Rational Rose Guide to Team Development

Contents

How This Manual Is Organized ix

Related Documentation x

Chapter 1 Understanding Team Development 1

Planning for Team Development 1
- Developing a Strategy 2
  - Current Projects 2
  - Developing for reuse 2

How Rational Rose Supports Team Development 3

Using this Guide 4

Chapter 2 Breaking a Model into Controlled Units 7

What is a Controlled Unit? 7
- What Can be a Controlled Unit 8
- How Controlled Units are Related and What They Contain 9

Working with Controlled Units 12
- Creating Controlled Units 12
- Loading, Reloading and Unloading Controlled Units 12
- Creating and Using Model Workspaces 14
  - Understanding Workspaces 14
  - How a Saved Model Differs from a Model Workspace 15

Workspace Example 15
Creating and Saving a Model Workspace 17
Loading a Model Workspace 17
Protecting Controlled Units 17
To Write-Protect a Controlled Unit 18
To Write-Enable a Controlled Unit 18
Splitting a Controlled Unit 18
Merging Controlled Units 19
Adding Controlled Units to a Model (Importing/Loading) 19
Uncontrolling Controlled Units 20

Creating Virtual Paths to Controlled Units 21
Understanding Virtual Path Maps 21
How Virtual Paths Work 22
How to Create Virtual Path Maps 23
Defining a Path Map Relative to the Location of the Model File 24
Defining a New Path Map Using Another Path Map Symbol 25
Defining a Path Map with Wildcards 25
Using Virtual Paths for the Value of a Model Property 26
Using Path Maps for other Artifacts 26
Where Virtual Path Maps are Stored 27

Checking References and Access Violations 27
Check Model 27
Show Access Violations 28

Organizing Controlled Units for Teams 29
Suggested Strategies 29

Chapter 3 Comparing and Merging Models 31

About the Model Integrator 31
About the Model Integrator Interface 32
Browser view 32
Property view 33
Text views 33
Other Interface Features 33
About Contributors 34

About the Base Model 34
Comparing Models 34
Merging Models 35
About Differences and Conflicts 35
About Model Files and Model Integrator 36
Understanding Semantic Checking 38
Limitations of Semantic Checking 40
About Memory Requirements and Performance 40

About Model Integrator and ClearCase 42
Merging Whole Models with Controlled Subunits 42
Starting Model Integrator in a ClearCase Integration 43

Comparing and Merging Models 44
Starting Model Integrator 44
Preparing Models for Merging 44
Selecting the Contributors 44
Loading or Unloading Controlled Units 45
Subunit Status 46
Loading Subunits 47
Saving Subunits 47
Subunit File and Path Names 48
How to Resolve Subunit Loading Errors 48
Setting a New Context for Subunits 49
Using Compare Mode 50
Using Merge Mode 50
AutoMerge 50
Interpreting Compare and Merge Results 51
Navigating through a model 53
Searching for a Model Element 53
Viewing Conflicts and Differences 54
Viewing Conflicts and Differences with Auto Advance 55
Viewing Model Elements that have Moved 56
Viewing the Parent of a Node 56
View Nodes Referenced by this Node 57
Accepting Changes from Contributors 58
Deciding Which Contributor to Select 59
Two Ways to Accept Changes 59
Changing Nodes with Differences 60

Reversing Changes to Nodes 61
Using Subtree Mode 61
Using Semantic Checking On-the-Fly 62
When a Contributor is Disabled by Semantic Checking 62
Checking Merged Model for Consistency 62
Correcting Merge Errors 63
Saving the Results 65

Performing a Partial Merge 65

Merging Models Without a Base Model 66

Viewing a Single Model File 67

Using Model Integrator from the Command Line 67

Chapter 4 Working with a Version Control System 71

Understanding Version Control 71
Types of Version Control Systems 72
File Based Version Control Systems 72
View Based Version Control Systems 72
Version Control Development Concepts 72
Development Activity 72
Integration 73
Lineup 73
Working in Isolation 73
Versioning Strategies 74
Single Stream Versioning 74
Parallel Stream Versioning 75

Using Rose's Integration with Version Control Systems 78
Version Control Add-In 78
ClearCase Add-In 79
Choosing and Activating a Version Control Add-In 79

Using Rational ClearCase 80
About ClearCase 80
Versioned Object Bases (VOBs) 81
ClearCase Views 82

Setting ClearCase Up to Work with Rose 82
Steps for Setup 83

Using Microsoft Visual SourceSafe 84
Setting Up Microsoft Visual SourceSafe to Work with Rose 84
Steps for Setup 84

Using Version Control Features From Rose 85
Using the Version Control Add-In on a Previously Controlled Model 85
Adding Controlled Units to Version Control 86
Checking in Controlled Units 87
Checking Out Controlled Units 88
Undoing the Check-Out of Controlled Units 89
Getting the Latest Version of Controlled Units 89
Removing Controlled Units from Version Control 90

Working with Non-SCC Version Control Systems 91
Repository Mapping Files (.rmf) 92
Version Control Operation Behavior with SCCS 92
RCS/SCCS Repository Setup 93
RCS/SCCS Workstation Setup 93
Command Line Access to the Source Control Tool 93
Create an RMF File 93
Set RMF Environment Variable 93

Chapter 5 Establishing a Model Architecture and Process for Team Development 95

About Model Architecture and Process 95

Establishing Roles and Responsibilities 95
Model Architect 96
Model Manager 96
Modeler/Developer 98
Integrator 98

Developing a Model Architecture 99
Understanding Subsystems 99

One Model versus Multiple Models 101
Mapping the Architecture to Subsystems 102
Checking Package Dependencies for Completeness 103
Show Access Violations 103
Determine the External Dependencies for a Package 103
Checking if a Subsystem is Self-Contained 104
Defining Subsystem Interfaces 105
Setting Up Subsystem Components 105
Components in Subsystems 105
Providing Support for Unit Testing 107
Using Property Sets for Build Settings 107
Creating Processors and Component Instances 108
Project Level Processors 108
Subsystem Level Processors 108
Component Instances 109
Preparing and Releasing Subsystems 109
Splitting a Model into Subsystem Models 110
Should you split the model before adding to version control? 111
Splitting a Model not in Version Control 111
Splitting a Model Under Version Control 114

Managing/Administering a Model 117
Setting Up Compatible Workspaces 117
Setting Up Version Control System and Repository 118
Partitioning the Model into Controlled Units 118
Save Model to Local Work Area 119
Adding the Model to Version Control 119
Defining Developer Work Areas 119
Creating Labels and Lineups 120
Manipulating the Version Control Repository 120

Developing/Implementing a Model 120
Setting up Version Control 120
Setting up Developer Work Areas 120
Getting a Specific Lineup of a Model 121
Opening a Model Under Version Control 121
Working under Version Control 121

Comparing and Merging Model Elements 121
Promoting Changes for Integration 122

Building and Integrating 122
Building using Automated Scripts 122
Building within a Larger Build Procedure 123
Reusing Build Artifacts 124
Creating Reusable Build Artifacts 124
Using Build Artifacts 125
Integrating Changes 126
Automating Model Validation 126

Chapter 6 Parallel Development Sample Using ClearCase 127

Overview 127

Using View Templates 129

ClearCase Entities 130

Initial Setup 131
Create the Integrator View 131
Create Project Labels 131
Create Initial Lineup 132
Creating the Developer View Template 132
Windows NT 133
Unix 135

Automated builds 135
Create the Build View 136
Unix 136
Label Build Files 136
Perform Build 137
When the Build Completes Successfully 137
Create a new lineup label and apply to build file versions 137
Apply TC_LATEST_STABLE to build file versions 137
Make New Lineup Available to Developers 138

Developer Process 138
Creating a Developer View 139

Windows NT 139
Unix 139
Starting a Development Activity 139
Windows NT 139
Unix 140
Working on a Development Activity 140
Finishing a Development Activity 140
Windows NT 140
Unix 140

Integration process 141

View Template Script Usage 141
vtadmin 141
vtsetview 142

Preface

This guide, Rational Rose Guide to Team Development, is intended for those users who work in or support teams of modelers/developers. It provides background information about Rose and the tools that support working in teams, as well as guidelines for managing parallel development.

How This Manual Is Organized

This manual contains the following six chapters:

- **Chapter 1**—Introduction

  Provides an overview of the material covered in the guide.
- **Chapter 2**—Breaking a Model into Controlled Units

  Introduces controlled units and describes how they can be used to partition a model into manageable parts.
- **Chapter 3**—Comparing and Merging Models

  Describes how to use Model Integrator to compare and merge the work of multiple developers.
- **Chapter 4**—Working with a Version Control System

  Describes the fundamental concepts behind version control systems and provides details about Rose's integration with ClearCase and Microsoft Visual SourceSafe.
- **Chapter 5**—Establishing a Model Architecture and Process for Team Development

Provides guidelines for defining team roles and responsibilities and for developing a process for managing parallel development.

- **Chapter 6**—Parallel Development Sample Using ClearCase

  Documents a sample environment using ClearCase as the version control system.

Related Documentation

The information in this guide spans numerous products, both from Rational and from other software vendors. To learn more about these products, you should become familiar with their documentation. The Rational Rose Guide to Using Rose and the Rational Rose online help provide detailed information about Rose models and the Model Integrator.

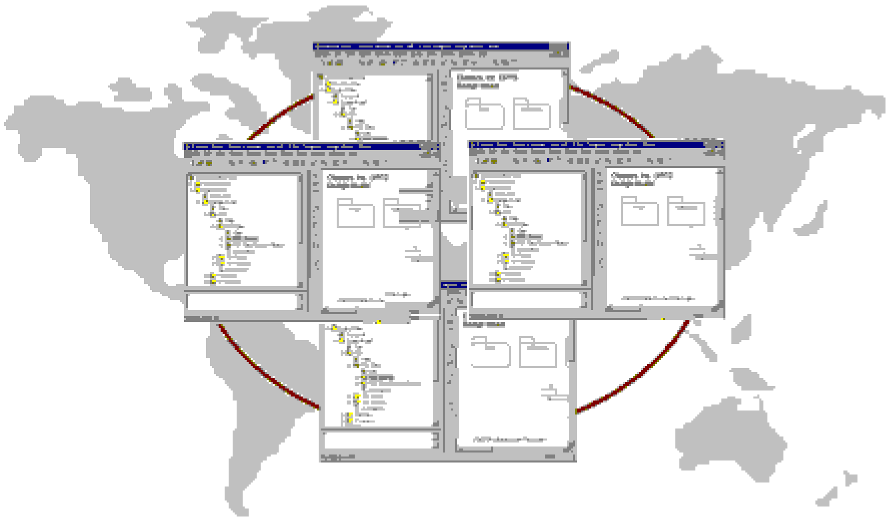
Chapter 1

Understanding Team Development

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 —simultaneously. 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 that can support that strategy



Developing a Strategy

When developing a strategy for working in teams, remember that there are two facets to consider:

- Developing a strategy that supports current development
- Developing a strategy for maintaining and retrieving the reusable modeling artifacts that result

Current Projects

When developing current projects, the tools 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/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 are developing 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

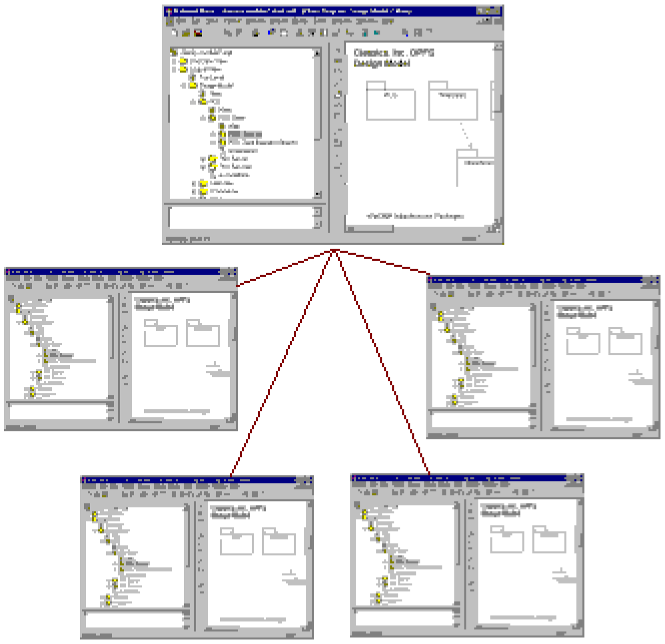
A reuse repository can differ from your project's configuration management system as long as it supports versioning. The repository should also support cataloging artifacts at an appropriate level of granularity (for example, at the component level).

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, Rose provides integrations with Rational ClearCase and with SCC compliant version control systems, such as Microsoft Visual Source Safe. By integrating configuration management systems, Rose makes the most frequently used version control commands directly accessible from Rose menus, including the typical check in and check out functions that are used every day.

A single model needs to be shared

by teams of designers and developers

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.

The information it contains spans product lines, including software from other vendors. Its primary goal is to help you develop and tailor your own guidelines that you can implement.

While it does provide detailed explanations of some features (such as Model Integrator), you will need to rely on additional product libraries for information. For example, you will need the ClearCase documentation in order to configure and set up ClearCase to work in your environment.

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

Chapter 2

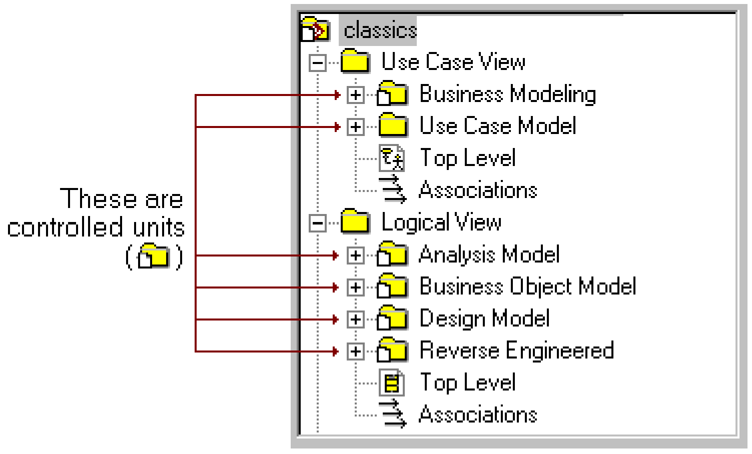
Breaking a Model into Controlled Units

What is a Controlled Unit?

By default, 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 breaking 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 each 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.



What Can be a Controlled Unit

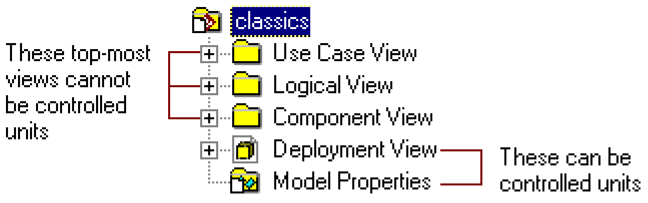
You can create controlled units for packages in your Use Case, Logical, Component, and Deployment Views, as well as a controlled unit for your model properties.

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

- 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 but since a Rose model supports one deployment diagram, there is only one .prc file. Similarly, there is a single set of model properties and only one .prp file.

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



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:

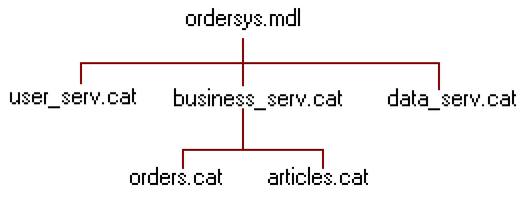
- All model elements that are in the package
- All packages that are in the package (or a reference to those packages if they are also controlled units)
- All diagrams that belong to the package

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

Note that 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 controlled units it encloses. For example:



In this case, articles.cat can be changed without affecting business_serv.cat and business_serv.cat can be changed without affecting ordersys.mdl.

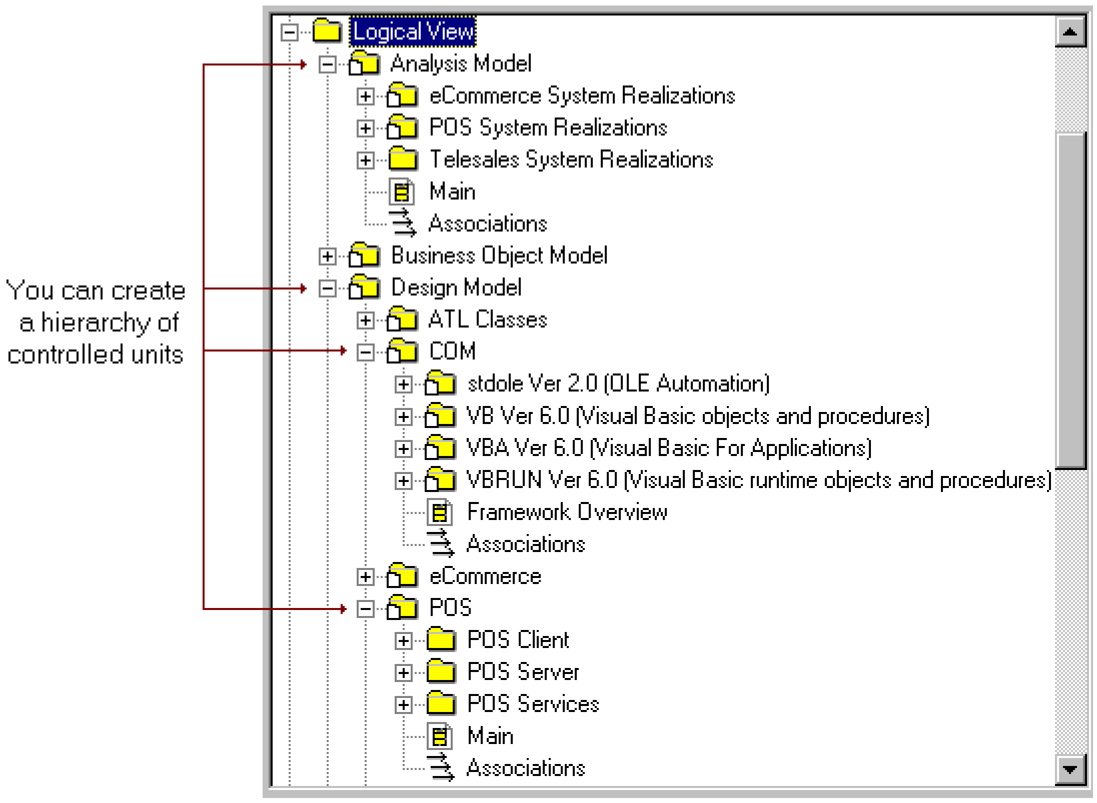
You can create a virtually unlimited hierarchy of controlled units where top level controlled units consist of references to other controlled units.

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

Note, too, that packages can be shared. By creating controlled units, multiple models can share the same packages, enabling 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).

The following illustrates how controlled unit hierarchies appear in the Rose browser.



- Logical View
  - Analysis Model
    - eCommerce System Realizations
    - POS System Realizations
    - Telesales System Realizations
    - Main
    - Associations
  - Business Object Model
  - Design Model
    - ATL Classes
    - COM
      - stdole Ver 2.0 (OLE Automation)
      - VB Ver 6.0 (Visual Basic objects and procedures)
      - VBA Ver 6.0 (Visual Basic For Applications)
      - VBRUN Ver 6.0 (Visual Basic runtime objects and procedures)
      - Framework Overview
      - Associations
    - eCommerce
    - POS
      - POS Client
      - POS Server
      - POS Services
      - Main
      - Associations

Note that the format Rose uses for the model (.mdl) and controlled unit files is called a petal format. It's a text based format, meaning that you can open and view the 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**. Rose prompts you to supply 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 that once you have created 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, make sure that the enclosing file is write-enabled. If it is under version control, you must check the file out.

Carefully consider the file structure you implement when creating/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.

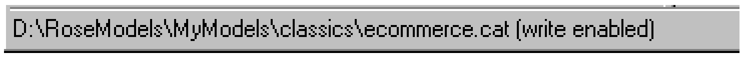
Also, in the early stages of analysis and design, the architecture of your model can change, sometimes drastically. During these early stages, modeling elements may be created, moved, or deleted often. When a controlled unit is moved in the model, the corresponding controlled unit file is not automatically 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:

- **By opening a model**, If a model has controlled units, Rose will prompt you whether or not to load all the subunits when you open the model. If you respond with Yes, Rose loads all of the controlled units associated with the model.

- **By manually loading units**. You can use **File > Units > Load** (or **Reload**) to individually load each controlled unit file you need. Note that when you point to a package in the browser, Rose displays the controlled unit file name on the status bar (whether or not it is currently loaded).


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

  If you double-click on an unloaded package in the browser Rose will load it.

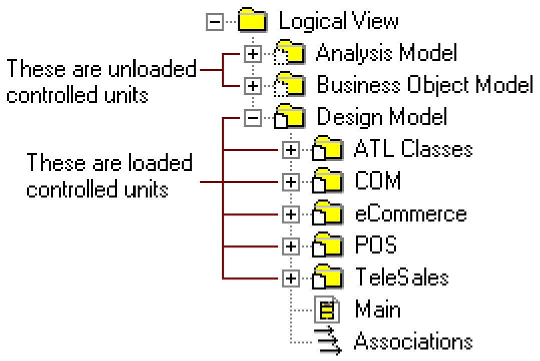
  Note that you can use **File > Units > Load** or **File > Import** if you want to add controlled units from another source to your model.

- **By loading a model workspace.** Rose enables you to save your working environment (the set of controlled units you have loaded). For complete details about model workspaces, see *Creating and Using Model Workspaces* later in this chapter.

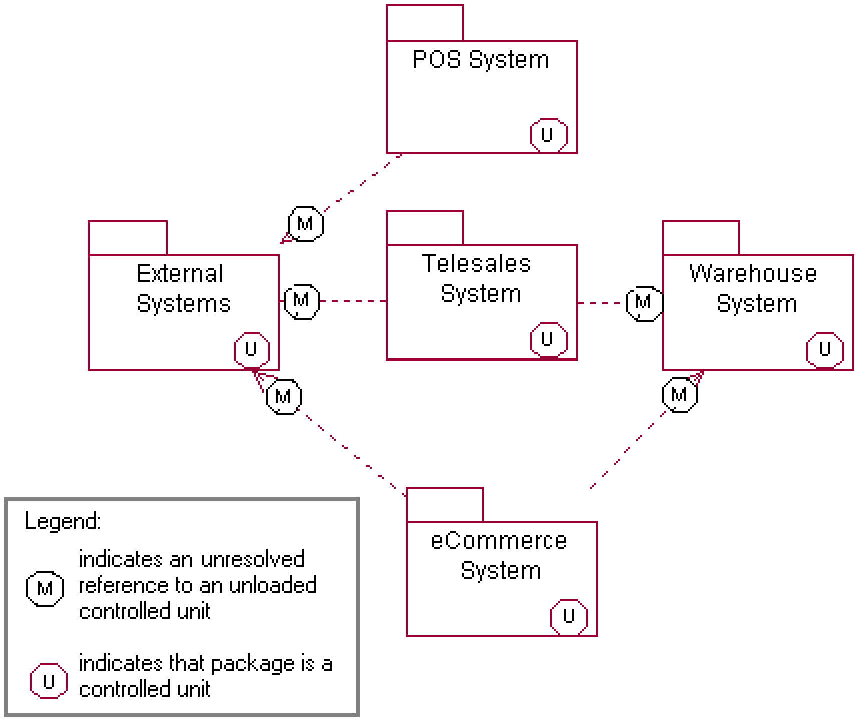
If your model is large, or you are planning to work on a few specific units, you can greatly reduce latency and resource consumption by manually loading individual controlled units or by creating and then loading a model workspace.

Note that to view or update a unit that has been modified by another developer since you loaded it, you must reload the unit by using **File > Units > Reload.**

Rose uses these icons in the browser to indicate which units are loaded and unloaded:



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


POS System
U
M
External Systems
U
M
Telesales System
U
M
Warehouse System
U
M
M
eCommerce System
U
Legend:
M indicates an unresolved reference to an unloaded controlled unit
U indicates that package is a controlled unit

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, Rose restores the snapshot by loading the specified controlled units and opening the correct diagrams.

If you are working 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 might want to define one model workspace that displays the most important analysis diagrams and controlled units, and another model workspace that displays the most important design diagrams and controlled units. Each workspace is different but uses the same model.

It is also important to note that saving a model workspace will not affect how the model is loaded on another machine. If a co-worker wants to load a model using a model workspace you defined on your machine, the co-worker must have a copy of the model workspace and model located in the same folder on his or her machine.

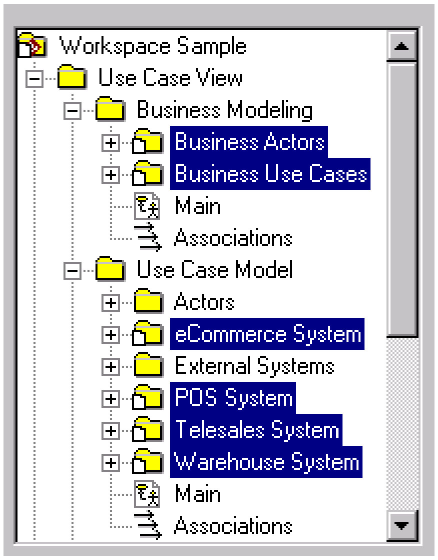
By default, Rational Rose will name the workspace <model name>-<Operating System User Name>.wsp. For example, the name of a saved model workspace might look like MyModelName-JillUser.wsp.

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

Workspace Example

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

A new software developer has just joined a distributed team that is working on a very large model containing over 200 controlled units. Through the course of the next several months, the new developer will model several systems in the Use Case Model and will modify the Business Actors and Use Cases (as shown in browser illustration). In order to help the new developer, the team's project manager created a model workspace that will load all of the units the software developer will be responsible for, as well as some of the more important diagrams.



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

The model workspace will help 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 Rose only has to load six out of 200+ 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 to create efficiency.

Creating and Saving a Model Workspace

To create a model workspace you load all controlled units that you will want to restore when loading this workspace. Open all diagrams to restore when loading this workspace. 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 Rose workspaces folder.

Loading a Model Workspace

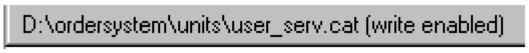
To load a model workspace, click **File > Load Model Workspace**. Select the name of workspace file (*.wsp) to load.

Protecting Controlled Units

When loading a controlled unit into a model, Rose makes the unit write-protected or write-enabled depending on the file's current status in the file system. Thus, a controlled unit, which is read only in the file system, is write-protected in the model and Rose prevents you from modifying it. This means that the toolbox is grayed out on all of its diagrams and you are not able to 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 is displayed in the Rose window's status bar when selecting the controlled unit in the browser. For example:



D:\ordersystem\units\user_serv.cat (write enabled)

If you are using a version control add-in, Rose handles the write-protection of controlled unit automatically. Thus, a checked-in unit is automatically write-protected.

There may be situations when you may 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 it. Manually write-protecting the unit assures that you will not inadvertently change it. Once you reload the model, write-protection no longer applies, and you will be able to edit the file.

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

To Write-Protect a Controlled Unit

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

- If you are using a version control system, place the controlled unit under version control and have it checked-in.
- Make the file read-only by setting the file protection of the .cat file to read-only in the underlying file system.
- By right-clicking on the controlled unit in the browser, and clicking **Write Protect** on the **Units** menu.

To Write-Enable a Controlled Unit

To write-enable a controlled unit under version control, click **Tools > Version Control > Check Out**. This will check the unit out 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, make sure that the controlled unit is write-enabled. If it is under version control, you must check the file out. 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 should belong to the new package (including the associations) from the unit into that 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 but NOT the associations. The associations must be moved manually.*

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

Create two new packages in the package that correspond to the controlled unit, then move the contents from the unit into the two packages by using drag-and-drop in the browser. Make sure to 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 the Model Integrator found on the Tools menu.

For more information, see Chapter 3.

Adding Controlled Units to a Model (Importing/Loading)

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

Rose imports controlled units to the diagrams that currently have focus control. Thus, be sure you are in the diagram where you want to import the controlled units. The import will fail if you import to the wrong type of diagram. (For example, you can't import .cat files to a deployment diagram.)

Note that when you import a controlled unit, Rose tries to resolve any references. If an element has a reference that Rose cannot resolve, the problem is logged.

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

To import a controlled unit, click **File > Import**. In the dialog box, be sure to 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 contents of the uncontrolled unit's file is inserted into the enclosing file.

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

If you are using one of the version control add-ins, 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 are not using 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 still exists in the file system.

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

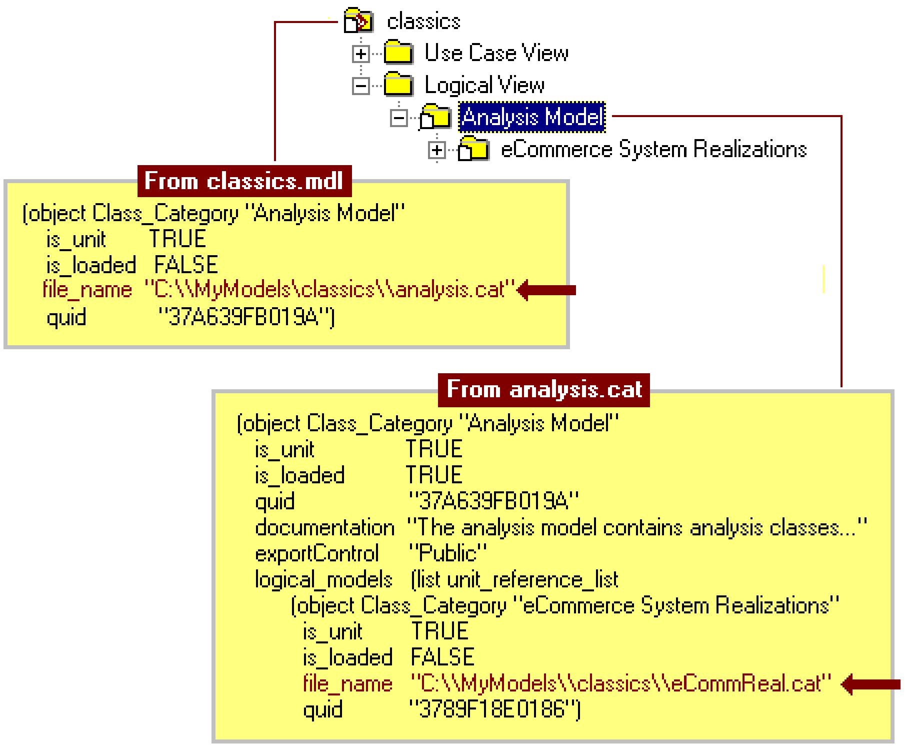
Creating Virtual Paths to Controlled Units

Understanding Virtual Path Maps

Rose provides the ability to define symbolic names for file paths. Each user can control how these symbolic or virtual names are defined in his or her 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. (Note that 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.



From classics.mdl

```
(object Class_Category "Analysis Model"
   is_unit     TRUE
   is_loaded   FALSE
   file_name   "C:\\MyModels\classics\\analysis.cat"
   quid        "37A639FB019A")
```

From analysis.cat

```
(object Class_Category "Analysis Model"
   is_unit          TRUE
   is_loaded        TRUE
   quid             "37A639FB019A"
   documentation    "The analysis model contains analysis classes..."
   exportControl    "Public"
   logical_models   (list unit_reference_list
      (object Class_Category "eCommerce System Realizations"
         is_unit     TRUE
         is_loaded   FALSE
         file_name   "C:\\MyModels\\classics\\eCommReal.cat"
         quid        "3789F18E0186")
```

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

How Virtual Paths Work

When Rose reads from or writes to a model, it tries to substitute every absolute path with a virtual path. When Rose opens a controlled unit, or uses a path specified in a model property, each virtual path is converted 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 his or her "X" drive, Rose resolves the internal reference to the controlled unit and loads the following file:

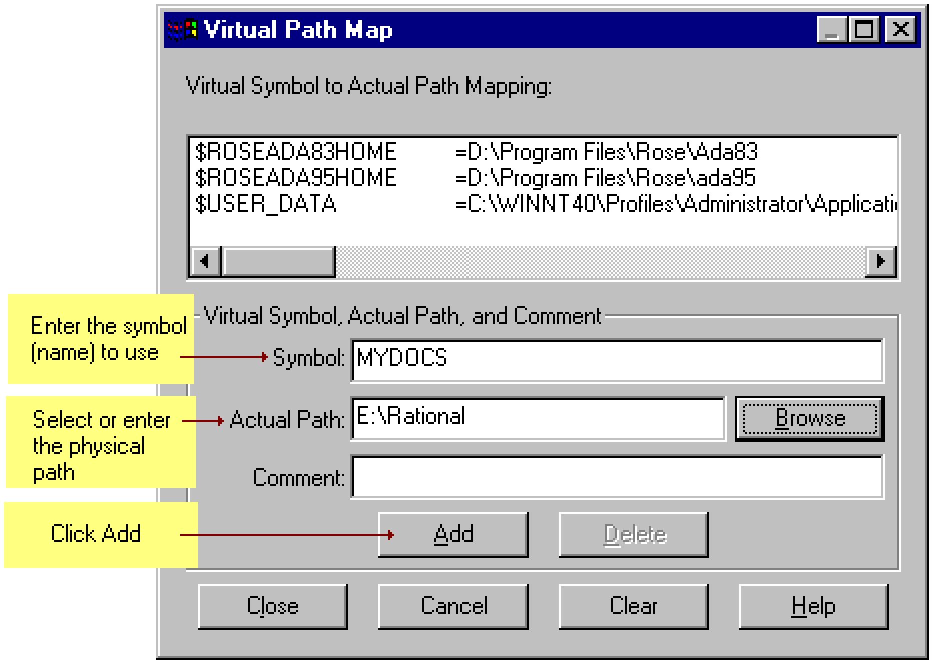
```
X:\MyModels\classics\analysis.cat
```

Once you create virtual path maps, when anyone on a team opens or saves a model, Rose will try to match the longest possible file path to the symbols in the path map and will keep trying, so you can end up with concatenated path map symbols.

Note: *Each user that is going to work on a model will have to define the same path map symbols before opening the model. For example, another user with the private workspace Y:\MyModels, must define $MYPATH=Y:\MyModels. It is also recommended that you not use path maps to point to network drives or shared files.*

How to Create Virtual Path Maps

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


Virtual Path Map
Enter the symbol (name) to use
Select or enter the physical path
Click Add
Browse
Add
Delete
Close
Cancel
Clear
Help

Virtual Symbol to Actual Path Mapping:

```
$ROSEADA83HOME     =D:\Program Files\Rose\Ada83
$ROSEADA95HOME     =D:\Program Files\Rose\ada95
$USER_DATA         =C:\WINNT40\Profiles\Administrator\Applicati
```

Virtual Symbol, Actual Path, and Comment

Symbol: MYDOCS

Actual Path: E:\Rational

Comment:

Note that you do not enter the $ before the Symbol name. (It's added for you.) When you click Add, the new path map is added to the list of existing path maps at the top of the dialog.

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, suppose you have created 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 is saved, the reference from the model file to the package is stored as:

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

When the model is opened 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 simply adding the path map $MYPATH=$ROOT\MyModels.

Defining a Path Map with Wildcards

A wildcard character (*) in the path map can be used 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 his or her own set of model files within each subsystem, then a controlled unit belonging to the display subsystem may have the following path:

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

The model file will refer to the unit as:

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

When the model is opened by user "susanne," who has the following virtual path definition:

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

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

```
\server\models\project\display\susanne\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.

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

Using Virtual Paths for the Value of a Model Property

Rose does not convert actual paths in model properties to virtual paths. In order 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 that are 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's not possible, you will need to 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, 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
```

Then, 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, Rose will not automatically convert the absolute to a virtual path when you save your model. (Note that it does automatically do the conversion for controlled units and .mdl files.)

To get around this, you simply need to move the artifact to another part of your model, move it right back to its appropriate location, then save the model. (There's 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 do the same thing in order for Rose to register the change.

Where Virtual Path Maps are Stored

Virtual path maps are kept in two places in your system registry: the users area and the system area. A user can typically see and access only the virtual path maps in his or her area of the registry.

There are also system virtual path maps that are in H_KEY_LOCAL_MACHINE. You need to be administrator on the machine to edit these virtual path maps.

Checking References and Access Violations

Once 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**) traverses the entire model looking for unresolved references and places the results into the log. It is designed to be used when you are saving 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, but there isn't an 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 of any kind of dependency, generalization, association, realizes, instantiation, etc.
- 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 get larger, access violations become more important and 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 direction as 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 sited 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 selecting Browse.

Organizing Controlled Units for Teams

When a model is shared among teams of developers, it's essential that the model be partitioned so that it can evolve in a controlled manner. One of the keys to successfully sharing a model is 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 the configuration management in your project. The granularity level at which you want to be able to perform version control may define what becomes a controlled unit. For example, all packages could be made 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, use **Report >Show Access Violations**.

For more details about model architecture, see Chapter 5.

Chapter 3

Comparing and Merging Models

About the Model Integrator

The Rose Model Integrator is a tool for comparing and merging Rose models. The 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.

Model Integrator can also be used to view the contents of a single model file. Model Integrator provides a different way of looking at the model than the view provided by Rational Rose. Model Integrator provides a low-level textual view of all the model elements and their properties. This way of examining a model can be a quick way of viewing all the property settings in use.

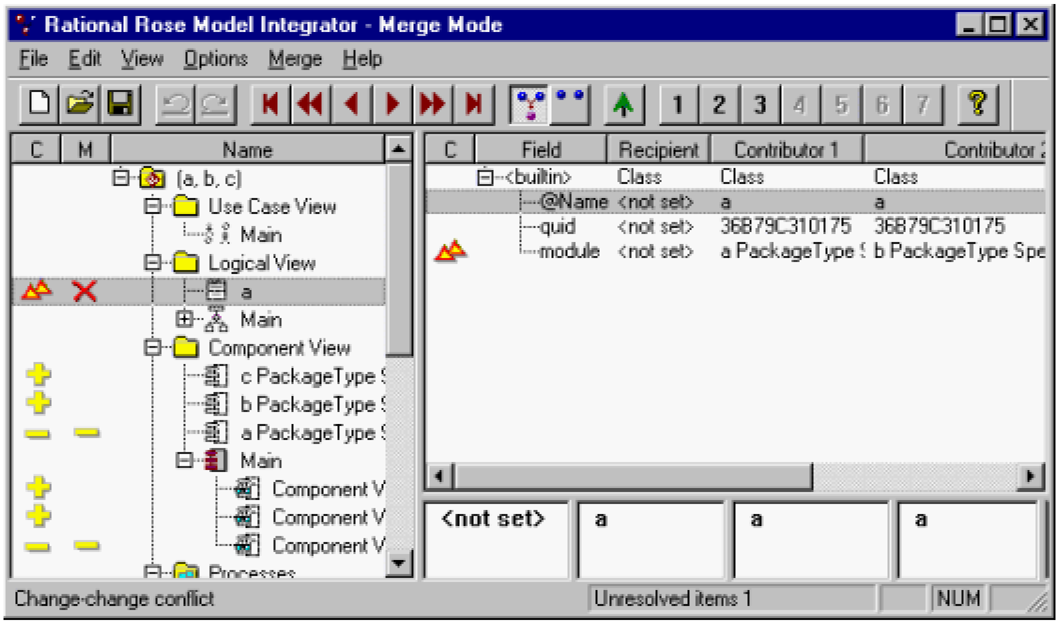
Model Integrator runs in two modes:

- Compare mode
- Merge mode

As described later in this chapter, you can switch between modes.

About the Model Integrator Interface

The Model Integrator runs outside of Rose and provides its own interface:



C	Field	Recipient	Contributor 1	Contributor 2
	<builtin>	Class	Class	Class
	@Name	<not set>	a	a
	quid	<not set>	36B79C310175	36B79C310175
	module	<not set>	a PackageType S	b PackageType Spe

Change-change conflict

Unresolved items 1

There are three major components to the interface:

- Browser view
- Property view
- 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 the display in Rose. Model Integrator displays some objects that Rose hides from your view. See *About Model Files* for a brief discussion of the objects Model Integrator displays.

Note that 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 tries to partner objects that are the same across all the models. If all of the contributors have the same model element located at the same place, the browser will only display 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 will be 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*.

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 views

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 leftmost 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 along the top 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.

About 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, you will be asked whether or not 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 (**Contributor 1**) has special significance to Model Integrator - it is the **base model** used for comparing the differences between the other models.

About the 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 the Model Integrator highlights the differences and conflicts between two or more models. You can switch back and forth between Compare mode and Merge mode, so 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 are disabled.

Merging Models

Merge mode incorporates all of the features of Compare mode, along with additional information to support the decisions you need to make in order 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 found between the models to be merged.

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

About Differences and Conflicts

Model Integrator uses the base model (Contributor 1) to identify what kinds of changes were made to the models being compared or merged. Each contributor is first compared to the base model. The Model Integrator displays additions, changes, and deletions between a contributor and its base model as differences. Symbols identify the types of differences that are 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 one of the different contributors should be incorporated 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 will automatically incorporate differences into the recipient model. However, it requires you to 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 will be displayed as a difference. Conflicts still have the same meaning in this mode.

About 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 Rose. The following is a brief introduction to the kinds of objects Rose models are made from.

Basic Objects

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

Diagram Objects

Each of the diagrams you create in your model is an object. They are displayed differently in Model Integrator than in Rose. The diagram titles will be 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 will be new because normally Rose does not show them to you. These objects include the view objects described below.

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, described below.

Mechanism

A Mechanism is another normally hidden part of the model that contains a set of objects used internally to implement parts of the model you have 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 Rose for each object when the object is first created in the model. quids are unique, so that they can be used to identify an object even 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 the objects have a common ancestry.

References

Much of the power of the 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 you will see 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 is completed. When objects are deleted or moved, Model Integrator must check to make sure that references from other objects are still valid. This semantic checking function is performed before the model is saved.

Unnamed Objects

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

Often a model will contain many unnamed objects which you are not aware of because Rose never shows you the $UNNAMED$ string. Model Integrator does display the actual name of every object, even unnamed objects. Usually you can tell what an object is by looking at its icon in the browser view, its properties (the object type will be at the top of the property view), and in some cases by looking at the children of the object.

Rose Model File Versions

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

It is a good idea to only merge models that have the same model file version numbers. This avoids the problem of creating merged models that declare themselves to be one version, but which contain some model elements accepted from contributors which may be incomplete or different from the expected version. Model Integrator itself is independent of model file versions, and it 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 make sure 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 it disables merge choices that would introduce errors into the model.

For example, suppose that your base model contains class A. Contributors 2 and 3 each make a change to one of the members of this class, while contributor 4 deletes the class. If you have already accepted changes from contributor 4 to delete the class, it does not make any sense to allow you to accept one of the changes that contributors 2 or 3 made to the class. However, with semantic checking turned off, Model Integrator will allow you to make these contradictory changes. Model Integrator would not discover the problem until either you decided to save the recipient, or you used 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 hard to remember exactly which contributors present valid choices at a particular node of the model. This problem can also arise when Model Integrator has made 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 to avoid these problems by making your choices clear at each step.

When semantic checking is activated, and the user moves the current selection to a new node of the model tree, the checker determines which choices of contributor (if any) would result in an invalid model if they were chosen by the user. These choices are then disabled in the interface by disabling 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 a change now and fix it later. In this case, semantic checking can be disabled. Merge choices can then be made in the normal manner. Once the merge choices are made, the user can then select the **Merge > Check Merge** menu function to check and repair the model. The model will always be checked for validity before it is written to disk.

Limitations of Semantic Checking

For performance reasons, on-the-fly semantic checking is limited to checking only the nodes of the model you are viewing, and only when you view them. Consequently it is still necessary to perform a check of the whole model before it is saved to disk, and this check may reveal errors that need to be corrected.

A more important limitation that affects both kinds of semantic checking is that references to subunits that are not loaded into the merge session cannot be checked.

About Memory Requirements and Performance

For a typical merge operation, Model Integrator must load three 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, this will not be a problem, but if you have 30 MB (megabyte) set of models to merge, this can put a strain on your system. A typical sign of a serious memory deficiency is that loading the models is extremely slow (Model Integrator may even appear to be frozen) while the disk drive is constantly busy. This condition is known as thrashing. It occurs because Model Integrator constantly requires access to the entire data set of all the models you are merging, but because of the physical memory shortage, much of this data is stored in virtual memory on your hard disk (in your computer's pagefile or swap file, depending on which operating system you are using). The computer ends up spending all of its time reading and writing the disk, and very little real work is done. 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 Model Integrator's performance:

- Configure your computer with enough RAM memory to meet or exceed the 5x requirement stated above. If you have 30 MB of models to merge, you should have at least 150 MB of RAM memory in your computer. Anything less will compromise performance, as Model Integrator will have to store its data on the disk.
- If there is not enough physical memory to meet Model Integrator's requirements, make sure that you have allocated enough virtual memory to accommodate Model Integrator's needs. Consult your operating system documentation or ask a system administrator to adjust the available virtual memory.
- Close other programs to free up memory. Even if you have a lot of RAM and virtual memory in your computer, other programs may be claiming large portions of it. In some extreme cases, applications may load system components which are not unloaded even 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.
- Your operating system comes with tools to measure and report on memory usage. For example, in the Windows NT environment, a simple tool to use is the Task Manager and its Performance page.

About 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 on a model file version displayed in the ClearCase Version Tree Browser window to bring up a context menu and select **Compare > with Previous Version**. ClearCase will invoke Model Integrator to display the differences. Or, from Windows Explorer you can right-click on 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's likely that the ClearCase integration with Rose hasn't been set up. See Chapter 4 for instructions.

Merging Whole Models with Controlled Subunits

Currently 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 that 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. Currently the way this is performed is 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 which will receive the merge result.

Starting Model Integrator in a ClearCase Integration

Model Integrator is started, not from a ClearCase menu, but from the Rose Tools menu or by the standard method for the system you are using (e.g., 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

Do one of the following to start Model Integrator:

- From within Rose, select **Tools > Model Integrator**.
- In Windows, Model Integrator appears on the Start menu as part of the Rose installation. You can select the Rational Rose Model Integrator shortcut to begin working with Model Integrator without starting Rose itself.
- From a UNIX shell process or a Windows console process, enter
  ```
  modelint file.mdl
  ```
  and click **Return**. For more information about the command line interface, see *Using Model Integrator from the Command Line* (Note that the directory containing the Model Integrator executable must be in your path for this to work).
- You can also start Model Integrator from Rational ClearCase as part of a ClearCase compare or merge operation. See *About Model Integrator and ClearCase*.

Preparing Models for Merging

Before merging models, it is a good idea to check each model with the Rose **Tools > Check Model**. If errors are reported, those errors should be corrected before performing a merge with Model Integrator.

Selecting the Contributors

The easiest way to specify contributor files is to drag and drop the files from the Windows Explorer onto the Model Integrator window (Windows platform only). If the Contributors dialog is not open, Model Integrator will open it for you. If it is open already, then you must drop the files onto the dialog box, not the main window.

Alternatively, select **File > Contributors** to display the Contributors dialog. 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 the **Browse** button at the top of the Files list control and use the file browser to find a file to add to the list.

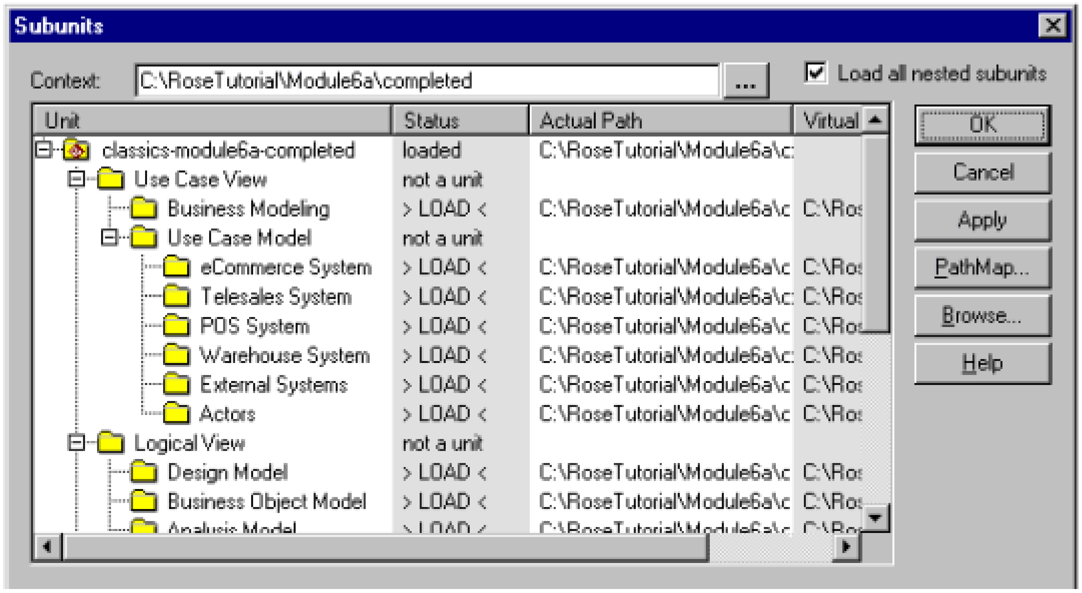
2. Click **Enter** to confirm the file name.
3. Click the (New) button to create a new file input field.
4. Repeat steps 1 through 3 until all files are specified
5. Click **Compare** or **Merge**.

Note: *If the Compare/Merge Against Base Model checkbox is checked, then the first specified file must 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 on it, then click the arrow key 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.*

Loading or Unloading Controlled Units

If one or more of the contributor files you specify have controlled units, Model Integrator displays the Subunits dialog. This dialog 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.



Context: C:\RoseTutorial\Module6a\completed

[✓] Load all nested subunits

Unit	Status	Actual Path	Virtual
classics-module6a-completed	loaded	C:\RoseTutorial\Module6a\c:	
Use Case View	not a unit		
Business Modeling	> LOAD <	C:\RoseTutorial\Module6a\c	C:\Ros
Use Case Model	not a unit		
eCommerce System	> LOAD <	C:\RoseTutorial\Module6a\c	C:\Ros
Telesales System	> LOAD <	C:\RoseTutorial\Module6a\c:	C:\Ros
POS System	> LOAD <	C:\RoseTutorial\Module6a\c	C:\Ros
Warehouse System	> LOAD <	C:\RoseTutorial\Module6a\c:	C:\Ros
External Systems	> LOAD <	C:\RoseTutorial\Module6a\c	C:\Ros
Actors	> LOAD <	C:\RoseTutorial\Module6a\c	C:\Ros
Logical View	not a unit		
Design Model	> LOAD <	C:\RoseTutorial\Module6a\c	C:\Ros
Business Object Model	> LOAD <	C:\RoseTutorial\Module6a\c	C:\Ros
Analysis Model	> LOAD <	C:\RoseTutorial\Module6a\c	C:\Ros

Subunit Status

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

Status	**indicates...**
loaded	This subunit has been loaded successfully.
not a unit	This entry is not currently a separate subunit. This model section is part of the main .mdl file.
LOAD	Model Integrator will load this entry when you click OK or Apply.
SAVE	Model Integrator will save this entry to a separate file when you click OK.
unloaded	This subunit will not be loaded.

Loading Subunits

Subunits for each contributor are loaded separately, so you will get a separate Subunits dialog for each contributor .mdl file that has subunits. You can change the Status value back and forth between *LOAD* and *unloaded* by clicking on 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 will be skipped. If you do not want to load any subunits, click the **Cancel** button in the dialog.

When you complete one dialog and click **OK**, Model Integrator tries to load the subunits you have specified with the LOAD Status field value. If there is an error and some of the subunits cannot be loaded, the Subunits dialog will be redisplayed.

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

Each contributor with subunits will bring up its own Subunits dialog. When you complete the final Subunits dialog, Model Integrator immediately begins the Compare or Merge session.

Saving Subunits

When you save a model using **File > Save** or the save icon on the toolbar, Model Integrator will also save your subunits to the same place relative to the main .mdl file's location. In this case, the Subunit dialog will not be displayed. 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 will be displayed. It allows you to:

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

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

Subunit File and Path Names

The Subunits dialog 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 try to load the subunit.

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

Model Integrator shares path map variables with Rose and will use 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.

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

When saving a subunit, we recommend that you define a path map variable (in the PathMap dialog) and set it equal to the value &. This will prevent 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.

How to Resolve Subunit Loading Errors

If you tell Model Integrator to load a subunit, but the load fails, Model Integrator will display the Subunits dialog again to allow you to correct the problem. The Status column of the dialog will show you the current status of each subunit. You may need to scroll the dialog box down to find a subunit that has not loaded. It will still display the LOAD status.

At this point 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 the **PathMap** button and modifying the variable in the Pathmap dialog.
- You can first select the subunit in the list and then click the **Browse** button to open a directory browser window and use it to look for the file. Select a file in this window and click **OK**. The filename will be placed in the Subunit dialog.
- You can change the current directory for path map variables that take the value & by changing the **Context** field located at the top of the Subunit dialog.
- You can elect to not load the subunit. Click on the **Status** field for the subunit. You should see the status change to “unloaded.” The subunit will not be included in the merge.

Setting a New Context for Subunits

The Context field is displayed at the top of the Subunits dialog. This field shows the default path that Model Integrator will use to substitute for the & path map symbol (see Chapter 2 for a description of how to use path map symbols).

If you have created your models using a virtual path map, you can define the value of the symbol to be &. When Model Integrator sees the & symbol in the definition of a path map, it replaces the symbol with the actual path specified in the **Context** field.

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

You can select a new **Context** path by either entering a new value directly into the **Context** field, or by clicking the **Browse** button to the right of the field. This button brings up a standard file browser dialog you can use to find the drive and folder you want.

Using Compare Mode

Use compare mode to scroll through the model and observe the differences between the contributors. If you decide you want to merge the models, change the mode to merge mode or exit the program when you're 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 the following table for a typical case of three contributors (not shown is a move operation, but it behaves like a change).

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.*

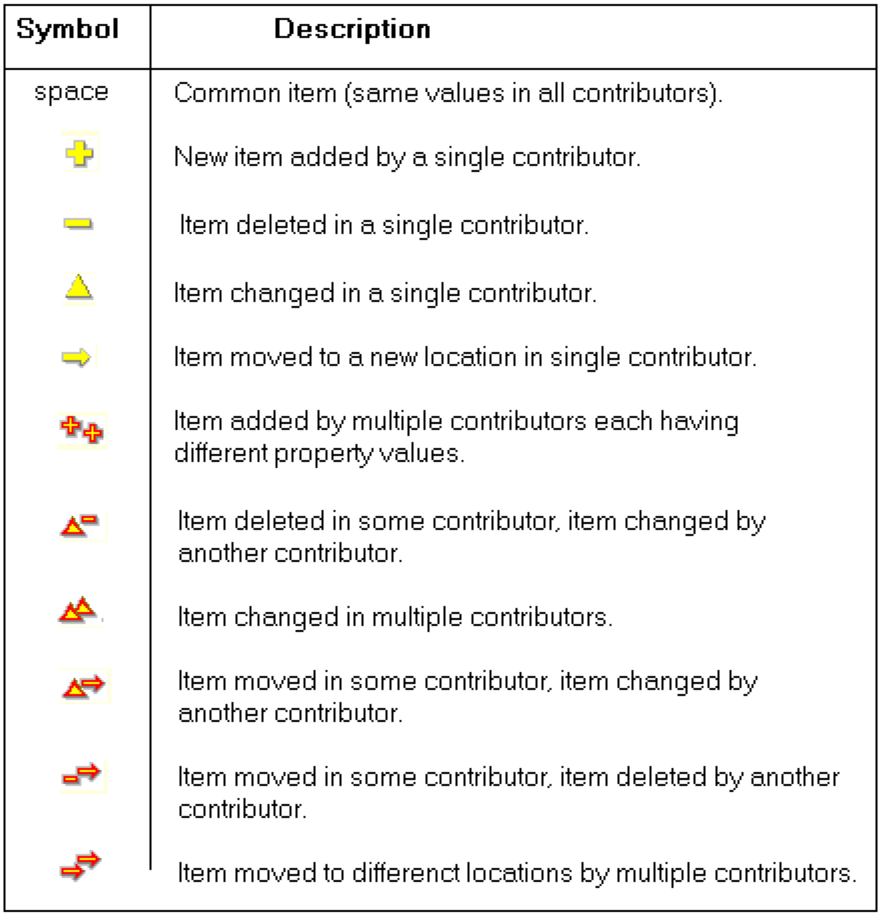
The essence of this procedure is: if a contributor that is not the base model introduces a change (adds, modifies, moves, or deletes an object), that change is copied to the merged output instead of the original object. However, if two or more contributors change the same thing, then the AutoMerge procedure 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, you will see a message 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.

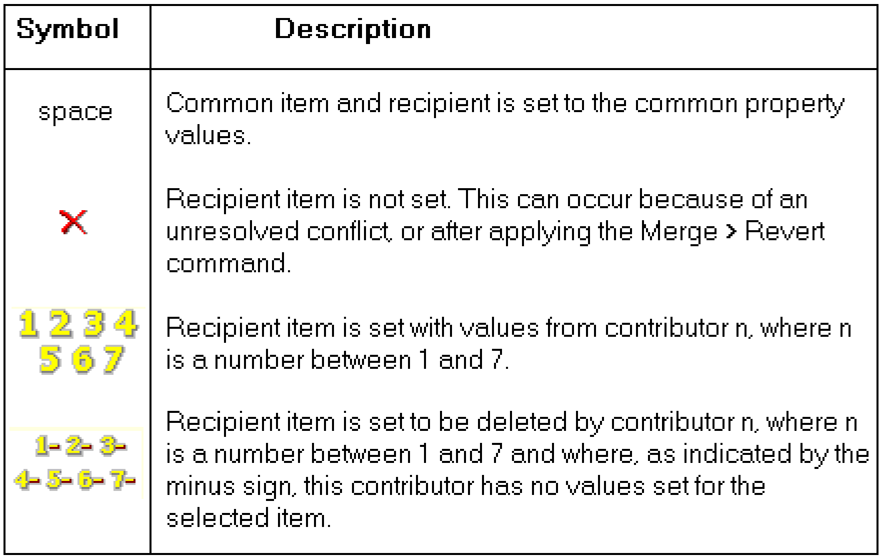
Interpreting Compare and Merge Results

Model Integrator shows you the results of comparing the contributing models by displaying an icon to the left of each node in the browser view. Icons indicating the results of comparing the models appear in the C column. The following table depicts the Compare status icons and their meanings. Note that the icons for differences are yellow, and the icons for conflicts are red.



Symbol	Description
space	Common item (same values in all contributors).
✚	New item added by a single contributor.
▬	Item deleted in a single contributor.
▲	Item changed in a single contributor.
⇨	Item moved to a new location in single contributor.
✚✚	Item added by multiple contributors each having different property values.
▲▬	Item deleted in some contributor, item changed by another contributor.
▲▲	Item changed in multiple contributors.
▲⇨	Item moved in some contributor, item changed by another contributor.
▬⇨	Item moved in some contributor, item deleted by another contributor.
⇨⇨	Item moved to differenct locations by multiple contributors.

Symbols indicating the status of a Merge operation appear in the M column. The following table depicts the Merge status icons and their meanings.



Symbol	Description
space	Common item and recipient is set to the common property values.
✗	Recipient item is not set. This can occur because of an unresolved conflict, or after applying the Merge > Revert command.
1 2 3 4 5 6 7	Recipient item is set with values from contributor n, where n is a number between 1 and 7.
1- 2- 3- 4- 5- 6- 7-	Recipient item is set to be deleted by contributor n, where n 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: *The 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**.

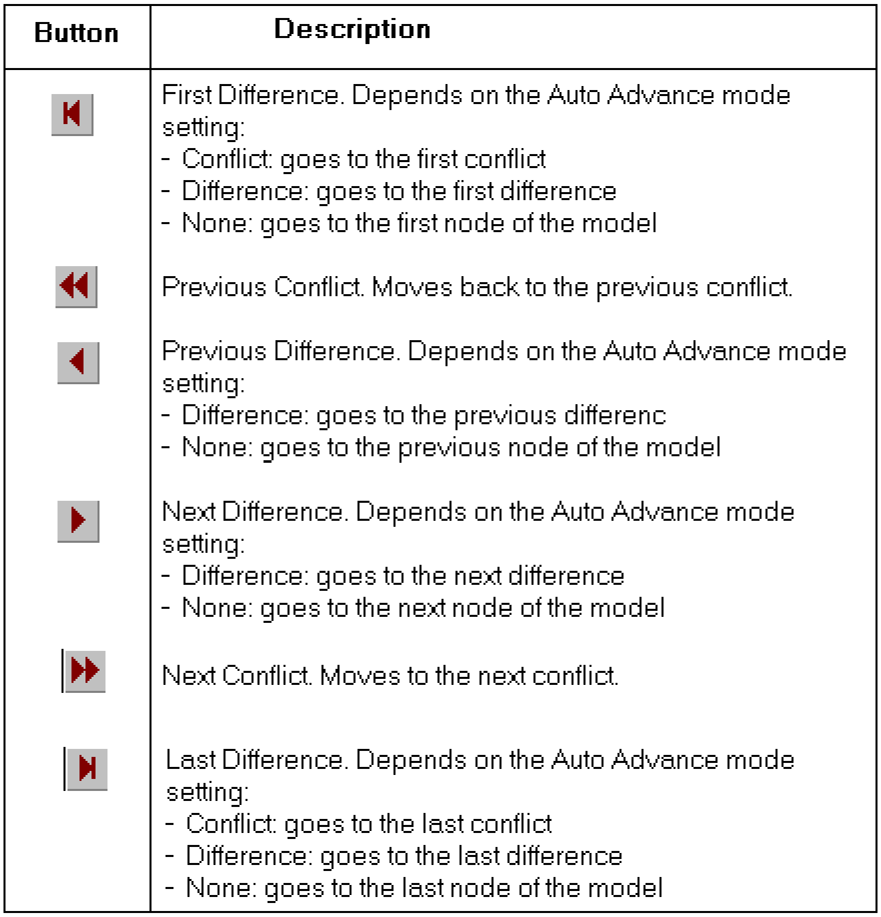
The search starts at your current location in the browser window and proceeds through all the nodes in the model that are displayed in the browser window. (Use **Edit > Expand All** to display every model object in the browser.) If the string is found, the browser window scrolls to display the desired 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. If the string can not be found, the function will beep. Once 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, making sure that you visit all the conflicts and differences. These commands will automatically expand the browser tree to make the next conflict visible, so you do not have to go hunting for it.

Some of these commands, as noted in the following table, operate in the same mode as the setting for Auto Advance mode. The text displayed in the menus and tool tips will change to reflect this.



Button	Description
⏮	First Difference. Depends on the Auto Advance mode setting: - 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: - Difference: goes to the previous differenc - None: goes to the previous node of the model
▶	Next Difference. Depends on the Auto Advance mode setting: - Difference: goes to the next difference - None: goes to the next node of the model
⏩	Next Conflict. Moves to the next conflict.
⏭	Last Difference. Depends on the Auto Advance mode setting: - 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 models have 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 have moved items from one place to another within your model (for example, by using drag and drop editing or by using the clipboard within Rose). When you merge models with elements that have been moved, Model Integrator will display 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 selecting the **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 will be disabled in the menu.

You can use the **View > Previous Location** menu item to quickly 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.

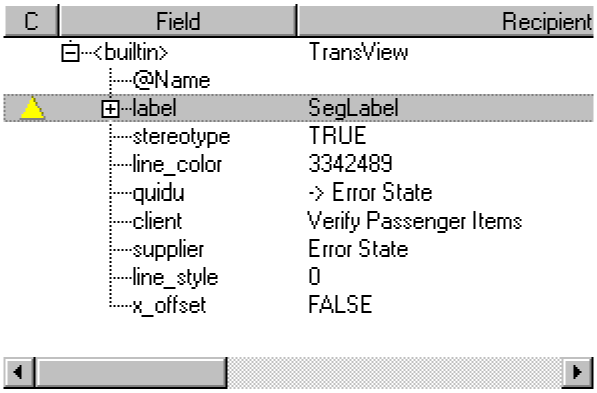
While merging models you may need to view the parent of a node you are looking at, but if the model is large, the parent node may not be visible on the screen. Use the **View > Parent** menu function to quickly bring the parent node into view. You can use the **View > Previous Location** menu function to quickly return to the node you were originally viewing.

View Nodes Referenced by this Node

It is not uncommon for a particular node of a 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. The **View > Referenced Node** command makes this easy to do for those nodes which have one or more of the three common types of references: client, supplier, and quidu.

Note: *For our purposes, these reference types are not important in and of themselves. They are used internally within the Rose model and their meaning changes depending on the node viewed. The only real significance they have in Model Integrator is that they link two different objects in the model together.*

Nodes that have these references will display them in the property view. Shown below is 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, but this command makes accurately finding it easy.

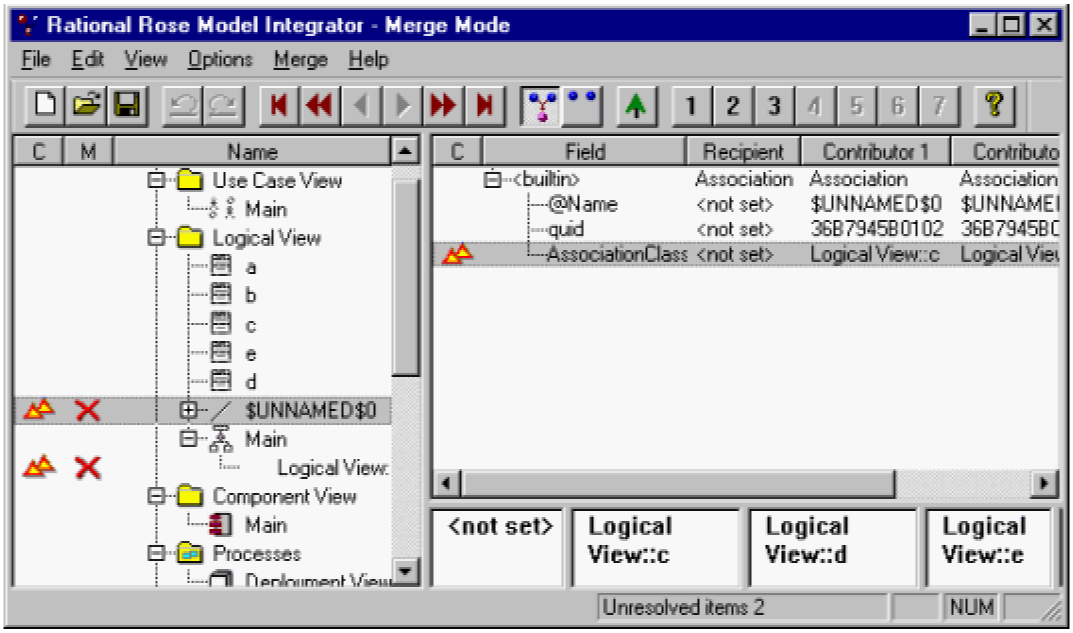


C	Field	Recipient
	⊟ <builtin>	TransView
	@Name	
⚠	⊞ label	SegLabel
	stereotype	TRUE
	line_color	3342489
	quidu	-> Error State
	client	Verify Passenger Items
	supplier	Error State
	line_style	0
	x_offset	FALSE

When a node in the model contains any of these references, the **View > Referenced Node** menu will be 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 the Model Integrator have already accepted a change for the referenced node, the Recipient menu item will be active. Normally you would choose to view this one because it is the one that will be saved in the merged model. If the Recipient choice is not active, that means the referenced node is an unresolved item.

Accepting Changes from Contributors

The results of the Merge are displayed in the main Model Integrator window, as shown in the following example.



C	Field	Recipient	Contributor 1	Contributo
	<builtin>	Association	Association	Association
	@Name	<not set>	$UNNAMED$0	$UNNAMEI
	quid	<not set>	36B7945B0102	36B7945BC
	AssociationClass	<not set>	Logical View::c	Logical Viei

<not set>	Logical View::c	Logical View::d	Logical View::e

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 of course deciding which changes you want to keep, and which to throw away. There are a few simple rules you can follow that will make this job easier:

- Merge often. Do not wait until you have forgotten why you did something.
- Partition the work and the model so that people can work on different parts without stepping on each other's toes. This will reduce the number of conflicts you have to resolve.
- Know the models you are merging. You should try to know in advance which of the contributors you want to select for major components of the model, such as classes and diagrams. This will help guide the finer-grained choices you must make.
- You may encounter internal parts of the model that you do not necessarily understand. There is a simple rule of thumb for making merge decisions about these objects that are normally hidden: Choose the same contributor for these items as you are selecting for the related items you are familiar with.

  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 will maintain 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:

1. 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.

2. Resolve an individual conflict or difference by selecting its node and then using **Merge > Resolve Selected Nodes Using** or one of the corresponding toolbar buttons. This function copies one of the available contributor choices to the recipient. Unlike Resolve All Conflicts, it operates on any node that displays either a conflict or a difference, and previous choices are overwritten.

   This function can also be used with 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 will change values that have been set previously. Warning: Subtree mode is very powerful. Use it with care.

Note that when semantic checking is enabled, Model Integrator will disable choices of contributors which may produce errors in the recipient model. If you wish to make the change anyway, you must turn off **Merge > Semantic Checking**.

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

Changing Nodes with Differences

You can accept changes from nodes which do not have conflicts, but do have differences. In this case, Model Integrator's AutoMerge procedure has already made a choice for you. The choice of contributor is not displayed in the M column of the browser window for clarity's sake, but you can see which contributor is currently chosen by looking at the property view. The Recipient column displays the values for the chosen contributor. The AutoMerge choice will be the contributor that is different from the others.

You can override this choice by selecting the node in the browser and applying **Merge > Resolve Selected Nodes Using** to select a different contributor. The effect of this is to not accept the change because you are choosing a contributor that did 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 use **Edit > Undo** to restore it to its original state. However, if you changed some time ago in your merge session and you do not want to undo other work you have completed, there is another choice. The **Merge > Revert Selection** command will restore a node to the unmerged state.

The effect of this command is to make the node unresolved whether it is a conflict or not. The M column for this node will change 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 can only be applied to nodes that have been reverted. Applying the AutoMerge command to reverted nodes will restore 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 both the current node and to all of its children. Subtree mode can be activated through either a toolbar icon, or by toggling **Merge > Subtree Mode.**

With Subtree mode turned off, you can visit each subtree node and make independent choices of contributor for each node. With Subtree mode turned on, Model Integrator will automatically apply the selected command to all of the children of the current node.

Merge mode commands that are 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 care, and remember to turn it off when you're done with it. But if you forget, you can always use **Edit > Undo** to undo any unwanted changes.

Using Semantic Checking On-the-Fly

Semantic checks are performed by the Check Merge function before you save the merged model, but they are also available while you work. Selecting **Merge > Semantic Checking** will enable Model Integrator to perform some reference checking when you select a new node in the browser to visit. Based on this check, Model Integrator will disable merge choices that will result in merge errors later in the session.

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

When a Contributor is Disabled by Semantic Checking

When this happens you have two choices:

- 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**). Usually 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 the choice you want.
- Turn off Semantic Checking, and make the choices you want. Rely on the Check Merge function to find any errors when you are done with your merge.

Checking Merged Model for Consistency

Use **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 you are merging deletes model elements that are still in use by one of the other contributors to your merged model. This can occur because of:

- Decisions you make when you resolve conflicts between contributors
- Decisions made by Model Integrator’s automatic merging feature

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 will open.

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 will not allow you to save it.

Correcting Merge Errors

The Merge Errors dialog floats above the main Model Integrator window. It's purpose is to provide a set of tools that can help you find and correct errors that Model Integrator has 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. Then, in general, to correct a merge error, you must select a different contributor for some node of the model (see *Accepting Changes from Contributors*). The Merge Errors dialog has buttons that will help you to find the node you need to change:

View Error - Takes you to the node of the model where the error was discovered. We will call this the error node.

View Definition - Takes you to the node of the model which 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 function to search for the parent of a definition node. This button is used when you have a node whose parent is deleted.

View Other Locations - If the node you are viewing has been moved to different locations by a separate contributors, this function will take you to the one of the other locations where the node exists. Only one of these locations will actually exist in the merged output model; the other nodes are marked for deletion. This button is used when you have a forward reference error.

Refresh List - This command clears the error list and recomputes the Check Merge function. If new errors are found 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 will be reported at a time. Sometimes you will fix one error and that will fix other errors as well. Refresh List will always display 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 node that is deleted.

The error node references another node in the model that has been deleted in the merged model. The View Definition button will display the location where the deletion occurs. This 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 which 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 will be deleted as well, so you must change the parent node so that it is not deleted. Use the View Definition button to go to the defining node in the model. Then use the View Parent button to move up the model tree until you find the parent node that is being deleted. Choose a contributor for this node that contains a definition of the node rather than deletes it.

This node has a forward reference

The error node references another node in the model which has been moved backward in the merged model (i.e. the definition previously occurred before the reference in the original model, but now it occurs after the node that references it). Certain forms of Rose model references are only allowed to nodes that are defined first in the model. The View Definition button will display the location where the node originally was located. You can use the View Other Locations button to find out 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. If you wish to move the nodes to the new location, you must do it in Rose after the merge is complete. This will allow all of the references and definitions to be updated properly.

Saving the Results

Once the number of unresolved items is zero, you can begin saving the model. Click the toolbar button to initiate the save operation. First, Model Integrator will check the model for errors. If errors are found, you must correct them before the model can be saved. The Merge Errors dialog has the tools and help topics to help you correct these problems. After you are finished, close the Merge Errors dialog and save again.

Model Integrator will ask you to specify where to save the main model file. Select a name and a directory for the output.

If your model has subunits and you loaded them, the Subunits dialog appears to let you save the subunits. Click **OK** to continue the save operation. Once the Subunit dialog closes, the merge is complete and saved.

Performing a Partial Merge

You may have confined your editing in Rose to only a part of the model, but when you load the model into Model Integrator, differences appear in other parts of the model that you may not expect. This is not an error on the part of 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 part of the model, here is a procedure you can follow to do so. It requires that you use a base model, which will 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 or by selecting **Options > Merge Mode**.
2. Turn subtree mode on by selecting **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 will cause the base model to be selected for all conflicts and differences in the entire model. The M column for the entire model will change to the **1** icon (nodes that were added by other contributors will change to the **1-** icon).
5. Select the part of the model to be actively merged. 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 (in other words, use the standard way of selecting multiple items with the mouse). If you are selecting with the mouse, make sure that you have expanded the model tree so that all nodes can be selected (use **View > Expand All**).
6. Apply **Merge > Revert Selection** to this part of the model. This entire part of the model should now display the **X** icon for each node.
7. Apply **Merge > AutoMerge Selection** to the same part of the model.

Now you have successfully restricted the AutoMerge function to a part of the model. There should only be conflicts in the part of the model you have reverted and automerged (if there are any at all). Complete your merge of this part of the model and save the model normally.

Note: *The Check Merge function may find errors due to references to the parts of the model you have excluded from the merge with this procedure. If this happens, you must resolve the reference errors even if 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, do the following in the **File > Contributors** dialog:

1. Before clicking the **Compare** or **Merge** button to load the models, uncheck the Compare/Merge Against Base Model check button in the Contributors dialog box.
2. Load the models normally.

Model Integrator will automatically create a base model that is empty. The base model will occupy the slot for Contributor 1, but it will not normally be 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 will let you specify a merge session with as few as two files when Compare/Merge Against Base Model is not checked.*

When merging models using this feature, all nodes in the contributors that do not conflict with each other will appear with the **+** indicating that they are being added to the merged model. Conflicting nodes will behave normally.

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. Select **File > Contributors**.
2. Select a single file in the Contributors dialog, and then click the **View** button.

Note: *If two files are displayed, or if you have begun to enter a second filename, the button text will change from View to Compare. When the button displays Compare but you have only entered a single file name, clicking the Compare button will still enter view mode.*

Using Model Integrator from the Command Line

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

Command	**Description**
modelint file.mdl	Starts Model Integrator with file.mdl in the View Mode.

Command	Description
modelint file1.mdl file2.mdl	Starts Model Integrator in Compare mode for the two 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 the following command line options can be used. You can 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.
/compare	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: 0 for identical models or 1 for models with differences.

Command	**Description**
/merge	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.
/out *filename*	Specifies the name of the file to write the merged output file to. You must specify an absolute or relative pathname for the file. Either of the following are valid: /out c:\models\test.mdl /out .\test.mdl but this is not valid /out test.mdl
/ask /all /none	Subunit policy options: 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. The /all option will load/save all the subunits without prompting the user with subunit dialogs. The /none option will suppress the loading/saving of subunits.

Chapter 4

Working with a Version Control System

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 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.
- Ability to define configurations of related artifacts then checkpointing and retrieving them at any time.

Version control systems are intended to 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 he or she 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, RCS (Revision Control System), and SCCS (Source Code Control System).

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 don't 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 that will be 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 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 adding a new feature. When the changes for an activity are submitted to the repository, the model will evolve to a consistent new state.

Integration

Integration is the process of making changes available for use by other developers. Sometimes integration will always be performed by a specific person, but it is also normal for developers to play this role as well.

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 that is dated before midnight, May 12
- The version labeled "Build 6.1.112" of each file in the project

Lineups are used to 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 that other developers are doing. 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 the rest of the team until it is ready for integration.

To support these basic team development requirements, each developer should 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 would be 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, a "latest is greatest" approach is usually taken. This means that 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 changes have been made, they are submitted. This immediately makes the new version visible to other users, and will become 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 to doing bug fixes on an existing release while doing new development for a future release.

Both file and view based version control systems can be used for small projects without the need for branching or multiple stream development.

Benefits:

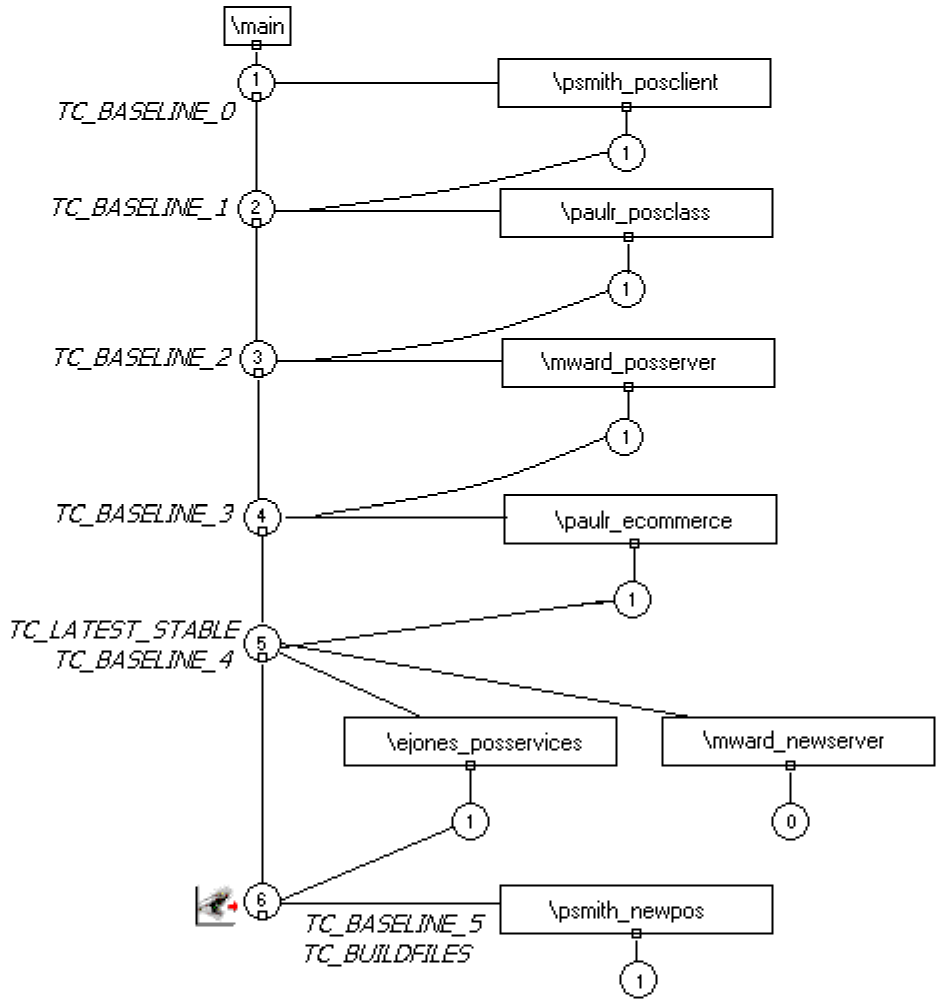
- Simple to set up.
- Work area configurations do not need to be modified.
- Can browse any lineup stored in the repository.

Drawbacks:

- Work is always based on latest version of elements in version control system.
- 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 may arise.

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. The following figure shows the version tree for a typical file in a parallel development project:



Most parallel development environments involve nominating a branch in the version control system as the integration branch. The integration branch is used for collecting all changes to the project (/main is the integration branch in the above diagram). Testing, release builds, and new development are all based on the contents of the integration branch.

All labeled lineups should consist of file versions from the integration branch. Once established, a labeled lineup can serve as a the basis for builds, testing, or further development. Frequently, a temporary lineup is established and built. If the build completes successfully and passes basic sanity tests, the lineup is then made available as a baseline. This process is usually automated, and should be done on a nightly/weekly basis. In the version tree above, the TC_BASELINE_n labels indicate stable baselines on the integration branch.

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:

- Controlled baselines
- Baselines allow reuse of build results
- Provides better control over exposing changes to development team

Drawbacks:

- Requires more sophisticated version control system knowledge
- Separate integration step
- Work area configurations must be updated regularly

Using Rose's 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 SCC[1] compliant version control system.

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

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 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**).

The Version Control Add-In automatically determines which version control system is installed. To see which version control system and SCC API version the Version Control Add-In is using, see the Version Control Options dialog.

Note: *For the Version Control Add-In to work with your version control system, the version control system has to be correctly set up, see Setting Up ClearCase to Work with Rose or Setting Up Microsoft Visual SourceSafe to Work with Rose.*

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

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 correctly set up. See Setting Up ClearCase to Work with Rose.*

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 this detection, either the Version Control Add-in or the ClearCase Add-In is activated by default on your system.

If you install or change your version control system after you have installed Rose, you must ensure that the appropriate add-in is activated. In addition, make sure that only one of the version control add-in is active at a time. Because the Version Control and ClearCase add-ins use many of the same commands, you may get 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 are using Rational ClearCase for version control, we recommend that you activate the ClearCase Add-In, even though in the Windows NT environment, you can also use the Version Control Add-In. The ClearCase Add-In provides a much tighter integration and gives you 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.

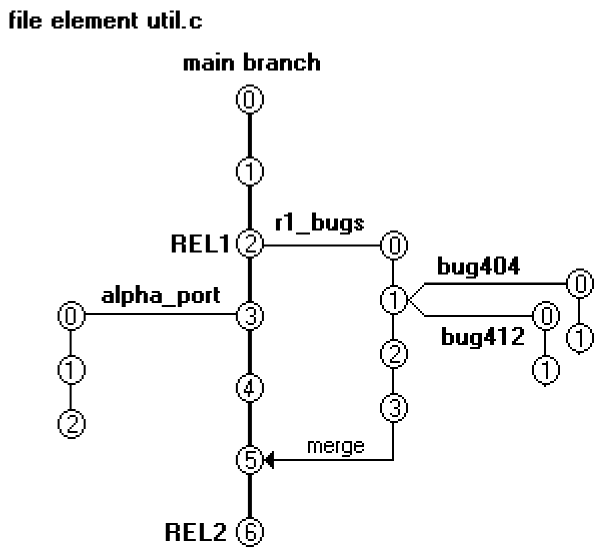
In addition, the repository stores detailed accounting data on the development process itself, such as who created a particular version, what versions of source went into a particular build, and other relevant information.

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:



As this figure shows, branches have user-defined names, typically chosen to indicate their role in the development process. All versions have integer ID numbers. Important versions can be assigned version labels 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.

Setting ClearCase Up to Work with 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.) Rose then sees the files in the current view as if they were stored on a regular (non-ClearCase) file system.

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. Thus, in order for the files to be added to version control the model must be saved to a view directory that is not view-private.

ClearCase lets you define a new element type, including specifying the merge and differencing tool that should be used on files of the new type. Rose uses this feature to define an element type that applies to all Rose files placed under version control. With this element type defined, all new Rose files that are placed into a VOB are associated with the file type and will use Model Integrator as their default merge and differencing tool. (For more information about Model Integrator, see Chapter 3.)

Steps for Setup

Follow these steps to set up your ClearCase environment to work with Rose:

1. Create and mount a ClearCase VOB (for example, ProjectRose). Note that both VOBs and views must be created directly in ClearCase; that is, outside of Rose.
2. Create a ClearCase view to provide access to the VOB you created. If you are using ClearCase on Windows, you will need to map the view to an appropriate Windows NT drive (for example, z:\)
3. Create all necessary views and prepare the model for team development and organize the model in 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 Rose in the Windows NT 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 will be.
   - Create the element type in this VOB, by typing:

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

   If you are using Rose in a Unix environment:

   In 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) works by storing and retrieving files on your local disk. Each VSS project has a working folder specified for it. Rose saves model elements to and loads elements from this working folder. VSS then checks those local files into and out of its repository.

Setting Up Microsoft Visual SourceSafe to Work with Rose

Steps for Setup

1. Make sure there is a Microsoft Visual SourceSafe database available to store the model. The database must be created directly in SourceSafe.
2. Since the Version Control Add-In uses the current user name to identify the SourceSafe user, the system administrator must enter you as a user before you can use the integration. (To find out what is the current user name, click **Version Control** on the **Tools** menu in Rose, and then click **Version Control Options**.)
3. In Microsoft Visual SourceSafe, open the database where the model is going to be stored and create a project for the model. (You can start the application from within Rose by clicking **Start Version Control Explorer** on the **Tools** menu's **Version Control** menu.)
4. Right-click on 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 SourceSafe project, but you specify different working folders for the files (for example, c:\NewPackage.cat and c:\temp\NewPackage.cat), SourceSafe controls the first file in the project (c:\NewPackage.cat), but it will not control the second file (c:\temp\NewPackage.cat). No error message will inform you that the second file was not controlled. For this reason, it is highly recommended 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. Choose **Options** from the **Tools** menu. On the **Command Line Options** tab, select the **Assume Project Based on Working Folder** check box.

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

Using Version Control Features From Rose

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

If a model has already been put 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 do this, the Version Control Add-In will be unaware of the previously controlled units.

Note: *This information applies only to the Version Control Add-In. If you are using the ClearCase Add-In, you do not have to do anything else. The ClearCase Add-In can work with previously controlled units.*

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

1. Make sure that the model is not opened in 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 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 the **Keep Checked Out** option.
5. If you are using Microsoft Visual SourceSafe: Click the **Browse** button and search for the project, then click **OK**. If all the files are located in the same project, click **Select All**, then click **OK**. Otherwise, click the check box next to each file that is located in the selected project, click **OK**, and then repeat steps 3-5 for each set of files that are 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, but from now on 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 SourceSafe. The same procedure is used to control the model file, the deployment diagram, or the model properties.

1. Make sure that the unit to which the package belongs—that is, the model file or the enclosing package—is checked out.
2. Right-click on the package in either the browser or diagram, then click **Add to Version Control** on the shortcut menu.
3. A list with all model elements that can be added to version control is shown. The selected packages are selected by default in the list. Make sure that all packages that you want to add to version control are selected in the list.
4. If you are using Microsoft Visual SourceSafe, make sure 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 then select the appropriate project.

   Note: *It is not recommended that you create new projects in the dialog box that is displayed when clicking the Browse button. However, if you do, the working folder for the new project becomes the same as the folder where you save the controlled units that you are adding to version control.*

5. If you want to keep the files checked in after this operation, clear the **Keep Checked Out** check box.
6. Optionally, write a comment in the **Comment** box. Your version control system inserts the comment as a description of the new unit.
7. Optionally, click the **Advanced** button to display a dialog box with additional options. Note that this button is not available for all version control systems.
8. Click **OK**.

9. For each selected unit that has not been saved to file yet, a **Save As** dialog box is displayed. In the displayed dialog box, 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 SourceSafe or in the appropriate ClearCase view.
10. Click **Save**.

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

Note: *If you are using Microsoft Visual SourceSafe, the Add to Version Control command can only handle files that are located in the same SourceSafe project and working folder. Thus, 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 diagram and click **Check In** on the shortcut menu. A list with all loaded, checked-out controlled units in the current model is displayed. Any units that are 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 checkin in the **Comment** box. The text you type will be stored by your version control system as history for the current check-in.
4. Optionally, click the **Advanced** button to display a dialog box with additional options. For example, in order to check in an unchanged unit in ClearCase, you must select the Check in even if identical option in the **Advanced** dialog box. If you do not select that option when checking in an unchanged unit, you will get an error message. (Note that the **Advanced** button is not available for all version control systems.)
5. Click **OK**. Rose checks in the units and makes the corresponding model elements read-only in the model.

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

If you are using the Version Control Add-In with Microsoft Visual SourceSafe, the Check In command can only handle files that are located in the same SourceSafe project and working folder. Thus, to check in files that belong to different projects, you must repeat the Check In 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 diagram and click **Check Out** on the shortcut menu. A list with all loaded, checked in controlled units in the current model is displayed. Any units that are currently selected in the browser or 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 will be stored by your version control system as history for the current checkout.
4. Optionally, click the **Advanced** button to display a dialog box with additional options. For example, if you are using ClearCase, you can make an unreserved check-out with the advanced options. (Note that the **Advanced** button is not available for all version control systems.)
5. Click **OK**. Rose checks out the files and makes the contained model elements editable.
6. If Rational Rose asks whether you want to load the unit, click **Yes**.

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

To be able to check out a controlled unit, the unit must previously have been added to version control from within Rose by 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 is displayed. Any packages that are currently selected in the active diagram are selected by default in the list.
2. Select the checked-out unit.
3. Optionally, click the **Advanced** button to display a dialog box with additional options. Note that this button is not available for all version control systems.
4. Click **OK**.
5. In case Rose asks whether you want to save the changes before loading a new unit, click **No**.

Note: *If you are using Microsoft Visual SourceSafe, the Undo Check Out command can only handle files that are located in the same SourceSafe project and working folder. Thus, 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 in order 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 is displayed. Any packages that are currently selected in the active diagram are selected by default in the list.
2. Select the appropriate unit.
3. Optionally, click the **Advanced** button to display a dialog box with additional options. Note that this button is not available for all version control systems.

4. Click **Get**.
5. In case Rose asks whether you want to save the changes before loading a new unit, click **No**.

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

If you are using 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. Make sure 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 with all loaded controlled units that have been put under version control by the Version Control Add-In is shown. Any packages that are 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 the **Advanced** button to display a dialog box with additional options. Note that this button is not available for all version control systems.
5. Click **OK**. The selected unit is removed from version control and its contents is 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 are using Microsoft Visual SourceSafe, the Remove From Version Control command can only handle files that are located in the same SourceSafe project and working folder. Thus, to remove files that belong to different projects, you must repeat the command for each project.*

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

Working with Non-SCC Version Control Systems

You can enable Rose to work with SCCS and RCS systems via scripts that you write. Check Rational's website for existing scripts that you can modify.

Note that neither RCS nor SCCS directly support directory hierarchies. To support a hierarchical repository, you will need to create a separate RCS/SCCS storage directory for each level in the model hierarchy

For example, the repository structure might look something like the following, where <dir> indicates a directory:

```
<repository>
  <models>
     <RCS>
       MyModel.mdl,v
  <MyModel>
    <RCS>
        LogicalView.cat,v
        ComponentView.sub,v
        UseCaseView.cat,v
        DeploymentView.prc,v
    <LogicalView>
        <RCS>
          ...
         <ComponentView>
            <RCS>
                ...
```

Repository Mapping Files (.rmf)

Each developer in a team will use their own local working directory for working on models. A special mapping file is then required to map the local working directory to the repository directory representing the root of the hierarchy. This map file is referred to as a Repository Mapping File (RMF). Each line in the RMF is a file name prefix mapping that works similarly to the virtual path map mechanism within Rose. Each entry consists of two path prefixes, separated by an equals sign (=)

Example:

```
/home/john_doe/Rose/models=/repository/models
```

By applying this map file, the Rose RCS integration will map local working directory

```
/home/john_doe/Rose/models
```

to repository directory

```
/repository/models/RCS
```

The RMF may contain multiple entries. The first valid prefix will be used, and successive substitutions will not be applied

Before determining if an RMF source prefix is valid for a given path, both the source and destination prefixes will have environment variable substitution performed on them. Thus, assuming every user had a Rose/models directory in a home directory, the following RMF file could be used by all users working from the given repository:

```
/home/$user/Rose/models=/repository/models
```

Note: *The RMF must not contain softlinks to directories. It must contain the actual path to the directory*

Version Control Operation Behavior with SCCS

SCCS does not support labeling. All labeling operations will be unavailable from Rose.

RCS/SCCS Repository Setup

You must create the repository root directory. Be sure to place appropriate access permissions on the directory so that users will have the required access to the files.

If you will be using a global RMF for all users accessing the repository, you should create it and place it in a location accessible to all users

RCS/SCCS Workstation Setup

Command Line Access to the Source Control Tool

The RCS/SCCS executables must be available from your path in order for Rose to integrate with them.

Create an RMF File

Use a text editor to create the RMF file that will contain the mapping between your local working directory and the RCS/SCCS repository. Create an entry in your RMF to point to the working directory set aside for your models (create a working directory if you don't already have one).

Set RMF Environment Variable

Your RCS/SCCS script can examine an environment variable to determine which RMF to use.

For RCS:

- Set the environment variable to the name of the file containing the map entry. Example:

```
setenv ROSE_RCS_MAPFILE ~/MyRCSMap.txt
```

For SCCS:

- Set the environment variable to the name of the file containing the map entry. Example:

```
setenv ROSE_SCCS_MAPFILE ~/MySCCSMap.txt
```

Chapter 5

Establishing a Model Architecture and Process for Team Development

About Model Architecture and Process

The previous chapters 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 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 that 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

The process described in this chapter rely on these roles:

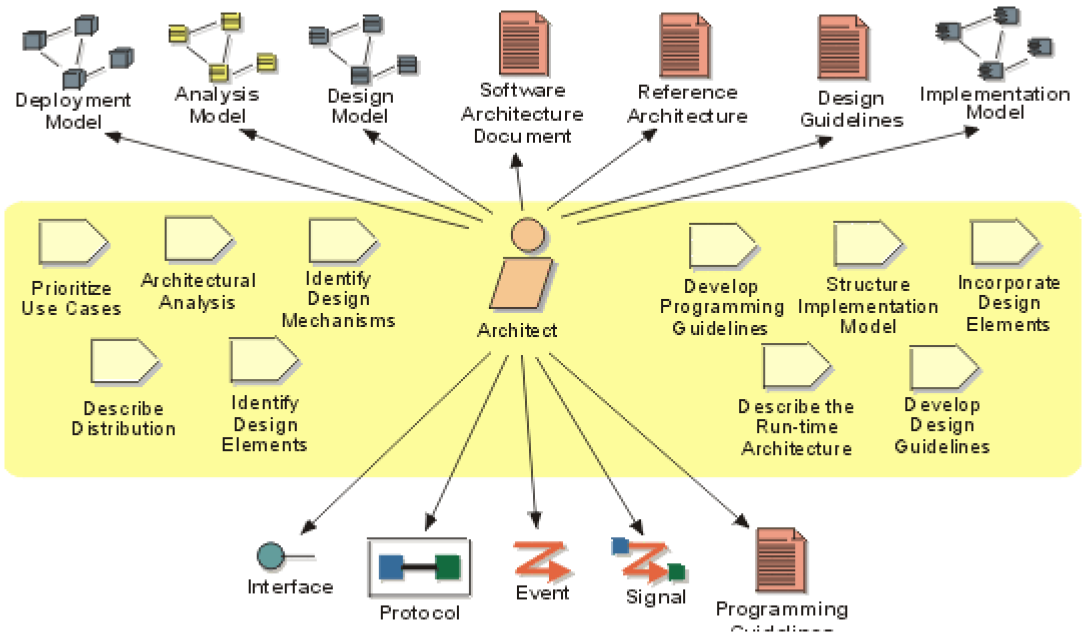
- Model Architect
- Model Manager
- Modeler/Developer

- Integrator

Depending on the scale of your project and your staffing, some roles may be carried out by one person or by a team.

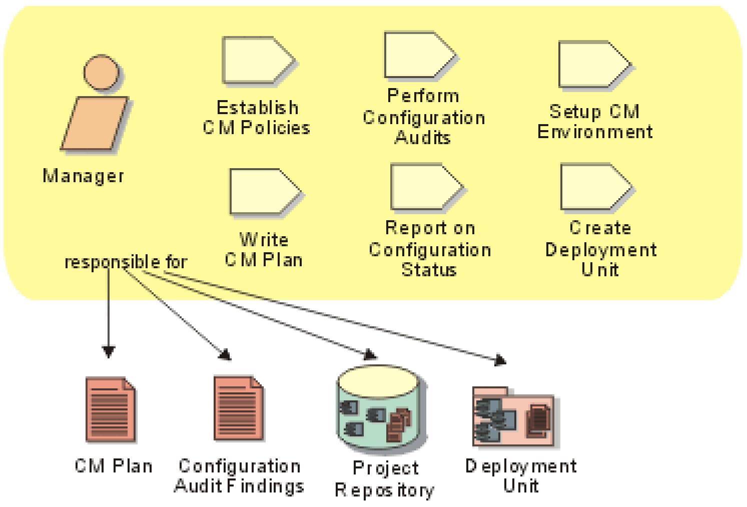
Model Architect

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.



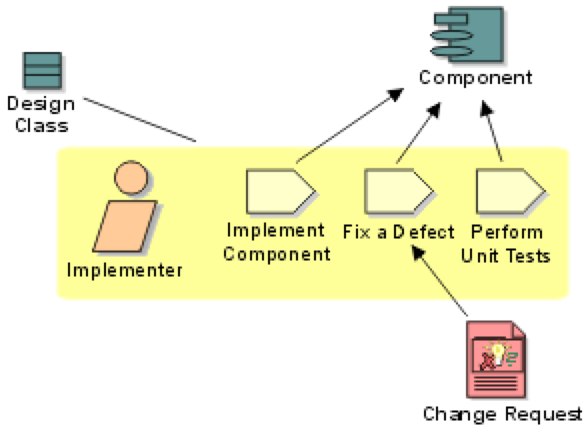
Model Manager

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.



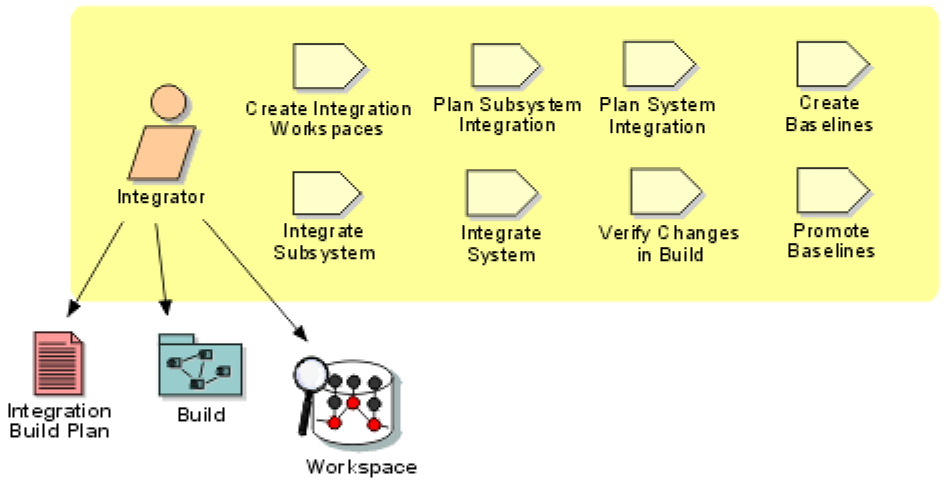
Modeler/Developer

This is the collective name for those workers who view or modify Rose models.



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.



Developing a Model Architecture

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

Product development often starts with a small team working on one 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 parts of 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 that will help you get started. Remember that once a model is well partitioned into subsystems, you can either work with one model or split the model into separate models for each subsystem.

Understanding Subsystems

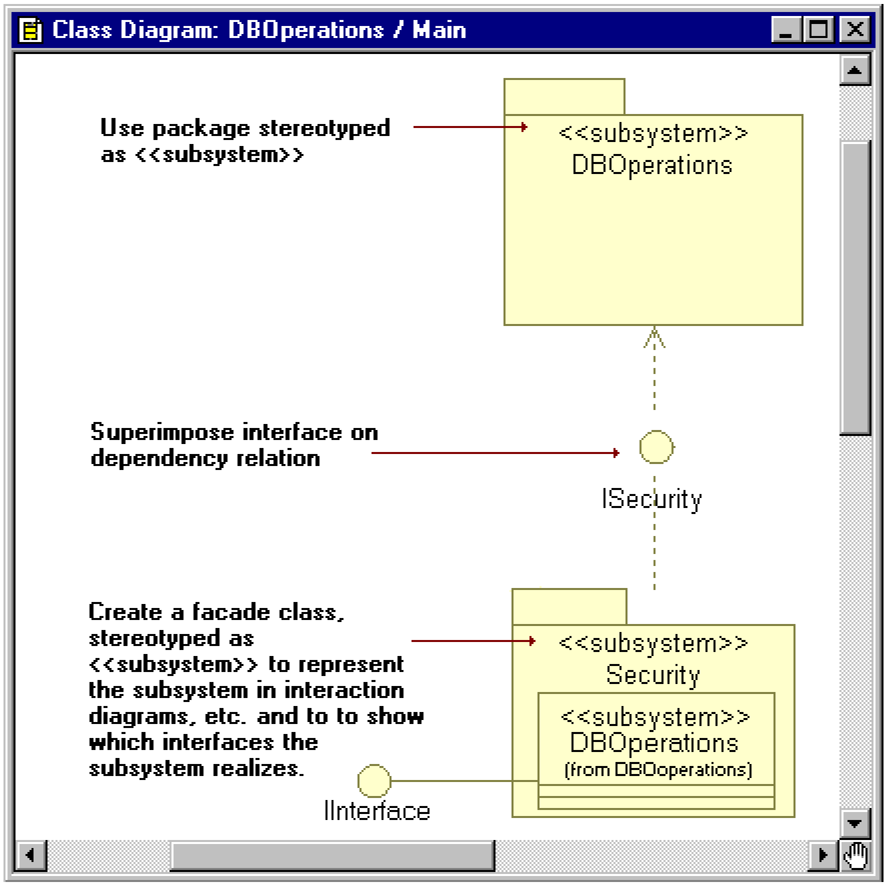
Packages are used to group model elements. There are four kinds of packages in Rose: use case, logical, component, and deployment packages. Each kind of package can only group certain model elements. For example a logical package can group classes, whereas a component package groups component diagrams 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 Rose. The term is used to represent a set of related packages that can be developed, tested, and released together. Subsystems provide strong separation between major portions of your model and they form the basis for reuse between models. In a layered development approach, the model for each layer will share in the 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 will be better able to control dependencies. Subsystems expose services only through an interface and subsystem internals should depend only on the interfaces that are offered by other systems.


Class Diagram: DBOperations / Main
Use package stereotyped as <<subsystem>>
<<subsystem>> DBOperations
Superimpose interface on dependency relation
ISecurity
Create a facade class, stereotyped as <<subsystem>> to represent the subsystem in interaction diagrams, etc. and to to show which interfaces the subsystem realizes.
<<subsystem>> Security
<<subsystem>> DBOperations (from DBOoperations)
IInterface

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 Rose performance and memory footprint simply 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
- Setting up 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 sometimes define class-level relationships that violate dependencies between packages and subsystems. Once you have created packages and moved the model elements into the packages (subsystems), you will want to ensure that the subsystems you have created have the dependencies that you expect. If the dependencies between subsystems are too complex, it will be difficult to work in teams (changes won't be isolated) and split the model.

Show Access Violations

Use **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.

Use 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 for a package contains a **Relations** tab which shows the dependencies for this package. This is a quick way to see if a package has any dependencies, but it can be difficult to visualize the dependencies if you just look at this list. In order to properly visualize the package relationships, use a class diagram.

To quickly create a class diagram showing the relationships for a specific package, try the following:

1. Open the class diagram.

2. If this package is not already on this diagram, then drag it from the browser onto the diagram.
3. Use **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 chosen in the Relations dialog.

By varying the options chosen in these dialogs you can quickly produce a diagram showing the desired information. If many packages were added to the diagram, then you may wish 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*), 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. For ease of use, it is also recommended that the subsystem contain a set of class diagrams that illustrate the public classes.

Best practices

1. Specify the visibility of each class (public or implementation).
2. 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.

Setting Up Subsystem Components

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 will have an executable component and many library components, typically corresponding to the layering in the architecture.

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

Components in Subsystems

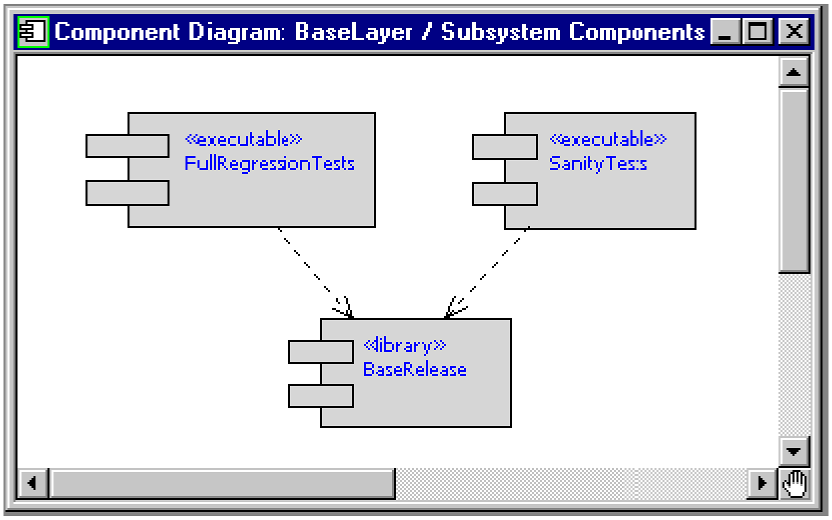
Ideally each subsystem will contain one or more external library components. These components are built as part of the build process of a subsystem and are referenced in models that use the subsystem.

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

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

The subsystem model will need one or more executable components that are used to test the subsystem. Typically, the executable component will only contain the testing classes and it will have 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.



Once you have created the necessary components and the dependencies between them, you will have to determine which classes belong to which components. Typically, this will follow naturally from the architecture of the model although there can be issues that arise during development. As new classes are created, they will need to be added to the appropriate component(s). If multiple developers are creating classes that are 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. Remember that 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 end up containing 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 are creating 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 that are 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:

- Builder or architect defines custom sets of component properties which 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 deployment package that is shared between the various subsystem models.

The builder should set up a deployment 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 deployment 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 deployment 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 would include 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 are 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 deployment 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 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:

1. Label Subsystem Elements. If the model is under version control, then a label can be applied to the elements in the subsystem.

2. 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 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, it is recommended 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 will allow each subsystem to exist in a model that does not contain any of the 'higher level' subsystems. See *Checking Package Dependencies for Completeness*.

Should you split the model before adding to version control?

If your model is not already in version control then it is best to split the model before adding it to version control. If your model is already in version control, then it is still possible to split it into separate models but the process is a bit different.

Splitting a Model not in Version Control

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

Lastly, this description also assumes that you will want to keep the controlled units for each subsystem model together so they will 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 Chapter 2 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.

   The base model (Base) directory hierarchy for the sample model would look something like:

   ```
   Base.mdl
   <Base>
     UseCaseView.cat
      <UseCaseView>
    LogicalView.cat
    <LogicalView>
       SubSystem1.cat
   ```

```
    <SubSystem1> SubSystem2.cat
    <SubSystem2>
 ComponentView.sub
<ComponentView>
    SubSystem1.sub
    <SubSystem1>
    SubSystem2.sub
    <SubSystem2>
DeploymentView.prc
 <DeploymentView>
```

2. Use **File > Edit Path Map** to create a Virtual Path Map variable for each top level package in the model (i.e., each subsystem package). In our example, we could create path map variables SubSystem1LogicalPkg, SubSystem1ComponentPkg, SubSystem2LogicalPackage, SubSystem2ComponentPkg, etc.
3. Save the Base model units that will be affected by the new path map variable.
4. If the Base model makes use of custom property sets, then these must be made available to the subsystem models. Use **Tools > Model Properties > Export** to create a file that can be imported to the subsystem models.
5. Create a new model by selecting **File > New**. This model will be used for the first subsystem. Ensure that the path map variables are still defined correctly.
6. If the Base model makes use of custom property sets, then ensure that these are available in the subsystem model. Use **Tools > Model Properties > Replace** to import the file containing the property sets.
7. Control all the elements in the new model by using **File > Units > Control**.
8. Save the model (.mdl) into an appropriate directory using **File > Save As**. We suggest that you create a dedicated directory for each subsystem. For example, we could name the subsystem model SubSystem1 and store it in a directory called SubSystem1.
9. 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 should look something like:

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

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

10. Next you will have to add the subsystem packages into the subsystem model using **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.
11. Save the subsystem model.

Steps 5 - 11 should be repeated for each remaining subsystem with the following addition.

Before adding the subsystem packages to the new subsystem model (step 8), 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, 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.

Splitting a Model Under Version Control

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

Lastly, this description also assumes that you will 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 much manage the files that make up each model.

See Chapter 2 for background information that should be understood before proceeding with this task.

Tasks

1. 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 would look something like:

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

2. Use **File > Edit Path Map** to create a Virtual Path Map for each top level package in the model (i.e., each subsystem package). In our example, we could create path map variables SubSystem1LogicalPkg, SubSystem1ComponentPkg, SubSystem2LogicalPackage, SubSystem2ComponentPkg, etc.
3. Check out the root packages in the Base model.
4. Explicitly save the Base model units that will be 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. Use **Tools > Model Properties > Export** to create a file that can be imported to the subsystem models.
7. Create a new model by using **File > New**. This model will be used for the first subsystem. Enable version control for this model by using **Version Control** from the Tools menu. Ensure that the path map variables are still defined correctly.
8. If the Base model makes use of custom property sets, then ensure that these are available in the subsystem model. Use **Tools > Model Properties > Replace** to import the file containing the property sets.
9. Control all the elements in the new model by right-clicking on the Model in the browser and choosing **File > Units > Control**.
10. Save the model in the appropriate local working directory for your version control system using **File > Save As** (e.g. /vob/SubSystem1). We suggest that you create a dedicated directory for each subsystem.

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

    If you choose, you may add the subsystem model to version control at this stage. Use **Tools > Version Control > Add to Version Control** to ensure that all the controllable units are added.
11. 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.

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

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 something like:

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

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

12. Next you will have to add the subsystem packages into the subsystem model using **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, then you will be prompted to check out the root packages that are affected.

13. Save the subsystem model.
14. Now, enter the changes for this subsystem model into version control.
15. It is recommended 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 Rose manager/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

Once these steps are completed, development can start. However, you should consider these additional responsibilities:

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

Setting Up Compatible Workspaces

In order to 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 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, etc.
- **Defining the physical storage structure for model elements**. In this task, you determine how the various model elements (specifically the controlled units) will be 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 Chapter 2 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.

Setting Up Version Control System and Repository

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

Before continuing, please review the tool-specific documentation in Chapter 4.

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

Partitioning the Model into Controlled Units

Controlled units are the smallest 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 Chapter 2.

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, simply 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 simplest way to add all applicable units to version control is to use **Tools > Version Control**.

Defining Developer Work Areas

At this point, the model manager should think about how each worker (developer, integrator, etc.) 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 Chapter 4 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.

For an example of an effective labeling and lineup strategy, see Chapter 6, *Parallel Development Sample Using ClearCase*.

Manipulating the Version Control Repository

From time to time 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 will be able to carry out the task most effectively.

Developing/Implementing a Model

Developers will be working day-to-day with a subsystem model under version control. Therefore, each developer should be familiar with the material in Chapter 4.

Setting up Version Control

Before using Rational Rose with your version control system, you must perform any tool-specific configuration as described in Section 4.

If you have customized 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’s a matter of setting up a local work area before running Rose.

Getting a Specific Lineup of a Model

When a developer begins a development task, he or she 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. Similar labeling strategies can be used with RCS/SCCS.

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 associated workspace file is the recommended way to load the model into 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

Once your model has been placed under version control, you will 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. Once you have a model under version control, you should check out elements before you edit them. Depending on the version control settings, Rose may force you to check out before editing.
- Undoing a Check Out. After you have checked out a controlled unit, you may choose to undo the check out and not submit a new version.

Comparing and Merging Model Elements

See Chapter 3 for details.

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. For an example of integration with a parallel stream development process, see Chapter 6, *Parallel Development Sample Using ClearCase*.

Building and Integrating

The Integrator combines the changes from multiple developers to produce a lineup that can be used as a basis for the next set of development activities. The Integrator is typically responsible for the automated building process.

Among the tasks involved in building and integrating are:

- Building using Automated Scripts
- Building within a Larger Build Procedure
- Reusing Build Artifacts
- Integrating Changes
- Automating Model Validation

Building using Automated Scripts

Starting with a valid model, it is possible to initiate a build from a clean directory using the following two steps.These are effectively the same steps used by Rose.

Note: The '\' character in the following command syntax represents the command line continuation character. This may be different on your system.

1. Build the makefiles:

```
${CodeGenMakeCommand} ${CodeGenMakeArguments} \
 -f $ROSE_HOME/codegen/bootstrap/${CodeGenMakeType}.mk \
 "HOME=${TargetServicesLibrary}" \
 "MODEL=${ModelFile}" "COMPONENT=${QualifiedName}" \
 Rmakefiles
```

where *CodeGenMakeCommand*, *CodeGenMakeArguments*, *CodeGenMakeType*, and *TargetServicesLibrary* are replaced by the corresponding value in the component; QualifiedName is replaced by the fully qualified name for the component; and ModelFile is replaced by the file name for the model (.mdl) file.

2. Generate the code and compile using:

```
${CodeGenMakeCommand} ${CodeGenMakeArguments} \
 -f Makefile Rcompile
```

For example, if the following substitutions are made:

Argument	**Sample Value**
${CodeGenMakeCommand}	clearmake
${CodeGenMakeArguments}	-k -J4
${CodeGenMakeType}	ClearCase_clearmake
${TargetServicesLibrary}	$ROSE_HOME/C++/TargetRTS
${ModelFIle}	/my/path/MyModel.mdl
${QualifiedName}	Component View::MyComponent

The resulting commands are:

```
clearmake -k -J4 \ -f
$ROSE_HOME/codegen/bootstrap/ClearCase_clearmake.mk \
"RTS_HOME=$ROSE_HOME/C++/TargetRTS" \
"MODEL=/my/path/MyModel.mdl" \
"COMPONENT=Component View::MyComponent" \
Rmakefiles clearmake -k -J4 -f Makefile Rcompile
```

Note that automated builds are not restricted to clearmake.

Building within a Larger Build Procedure

For integration into a larger build procedure, automated builds can generate the code and compile the code in two separate steps. This involves a slight change to the steps listed above:

1. Build the Makefiles using the same command as above.

2. Generate the code (without compiling it) by replacing "Rcompile" above with "Rgenerate":

```
$CodeGenMakeCommand} ${CodeGenMakeArguments} \
 -f Makefile Rgenerate
```

3. Compilation of the generated code (without regenerating it) uses "Rmycompile":

```
${CodeGenMakeCommand} ${CodeGenMakeArguments} \
 -f Makefile Rmycompile
```

Note: *The '\' character in the command syntax represents the command line continuation character. This may be different on your system.*

If we use the same example substitutions as above, then the resulting commands are:

```
clearmake -k -J4 \
 -f $ROSE_HOME/codegen/bootstrap/ClearCase_clearmake.mk \
 "RTS_HOME=$ROSE_HOME/C++/TargetRTS" \
 "MODEL=/my/path/MyModel.mdl" \
 "COMPONENT=Component View::MyComponent" \
 Rmakefiles
```

```
clearmake -k -J4 -f Makefile Rgenerate
```

```
clearmake -k -J4 -f Makefile Rtmycompile
```

Reusing Build Artifacts

A Rational ClearCase environment supports build artifact reuse by using the ClearCase "wink-in" feature. Both "clearmake" (Unix and Windows NT) and "omake" (NT only) provide the wink-in mechanism.

Creating Reusable Build Artifacts

In order for build artifacts to be able to be winked in, the following criteria must be met:

- The component's OutputDirectory must be in a view.
- All controlled units within the model must be version controlled in a ClearCase VOB.
- All controlled units must not be checked out to the view performing the build.

- The build must be performed from a clean directory. If a build is unsuccessful, the OutputDirectory must be completely cleaned in order to guarantee wink-in.
- In the component, the CodeGenMakeType and CompilationMakeType properties must both be set to either "ClearCase_clearmake" or "ClearCase_omake" as appropriate. Similarly, the CodeGenMakeCommand and CompilationMakeCommand properties must be set to something appropriate, typically either "clearmake" or "omake".

The OutputDirectory can be a view-private directory, but that requires that every developer create that directory in their view first. A recommended practice is to use a directory element that is stored in a VOB.

The following are encouraged practices:

- All external include files should be version-controlled in a ClearCase VOB.
- The TargetServicesLibrary should be version-controlled in a ClearCase VOB.
- Other linked libraries should be version-controlled in a ClearCase VOB.
- Optionally, $ROSE_HOME should be version-controlled in a ClearCase VOB.

Using Build Artifacts

A developer wishing to reuse the artifacts from a build should:

- Assign his or her environment variables (such as $ROSE_HOME and $PATH) appropriately,
- Use the same versions of elements that the build used,
- Create in his or her view, if it does not already exist, the same OutputDirectory used by the builder
- Perform the same activity that the builder performed (a compile or a generate, from within the toolset or from the command-line).

See Chapter 6, *Parallel Development Sample Using ClearCase*, for a description of a development process that provides significant build artifact reuse.

Integrating Changes

Integrating developer 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 that can be used as a basis for subsequent development activities. This will often involve merging changes from multiple developers (using the Model Integrator) and performing local builds to verify sanity. For an example of how integration can be performed in a parallel development environment with ClearCase, see Chapter 6, *Parallel Development Sample Using ClearCase*.

Automating Model Validation

Rose provides an automated way 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 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 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 (e.g., 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.

Chapter 6

Parallel Development Sample Using ClearCase

This chapter details how a parallel development process can be set up to use Rational Rose with Rational ClearCase. The process presented here is an example meant to explain parallel development and is not in any way a definitive guide for working with ClearCase. Feel free to use this process as is, or to modify and customize it as necessary to fit your project's needs.

Many of the techniques presented in this example are not specific to either ClearCase or parallel development, although the details certainly are.

Note: *Throughout this example, the prefix TC is used to indicate an identifier that is unique to the project being worked on. Using distinct labels for each project will help keep development progress self-contained and more manageable.*

Overview

The benefits of a proper parallel development process are:

- Reduced contention for checkouts
- Private version streams for development activities
- Shared build results to reduce incremental development times
- Stable and controlled evolution of the system being developed

As explained in *Parallel stream versioning* in Chapter 4, the integration branch plays a central role in most parallel development strategies. In this example, "/main" is used as the integration branch. All automated builds are generated from the integration branch, all lineups are created from elements on the integration branch, and all development is based off of the integration branch.

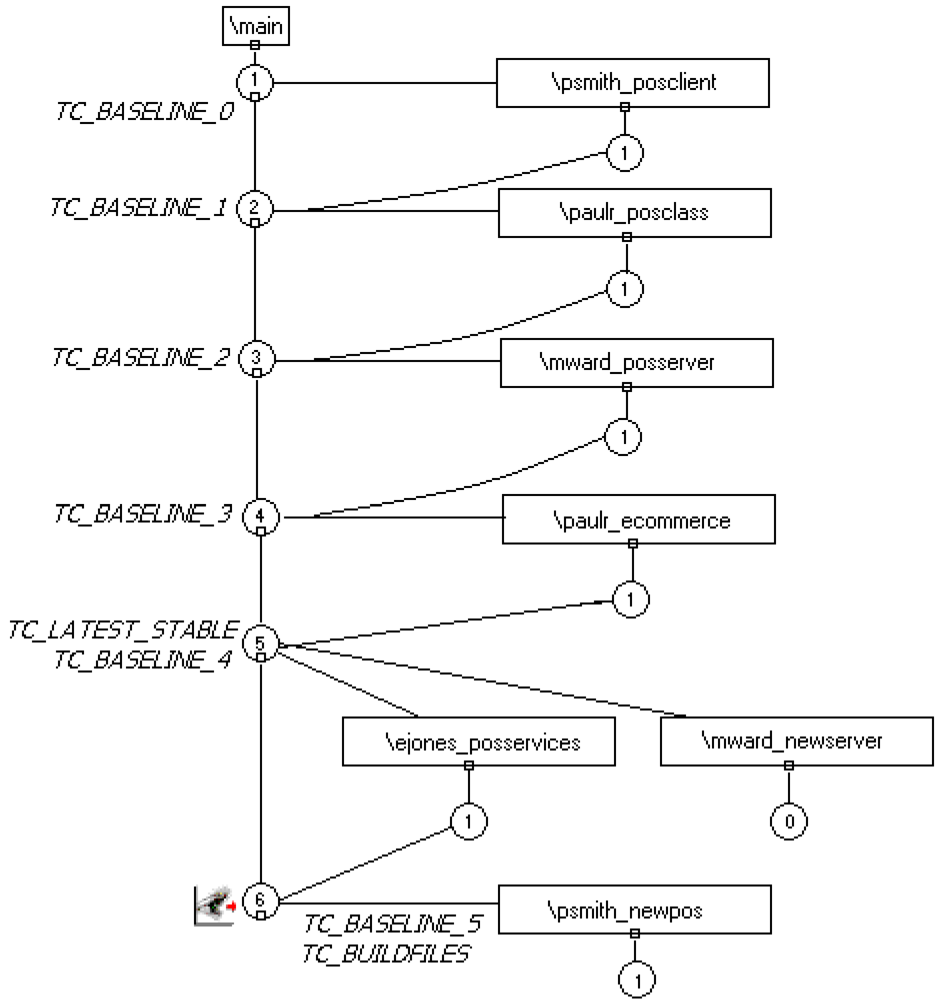
Automated builds are performed on the contents of the integration branch. To ensure reproducible builds (and provide wink-in of build artifacts), the latest version of each file and directory on the integration branch is labeled with an identifier such as TC_BUILDFILES. Using a label instead of a timestamp or whatever happens to be in view insures that a build is completely reproducible. If the version of a file labeled with TC_BUILDFILES causes compile problems, then a previous version of the file can be used simply by applying TC_BUILDFILES to the appropriate version and re-building incrementally.

When the build is successful, a new label is generated of the form TC_BASELINE_NNN. The label is then applied to the exact version of each file that was included in the build (i.e., every version that was labeled with TC_BUILDFILES is now labeled with TC_BUILD_NNN).

As far as development is concerned, no actual development occurs on the integration branch. All development is carried out on private branches, one per development activity. Each private branch is based off of a lineup on the integration branch, conveniently labeled by the automated build process. Since the file versions used in the build are also used by developers, wink-in of build artifacts comes for free.

Once a development activity is finished, an integrator is given the branch name and will then merge the changes for that activity onto the integration branch when time permits.

The following diagram illustrates a typical version tree for an element in this process:



Using View Templates

To ensure that developers use a common base for their view's config spec, and to make it easier to work on private branches, view templates are used. A view template specifies the integration branch to work from, lists labeled checkpoints that can be used to base a private branch on, and includes a config spec template that can be filled in with additional config spec rules.

- **Windows NT**. This functionality is provided with ClearCase 3.2.1 for NT through View Profiles.
- **Unix**. ClearCase for Unix does not include support for View Profiles.

Every developer will need access to a common location from which the view templates will be accessed. The view template scripts look for the view templates in the directory named by the CCVIEWTEMPLATES environment variable.

Each view template consists of the following parts:

- A list of labels that indicate integration branch lineups
- A config spec for browsing any specific integration branch lineup
- A config spec for performing a development activity on a private branch
- A config spec that will be used by the integrator
- A config spec that will be used by the builder

Since the config specs for each project will be different, a view template must be generated for each project.

See *View Template Script Usage* for complete details on how to use the view template scripts.

ClearCase Entities

This development process will require the creation and usage of the following ClearCase entities.

Views

A separate view will be needed for the integrator, for the builder, and for each developer.

View Template

A view template will be needed to provide a standard config spec for each developer.

Labels

Labels will be used to define various lineups. Significant labels include:

- TC_BASELINE_0: This represents the initial state of the project.
- TC_BUILDFILES: This label will indicate what element versions should be included in the next automated build. Only the builder should use this label.
- TC_LATEST_STABLE: This label will be applied to the most recent stable lineup on the integration branch. Note that this label is not fixed - the elements it refers to will change whenever a new stable lineup is established.

Initial Setup

Before starting with the parallel development process outlined here, it is assumed that the model is already under version control in a VOB.

Create the Integrator View

All project setup can occur from the integrator view. The integrator view will see the latest versions of elements on the integration branch, which in this case is "/main". The config spec should look like this:

```
element * CHECKEDOUT
element * /main/LATEST
```

Views are created with this config spec by default, so create a view with the name tc_int. If the integrator role will be played by multiple team members, be sure to choose a storage location for the view that will provide suitable performance for all. As always, integrators should not share views and so no two integrators should use this view at the same time.

Create Project Labels

The standard project labels mentioned above should now be created. These labels include TC_BASELINE_0, TC_BUILDFILES, and TC_LATEST_STABLE.

Each of these labels should be created before starting work on the project. A label type can be created with the following cleartool syntax:

```
[x:\dev]cleartool mklbtype -c "Initial Project State"
TC_BASELINE_0
```

Created label type "TC_BASELINE_0".

Create Initial Lineup

After the labels have been created, the initial lineup label should be applied to the VOB (\dev is the VOB being used in this example):

```
[x:\dev]cleartool mklabel -recurse TC_BASELINE_0 \dev
```

The initial model should be a valid stable model, so the TC_LATEST_STABLE label should be applied to all versions that are covered by the initial lineup:

```
 [x:\dev]cleartool mklabel -recurse -version TC_BASELINE_0
-
replace TC_LATEST_STABLE \dev
```

Creating the Developer View Template

To ensure consistent and controlled access to the model, and to ease the use of lineups and private branches, all developers should derive their config specs from a common base. There are two primary functions that developers will be performing, and each requires a different config spec:

- Browsing. Allows the view to see the latest stable lineup on the integration branch.
- Development. This sees a snapshot of the integration branch based on a labeled stable lineup, and branches files to a developer-private branch when files are checked out.

The rules for the browsing config spec are as follows:

```
element * TC_LATEST_STABLE
element * /main/LATEST
```

The TC_LATEST_STABLE label in the rule above can be changed to a different label if a developer wishes to view a lineup other than the latest. Optionally, the -nocheckout modifier can be added to the above rules so that checkouts can not occur accidentally while browsing.

For the development config spec, the rules should be:

```
element * CHECKEDOUT
element * ...\paulr_timing\LATEST
mkbranch paulr_timing

element * TC_BASELINE_5
element * \main\LATEST
```

In these rules, "paulr_timing" is the name of the private branch on which the development is taking place and TC_BASELINE_5 is the stable lineup that the development is based on. The rules have the following meaning:

- All versions checked out to the view will be seen.
- If there is no checked out version, then the latest version on the private branch will be seen.
- If there is no version on the private branch, then take the version labeled by the lineup.
- If an element from the lineup is checked out, immediately branch it to the private branch, and check out the newly branched version.
- If an element does not exist on the private branch and does not have the lineup label applied to it, simply choose the latest version on the main branch.

Windows NT

The developer view template can be implemented using view profiles by creating and maintaining a view profile, and having each developer associate their view with the view profile. Using the ClearCase View Profiles tool, create a new view profile using the supplied wizard, entering the following details:

- Name: tc_dev_profile
- Include the storage VOB for the model
- The work will take place on the LATEST versions of the integration branch
- Give the label for the initial lineup, TC_BASELINE_0, as the checkpoint label for creating private branches. This is not used for the default config spec, but instead marks TC_BASELINE_0 as a possible branching point.
- The diagram annotation can be modified as appropriate.

The default browsing config spec produced will look similar to the following:

```
# [CC_PROJECT - Checked Out Rule
element * CHECKEDOUT
#
#   Any modifications to the Profile config spec should
#   be made following this comment.

# CC_PROJECT]

# [CC_PROJECT - Profile Config Spec
# Do not directly modify the text below, it has been
# automatically generated by the ClearCase View Profile
# Tool. To change the Profile config spec, use the
# ClearCase View Profile Wizard to update the Profile
# status as needed.

element * \main\LATEST
# CC_PROJECT]
```

Unfortunately, this config spec will let developers see changes that have been merged to the integration branch but that have not yet been built and tested. What is wanted instead is a config spec that shows the latest stable build at any point in the development process. The change required is shown below:

```
# [CC_PROJECT - Checked Out Rule
element * CHECKEDOUT
#
# Any modifications to the Profile config spec should
# be made following this comment.
# CC_PROJECT]

element * TC_LATEST_STABLE
# [CC_PROJECT - Profile Config Spec
# Do not directly modify the text below, it has been
# automatically generated by the ClearCase View Profile
# Tool. To change the Profile config spec, use the
# ClearCase View Profile Wizard to update the Profile
# status as needed.
element * \main\LATEST

# CC_PROJECT]
```

The view profile is now ready for developers to use.

Unix

Use the supplied vtadmin script to create a new template. The following command syntax can be used:

```
 vtadmin -mktemplate -template tc -lateststable
TC_LATEST_STABLE
 -buildlabel TC_BUILDFILES -integrationbranch /main
```

After the command finishes, a template with the supplied parameters will have been created in the $CCVIEWTEMPLATES directory, and is now ready for use in the project.

To add the initial lineup label as a supported branching point, use the following call to vtadmin:

```
vtadmin -addlineup -template tc -baselinelabel
TC_BASELINE_0
```

Automated builds

To provide the ability to selectively choose the versions of files that go into the build, the builder will select all versions that are labeled with the build label TC_BUILDFILES. This allows flexibility in changing the exact versions that go into the build should it be needed (i.e., if the most recent version of a file contains code that does not compile, then the previous version can be labeled instead).

There are several steps involved in the build:

- Label Build Files
- Perform Build
- When the Build Completes Successfully
  - Create a new lineup label and apply to build file versions
  - Apply TC_LATEST_STABLE to build file versions
  - Make New Lineup Available to Developers

Before any of this can occur, though, the build view must first be created.

Create the Build View

The build view is similar to the integrator view in that it selects files from the integration branch, but different in that it needs to select labeled versions when performing the build.

When performing the labeling, the latest version of files on the integration branch need to be in view for the labeling to select the correct file versions. This config spec is identical to the one presented above for the integrator.

When performing a build, the build view must see the labeled version of all files that are contained in the build. For files and directories that are not labeled, it suffices to select the latest version on the main branch. The following config spec rules capture these requirements:

```
element * TC_BUILDFILES
element * \main\LATEST
```

For the build view to be used for both labeling and building, the config spec for the view must be switched back and forth. This can be done by having text files that contain the two config specs and using cleartool setcs to invoke the appropriate config spec.

Depending on your development environment, it may be possible to use the integrator view for labeling and leave the build view always configured to pick up the TC_BUILDFILES labeled files.

A typical name for the build view is "tc_build".

Unix

The view template scripts produce a text version of the build and integrator config spec rules indicated above. Use the vtsetview script to select the appropriate config spec rules into the build view.

Label Build Files

After ensuring that the current view has the integrator config spec, apply the TC_BUILDFILES label to the latest version of each element on the integration branch. The following command will do this:

```
cleartool mklabel -recurse -replace -version \main\LATEST
 TC_BUILDFILES \dev
```

Perform Build

After ensuring that the current view has the builder config spec, perform the build.

If the build does not complete successfully, or if the produced build does not pass sanity testing, determine if it is possible to fix the problem simply by backing up the version of a file used. If so, apply the TC_BUILDFILES label to the earlier version of the file and restart the build. Continue until a successful build is produced.

If there are build problems that cannot be resolved in the above manner, then ensure that the developers responsible for the problem are notified so that the next build will be successful.

When the Build Completes Successfully

Create a new lineup label and apply to build file versions

Create a label that will encompass all versions used in the build just completed. This should be a unique label in a regular form, such as TC_BASELINE_NNN, where N is an integer preferably generated automatically in an incremental manner from the previous lineup label.

Apply the label to all versions that were used in the build:

```
 cleartool mklabel -recurse -replace -version
TC_BUILDFILES
 TC_BASELINE_NNN \dev
```

If you wish to prevent the lineup contents from being changed in the future, you may wish to lock the TC_BASELINE_NNN build label at this point.

Apply TC_LATEST_STABLE to build file versions

As a convenience, the TC_LATEST_STABLE label is used to show the most recent successful stable build. To update the versions that TC_LATEST_STABLE applies to, use a similar mklabel invocation to the one presented above.

Make New Lineup Available to Developers

The newly labeled lineup should now be exposed for developers to use as a branching point for private branches. This is done by adding the TC_BASELINE_NNN label to the view template.

Although it may seem that TC_LATEST_STABLE could be added as a potential branching point label, this is not the case. Branching points are intended to be unchanging specifications of a lineup of versions.

However, TC_LATEST_STABLE will change with every build, and is therefore not appropriate for use as a branching point.

Windows NT

Using view profiles, the build label should be added to the tc_dev_profile view profile. This is done in the ClearCase View Profiles editor by using the context menu on the tc_dev_profile profile.

Unix

Use vtsetadmin to add the build label to the view template:

```
 vtadmin -addlineup -template tc -baselinelabel
TC_BASELINE_NNN
```

Developer Process

Each development activity is completed by a single developer and is performed on a private branch specific to that activity. Again, each developer requires their own view. The view is based on a branching point on the integration branch identified by a build label.

A unique branch name must be chosen that identifies the work being performed (such as paulr_timing). The view's config spec rules are set up to automatically check out and branch files from the branching point to the private branch. As well, new elements created during the development activity are immediately branched to the private branch.

Because the branch is hidden from other developers, the user may check in incremental changes to the branch. When the developer is satisfied that changes are completed and ready to be integrated, the developer informs the integrator that all changes on the private branch are ready for integration.

By basing developer private branches off of labels that correspond to the versions used by automated builds, each developer will be able to reuse most of the build results in the form of winked-in derived objects.

This significantly reduces the amount of building that is required by each developer when they make changes.

Creating a Developer View

It is important to note that each developer needs a view. Under no circumstances should multiple users work from the same view.

Windows NT

After creating the view, associate the view with the tc_dev_profile View Profile. The view will be set up for browsing as per the description in *Creating the Developer View Template*.

Unix

After creating the view, use the vtsetview script to set the view config spec to the default browsing config spec using the following command:

```
vtsetview -setview browse -template tc
```

The view will now show the latest stable build of the model.

Starting a Development Activity

Each development activity is performed on a private branch. The name of the private branch should be appropriate to the activity being worked on. One strategy for avoiding branch name clashes is to start each branch name with the user id of the developer doing the work (e.g., paulr_timing).

Windows NT

To start an activity, use the Set Up Private Branch wizard that is available from ClearCase HomeBase. Rather than base the branch on the elements currently in view, choose to use a different branch point. On the version selection page, choose "by View Profile checkpoint", and select the integration branch label you wish to work from, which is likely the most recent label in the list.

Unix

Use the vtsetview script with the -listbaselines option to see what lineups are available for basing the private branch on. To start the private branch, use the following invocation of vtsetview:

```
vtsetview -startbranch -template tc
-brname paulr_timing -brpoint TC_BASELINE_4
```

Working on a Development Activity

After the view has been set up like this, the model should be loaded into Rose. Work now proceeds until the entire development activity is complete. The developer may check in intermediate results, as they will not be seen by other developers since the changes will all occur on the private branch.

Finishing a Development Activity

When all development is complete on the activity and everything submitted to source control, the changes are ready to be propagated to the integration branch. The propagation is performed by the integrator, so the only task remaining for the developer is to end the private branch and notify the integrator that the changes on the completed branch are ready for integration.

Windows NT

Use the Finish Private Branch wizard in ClearCase HomeBase. Since integration of the changes made onto the integration branch will be done by the integrator, choose to leave the changes on the branch.

Unix

Use the following call to vtsetview to finish the private branch:

```
vtsetview -endbranch -template tc -brname paulr_timing
```

Integration process

Each development activity must eventually be merged into the integration branch. ClearCase has several tools available for performing such a merge. The "cleartool findmerge" command can be used to merge all changes from a branch onto another branch. From the integrator view, the following command syntax can be used:

```
 cleartool findmerge \dev -all -fversion
.../paulr_timing/LATEST
 -merge
```

Alternately, Windows NT users can use the ClearCase Merge Manager to perform the same merge.

Both of these methods will merge directory versions and also use Model Integrator to merge changes in model files. After performing the merge, the integrator should load the model into Rose and verify that no merge errors have occurred. If the model loads correctly, the changes should be checked in using the Tools > Version Control.

The following sequence of steps is quite efficient when integrating a series of development activities:

1. Load the model in the integrator view using the workspace.
2. Perform the merge as detailed above.
3. Reload all files that have changed in the merge.
4. Make sure that the merged differences are as desired.
5. Check changes into version control.
6. Repeat steps 2 through 5 for each activity that needs integration.

View Template Script Usage

vtadmin

The vtadmin script is used to list, create, delete, and update view templates. Each usage of vtadmin is detailed below:

```
vtadmin -lstemplates
```

This invocation lists the available view templates.

```
vtadmin -mktemplate -template <templatename>
-lateststable <stablelabel> -buildlabel <buildlabel>
 [-integrationbranch <intbranch>]
```

This invocation creates a new template with the specified name, latest stable label, build label and integration branch. If the integration branch is not supplied, then /main is assumed. Note that creating a view template does not create the labels and branches indicated - they are assumed to already exist.

```
vtadmin -lslineups -template <templatename>
```

This invocation lists the lineup labels associated with the specified view template.

```
vtadmin -addlineup -template <templatename>
 -lineuplabel <lineuplabel>
```

This invocation adds a lineup label to the specified view template.

```
vtadmin -rmlineup -template <templatename>
 -lineuplabel <lineuplabel>
```

This invocation removes the indicated lineup label from the specified view template.

When invoked with no parameters the script will output usage help.

vtsetview

The vtsetview script is used to configure config spec and perform common developer queries. Each usage of vtsetview is detailed below:

```
 vtsetview -startbranch -template <templatename>
 -brname <branchname> -brpoint <labelname>
```

This invocation attempts to start a private branch using the supplied parameters.

```
vtsetview -endbranch -template <templatename>
 -brname <branchname>
```

This invocation is used to end the indicated private branch.

```
vtsetview -setview (integrate | build | browse)
 -template <templatename>
```

This invocation is used to set a specific config spec into the current view.

```
vtsetview -lslineups -template <templatename>
```

This invocation lists the available lineups for the specified view template.

When invoked with no parameters, the script will output usage help.

Index

Symbols

$CURDIR 26
$UNNAMED$ 38

A

access violations 27, 28, 103
activating a Version Control Add-In 79
adding controlled units to a model 13
adding controlled units to version control 86
adornments on diagrams 14
Architect 96
architecture, model 95
artifacts, and virtual path maps 26
Auto Advance 55
automated builds 135
automatic merge 35
automating model validation 126
AutoMerge 50

B

base model 34
baseline 77
basic objects 36
browser view (Model Integrator) 32
build view 136
building using automated scripts 122

C

cat files 8
Check Merge 39, 62
Check Model 27
checking in controlled units 87
checking out controlled units 88
ClearCase
  about 80
  add-in 79
  and Model Integrator 42, 82
  config spec 82
  setting up 82
  views 82
comand line access to Model Integrator 67
compare mode 31
comparing models
  about 31, 34
  conflicts 36
  differences 36
  loading controlled units 45
component instances 109
config spec 82
conflicts 35, 36
  Auto Advance 55
  viewing 54
context field (Model Integrator) 49
contributors 31, 50
  about 34

accepting changes from 58
disabled by semantic checking 62
selecting 44
controlled units
about 7
access violations 27
adding to a model 13, 19
adornments 14
and version control 7
cat files 8
checking into version control 87
checking out of version control 88
contents 9
creating 12
getting latest from version control 89
icons 13
importing 19
loading 12, 45
logical packages 8
merging 19
organizing 29
partitioning a model 118
prc files 8
protecting 17
reloading 13
removing from version control 90
sharing 10
splitting 18
sub files 8
uncontrolling 20
undoing check out 89
unloading 14, 45
unresolved references 27
virtual path maps 21
write-enabling 18
write-protecting 17
correcting merge errors 63
creating
build view 136
controlled units 12
developer view template 132
initial lineup 132
integrator view 131
labels and lineups 120
model workspace 17
processors and component instances 108
project labels 131
reusable build artifacts 124
virtual path maps 23

D

defining subsystem interfaces 105
deployment view 8
diagram objects 36
differences 35, 36
Auto Advance 55
changing nodes 60
viewing 54

E

element type 82
export control 28

G

getting latest version of controlled units 89

I

icons, controlled units 13
import relationship and access violations 28
importing controlled units 19
integrating change 126
integration 73
Integrator 98
interfaces, and subsystems 105

L

labels 120, 131

creating 131
for build files 136
lineups 120, 121
about 73
creating initial 132
loading a model workspace 17
loading controlled units 12, 45

M

manually loading controlled units 13
mdl file 7
mechanism 37
merge mode 31
merging controlled units 19
merging models
about 31, 35
accepting changes from contributors 58
automatic merge 35
AutoMerge 50
Check Merge 62
Check Merge function 39
conflicts 35, 36
correcting merge errors 63
differences 35, 36
loading controlled units 45
partial merge 65
preparing for 44
selective merge 35
semantic checking 39
using subtree mode 61
whole models 42
without a base model 66
Microsoft Visual SourceSafe 84
Model Architect 96
model architecture
about 95
Model Integrator
about 31
accepting changes from contributors 58
and ClearCase 42, 82
Auto Advance 55
automatic merge 35
AutoMerge 50
base model 34
basic objects 36
browser view 32
changing nodes with differences 60
Check Merge 62
Check Merge function 39
comparing models 34
conflicts 35, 36
contributors 34
correcting merge errors 63
diagram objects 36
differences 35, 36
mechanism 37
merging models 35
merging models without a base model 66
partial merges 65
property view 33
quids 37
references 37
resolving subunit loading errors 48
searching for nodes 53
selecting contributors 44
selective merge 35
semantic checking 38
semantic checking on-the-fly 62
setting a new context for subunits 49
starting 44
subtree mode 61
text views 33
unnamed objects 38
user interface 32
using from a command line 67
view objects 37
viewing a parent node 56
viewing a single model 67
viewing conflicts 54
viewing differences 54

viewing references to nodes 57
virtual path maps 48
Model Manager 96
model properties
controlled units 8
using virtual path maps 26
model validation 126
model workspaces
about 14
creating 17
loading 17
loading controlled units 13
saving 17
Modeler/Developer 98
moving the contents of a controlled unit 19

N

nodes
about 35
changing, with differences 60
searching for 53
viewing a parent 56
viewing references to 57

O

on-the-fly semantic checking 40
opening a model 12
organizing controlled units 29

P

packages
access violations 103
as subsystems 99
component 8
logical 8
partitioning a model 118
sharing 10
parallel stream versioning 75
partial merges 65
partitioning a model 10, 29, 95, 118
path maps, See virtual path maps
performing a build 137
petal file format 11
petal files 38
prc files 8
project level processors 108
property view (Model Integrator) 33
protecting controlled units 17
prp files 8
ptl files 11

Q

quids 37
quidu 37

R

RCS 91
read only controlled units 17
references
checking 27
in Model Integrator 37
unresolved 14, 20
releasing subsystems 109
reloading controlled units 13
removing controlled units from version control 90
repository mapping files 92
resolving subunit loading errors 48
reuse
creating build artifacts 124
rmf files 92

S

saving a model workspace 14, 17
SCC version control systems 78
SCCS 91
searching for nodes 53

selective merge 35
semantic checking 38
  on-the-fly 62
setting a new context for subunits 49
setting ClearCase up to work with Rose 82
setting up compatible workspaces 117
setting up Microsoft Visual SourceSafe to work with Rose 84
Show Access Violations 28, 103
single stream versioning 74
splitting a controlled unit 18
splitting a model into subsystems 110
starting Model Integrator 44
sub files 8
subsystem level processors 108
subsystems
  components in 105
  defining interfaces 105
  releasing 109
  splitting a model 110
Subtree Mode 61
subunits 12

T

text views (Model Integrator) 33

U

uncontrolling controlled units 20
undoing check out 89
unit testing 107
unloading controlled units 14, 45
unnamed objects 38
unresolved references 14, 20
  checking for 27
URLs 26
using Model Integrator form a command line 67
using view templates 129

V

validating a model 126
version control
  about 71
  activating 79
  adding controlled units 86
  checking in controlled units 87
  checking out controlled units 88
  controlled units 12
  getting latest controlled units 89
  removing controlled units 90
  setting up 118
  uncontrolling controlled units 20
  undoing check out 89
  write-protecting controlled units 18
Version Control Add-In 78
versioned object base, see VOB
view
  version control system 72
view objects 37
viewing a parent node 56
viewing a single model file (Model Integrator) 67
viewing conflicts and differences 54
viewing model elements that have moved 56
viewing references to nodes 57
views
  ClearCase 82
views (Model Integrator) 32
virtual path maps
  about 21
  creating 23
  for artifacts 26
  for model properties 26
  how stored 27
  in Model Integrator 48
  using another path map 25
  using wildcards 25
VOB
  about 81

VSS 84
vtadmin 141
vtsetview 142

W

workspaces, model
    about 14
write enabling a controlled unit 18
write protecting controlled units 17
wsp files 17