational Rose using the **Add to Version Control** command. This restriction does not apply to the ClearCase add-in.

If you use the Version Control Add-In with Microsoft Visual SourceSafe, the **Check In** command can only handle files located in the same Visual SourceSafe project and working folder. To check in files that belong to different projects, you must repeat the **Check n** command for each project.

Checking Out Controlled Units

To check out a loaded controlled unit from ClearCase or an SCC-compliant version control system, use the following steps:

- 1 Right-click the unit in the browser or the diagram and click **Check Out**. A list with all loaded, checked in controlled units in the current model displays. Any units currently selected in the browser or the diagram are selected by default in the list.
- 2 Select the units you want to check out.
- 3 Optionally, write an explanation of the checkout in the **Comment** box. The text you type is stored by your version control system as history for the current check-out.
- 4 Optionally, click **Advanced** to display a dialog box with additional options.

For example, if you use ClearCase, you can make an unreserved check out with the advanced options. (The **Advanced** button is not available for all version control systems.)

5 Click **OK**.

Rational Rose checks out the files and makes the contained model elements editable.

- 6 If Rational Rose prompts you to load the unit, click **Yes**.

Note: If you use Microsoft Visual SourceSafe, the **Check Out** command can only handle files that are located in the same Visual SourceSafe project and working folder. To check out files that belong to different projects, you must repeat the **Check Out** command for each project.

To check out a controlled unit, the unit must have been previously added to version control from within Rational Rose using the **Add to Version Control** command.

Undoing the Check-Out of Controlled Units

To undo the check-out of a loaded controlled unit and to load the latest checked-in version:

- 1 Click **Version Control** on the **Tools** menu, and then click **Undo Check Out**.
A list with all loaded, checked-out controlled units in the current model displays. Any packages currently selected in the active diagram are selected by default in the list.
- 2 Select the checked-out unit.
- 3 Optionally, click **Advanced** to display a dialog box with additional options. This button is not available for all version control systems.
- 4 Click **OK**.
- 5 If Rational Rose prompts you to save the changes before loading a new unit, click **No**.

Note: If you use Microsoft Visual SourceSafe, the **Undo Check Out** command can only handle files located in the same Visual SourceSafe project and working folder. To undo the check-out of files that belong to different projects, you must repeat the **Undo Check Out** command for each project.

Getting the Latest Version of Controlled Units

To copy the latest checked-in version of a loaded controlled unit to your Microsoft Visual SourceSafe working folder, or dynamically access it via the ClearCase view to load that version into the model, follow these steps:

- 1 Click **Version Control** on the **Tools** menu, and then click **Get Latest**.
A list with all loaded, checked-in controlled units in the current model displays. Any packages currently selected in the active diagram are selected by default in the list.
- 2 Select the appropriate unit.
- 3 Optionally, click **Advanced** to display a dialog box with additional options. This button is not available for all version control systems.
- 4 Click **Get**.
- 5 If Rational Rose prompts you to save the changes before loading a new unit, click **No**.

Note: If you use Microsoft Visual SourceSafe, the **Get Latest** command can only handle files that are located in the same Visual SourceSafe project and working folder. To get the latest versions of files that belong to different projects, you must repeat the **Get Latest** command for each project.

If you use Rational ClearCase, the **Get Latest** command is only valid for snapshot views. See your ClearCase documentation for more information on views.

Removing Controlled Units from Version Control

To remove a loaded controlled unit from version control, follow these steps:

- 1 Ensure that the unit to which the unit belongs - that is, the model file or the enclosing package - is checked in.
- 2 Click **Version Control** on the **Tools** menu, and select **Remove From Version Control**.
A list shows all loaded controlled units under version control by the Version Control Add-In. Any packages currently selected in the active diagram are selected by default in the list.
- 3 Select the unit that you want to remove.
- 4 Optionally, click **Advanced** to display a dialog box with additional options. This button is not available for all version control systems.
- 5 Click **OK**. The selected unit is removed from version control and its contents are incorporated into the model, but will continue to exist as:
 - For Microsoft Visual SourceSafe: A file in your working folder.
 - For ClearCase: A file in your view, if you are using a snapshot view, but the file is automatically removed from all dynamic views.

Note: If you use Microsoft Visual SourceSafe, the **Remove From Version Control** command can only handle files that are located in the same Visual SourceSafe project and working folder. To remove files that belong to different projects, you must repeat the command for each project.

Note: To remove a controlled unit from version control from within Rational Rose, the unit must have been added to version control from within Rational Rose, by using the **Add to Version Control** command.

Index

Symbols

\$CURDIR 71
\$UNNAMED\$ 83
@petal property 83

A

access violations 13, 72, 73
 import 73
activating a Version Control Add-In 115
adding
 controlled units to a model 60
adornments on diagrams 61
Architect 7
architect role 43
architecture, model 5
artifacts, and virtual path maps 71
Auto Advance 97
automatic merging 80
automating model validation 30
AutoMerge 92

B

base model 80
baseline 113
basic objects 82
browser view (Model Integrator) 78

C

cat files 56
Check Merge 84, 102
Check Model 72
checking in controlled units 122
checking out controlled units 123
ClearCase

 about 116
 add-in 115
 and Model Integrator 86, 118
 config spec 118
 configuring 118
 views 117
command line
 access to Model Integrator 107
 starting Model Integrator 87
compare mode 77
comparing 80
comparing models
 about 77, 80
 conflicts 81
 differences 81
 loading controlled units 89
 using Model Integrator 80
component instances 18
config spec 118
configuration manager role 10
configuring
 ClearCase for Rational Rose 118
 Microsoft Visual SourceSafe for Rational
 Rose 119
 workspaces 26
conflicts 81
 Auto Advance 97
 viewing 96
contacting Rational technical support xvii
context field (Model Integrator) 92
contributors 77, 92
 about 80
 accepting changes from 99
 disabled by semantic checking 102
 file types 80
 selecting 88
controlled units 55
 access violations 72
 adding to a model 60, 66

- adornments 61
- cat files 56
- checking into version control 122
- checking out of version control 123
- contents 57
- creating 59
- getting latest from version control 124
- hierarchy 57
- icons 60
- importing 66
- loading 59, 89
- loading manually 60
- logical packages 56
- manually loading 60
- merging 66
- model workspaces 60
- moving 48
- opening a model 59
- organizing 74
- partitioning
 - strategies 74
- partitioning a model 27
- prc files 56
- protecting 64
- read only 64
- reloading 60
- removing from version control 125
- sharing 57
- splitting 65
- sub files 56
- uncontrolling 66
- undoing check out 124
- unloading 61, 89
- unresolved references 72
- use-cases 49
- version control of 55
- virtual path maps 67
- write-enabling 65
- write-protect 65
- write-protecting 64

correcting merge errors 103

creating

- controlled units 59
- labels and lineups 28

- model workspace 63
- processors and component instances 17
- virtual path maps 69

D

- defining subsystem interfaces 15
- deployment view 56
- developing
 - current projects 1
 - for reuse 2
 - strategy 1
- diagram objects 82
- differences 81
 - Auto Advance 97
 - changing nodes 100
 - viewing 96

E

- element type 118
- export control 73

F

- file types
 - cat 56
 - for contributors 80
 - mdl 107
 - petal 58, 83
 - prc 56
 - prp 56
 - ptl 58
 - sub 56
 - wsp 64
- file-based version control 110

G

- getting latest version of controlled units 124

I

- icons, controlled units 60
- import relationship 73
- importing controlled units 66
- integrating change 30
- integration 110
- Integrator 8
- interfaces, and subsystems 15

L

- labels 28
- lineups 28, 29
 - about 111
- loading a model workspace 64
- loading controlled units 59, 89
 - manually 60
 - model workspaces 60
 - opening a model 59
 - reloading 60

M

- manually loading controlled units 60
- mechanism 82
- merge mode 77
- merging
 - controlled units 66
 - file types 80
- merging models
 - about 77, 80
 - accepting changes from contributors 99
 - automatic merge 80
 - AutoMerge 92
 - Check Merge 102
 - Check Merge function 84
 - conflicts 81
 - correcting merge errors 103
 - differences 81
 - loading controlled units 89
 - partial merge 105
 - preparing for 88
 - selective merge 80

- semantic checking 84
- using subtree mode 101
- whole models 87
- without a base model 106

Microsoft Visual SourceSafe 119

model

- unresolved references 72

Model Architect 7

model architecture

- about 5

Model Integrator 51

- about 77
- accepting changes from contributors 99
- and ClearCase 86, 118
- Auto Advance 97
- automatic merge 80
- AutoMerge 92
- base model 80
- basic objects 82
- browser view 78
- changing nodes with differences 100
- Check Merge 102
- Check Merge function 84
- comparing models 80
- conflicts 81
- contributors 80
- correcting merge errors 103
- diagram objects 82
- differences 81
- mechanism 82
- merging models 80
- merging models without a base model 106
- partial merges 105
- property view 79
- quids 83
- references 83
- resolving subunit loading errors 91
- searching for nodes 95
- selecting contributors 88
- selective merge 80
- semantic checking 83, 84, 102
- setting new context for subunits 92
- starting 87
- subtree mode 101
- text views 79

- unnamed objects 83
- user interface 78
- using from a command line 107
- view objects 82
- viewing a parent node 97
- viewing a single model 107
- viewing conflicts 96
- viewing differences 96
- viewing references to nodes 98
- virtual path maps 91
- Model Manager 7
- model properties
 - controlled units 56
 - using virtual path maps 71
- model validation 30
- model workspaces
 - about 61
 - creating 63
 - loading 64
 - loading controlled units 60
 - saving 64
- Modeler/Developer 8
- modelint file.mdl 87, 107
- models 80
 - base 80
 - contributors 80
 - merging 80
 - selective merging 80
- moving controlled units 48
- moving the contents of a controlled unit 65

N

- nodes
 - about 81
 - changing, with differences 100
 - searching for 95
 - viewing a parent 97
 - viewing references to 98

O

- objects

- basic 82
- diagram 82
- view 82
- opening a model 59
- organizing controlled units 74

P

- packages
 - access violations 13
 - as subsystems 10
 - component 56
 - logical 56
 - partitioning a model 27
 - sharing 57
- parallel development 50
- parallel stream versioning 112
- partial merges 105
- partitioning
 - model 27
- partitioning a model 5, 58, 74
- path maps
 - artifacts 71
 - defining 70
 - wildcards 70
- path maps, See virtual path maps
- petal file format 58
- planning
 - developing a strategy 1
 - team development 1
- prc files 56
- project level processors 17
- property view (Model Integrator) 79
- protecting controlled units 64
- prp files 56
- ptl files 58

Q

- quids 83
- quidu 83

R

- Rational ClearCase Multi-Site 52
- Rational technical support
 - contacting xvii
- read only controlled units 64
- reference a controlled unit 57
- references
 - checking 72
 - in Model Integrator 83
 - unresolved 61, 66
- releasing subsystems 19
- reloading controlled units 60
- removing controlled units from version control 125
- resolving subunit loading errors 91
- resources xvi
- reusing artifacts 2
- roles 6
 - architect 43
 - configuration manager 10
 - source control administrator 9
 - team size 6

S

- saving a model workspace 61, 64
- SCC version control systems 114
- searching for nodes 95
- selective merge 80
- semantic checking 83, 84, 85
 - limitations 85
 - performing 102
- setting
 - new context for subunits 92
- Show Access Violations 13, 73
- single stream versioning 112
- source control administrator role 9
- splitting
 - controlled unit 65
 - model into subsystems 20
- starting Model Integrator 87
- strategies for partitioning a model 74
- sub files 56

- subsystem level processors 18
- subsystems
 - components in 15
 - defining interfaces 15
 - releasing 19
 - splitting a model 20
- Subtree Mode 101
- subunits 59

T

- tasks
 - source control administrator role 9
- team development
 - architect role 43
 - configuration manager role 10
 - developing for reuse 2
 - developing strategies 1
 - heuristics 52
 - parallel development 50
 - planning 1
 - roles 6
 - source control administrator role 9
 - support using Rational Rose 2
 - team size 6
 - typical roles 6
- text views (Model Integrator) 79

U

- uncontrolling controlled units 66
- undoing check out 124
- unit testing 17
- unloading controlled units 61, 89
- unnamed objects 83
- unresolved references 61, 66
 - checking for 72
- URLs 71
- use-cases
 - controlled units 49
- using Model Integrator form a command line 107

V

- validating a model 30
- version control
 - about 109
 - activating 115
 - adding controlled units 121
 - baseline 113
 - checking in controlled units 122
 - checking out controlled units 123
 - development activity 110
 - development concepts 110
 - file-based 110
 - getting latest controlled units 124
 - integration 110
 - lineups 111
 - removing controlled units 125
 - setting up 27
 - single stream 112
 - types 110
 - uncontrolling controlled units 67
 - undoing check out 124
 - view-based 110
 - working in isolation 111
 - write-protecting controlled units 64
- Version Control Add-In 114
- versioned object base 116
- versioned object base, see VOB
- view
 - version control system 110
- view objects 82
- view-based version control 110
- viewing

- conflicts 96
- differences 96
- moved model elements 97
- parent node 97
- references to nodes 98
- single model file (Model Integrator) 107
- views
 - ClearCase 117
- views (Model Integrator) 78
- virtual path maps
 - about 67
 - creating 69
 - for artifacts 71
 - for model properties 71
 - how stored 72
 - in Model Integrator 91
 - using another path map 70
 - using wildcards 70
 - wildcards 70
- VOB 116
- VSS 119

W

- working in isolation 111
- workspaces
 - configuring 26
 - model 61
- write enabling a controlled unit 65
- write protecting controlled units 64
- write-protect a controlled unit 65
- wsp files 64