## Contents

**PREFACE**

Overview ................................................................. xiii
About This Manual ................................................... xiii
  Suggested Learning Paths ........................................ xv
  Notation Conventions .............................................. xv

**Chapter 1**

**KEY CONCEPTS**

Overview ................................................................. 17
Packages ........................................................................... 18
  Units of Functionality ................................................. 18
    Simple Packages ..................................................... 20
    Using a Package to Hold Classes Related
      by Inheritance .................................................... 20
    Using a Package to Hold Cooperating Classes ......... 20
  Dividing Work Among Team Members ...................... 21
Characteristics of Packages ........................................... 22
  Package Operations .................................................. 23
    Opening a Package ................................................ 24
    Loading a Package ............................................... 24
    Committing a Package .......................................... 25
    Migrating to a Different Revision of a Package ....... 25
    Unloading a Package ........................................... 25
    Closing a Package ............................................... 25
Definitions ....................................................................... 25
  Kinds of Definitions ................................................. 26
    Initializers .......................................................... 26
    Versions of Definitions ........................................ 26
    Annotating Definitions ......................................... 27
Class-Related Definitions .............................................. 27
  Methods ............................................................... 27
  Instance Variables ................................................. 27
  Class Variables .................................................... 28
  Class Instance Variables ........................................ 28
  Pool Usage .......................................................... 30
Class Initializers ........................................................ 30
Global Variable Definitions .......................................... 31
Pools and Pool Variable Definitions .............................. 31
Ad Hoc Initializer Definitions ....................................... 32
Clusters ............................................................................................33
Characteristics of Clusters .............................................................35
Specifications ....................................................................................35
Configurations and Groups ............................................................38
Cluster Operations .............................................................................38
The Top-Level Cluster .....................................................................39
Resolving Specifications and Loading Clusters .............................40
Using Groups and Configurations .................................................41
  Using Groups to Record Relationships
  Between Modules ...........................................................................41
  Using Configurations to Record Revisions .................................41
Combining Groups and Configurations .......................................42
Modifying Clusters and Specifications .........................................43
Optional Specifications .....................................................................44
  Using Optional Specifications with Configurations .................44
Conflicts when Loading Optional Specifications ........................45
Conditional Specifications ..................................................................45
Specification Tracking ........................................................................46
Revision Control ................................................................................47
Revisions ........................................................................................48
  Package Revisions .........................................................................48
  Cluster Revisions ..........................................................................49
Revision Number Restrictions ......................................................49
Revision Attributes ..........................................................................50
Repositories ........................................................................................51
  Repository Operations ..................................................................51
Kinds of Repositories .........................................................................52
  PVCS Repository ..........................................................................52
  File-Based Repository .................................................................52
Access Control ..................................................................................52
Unresolved References ......................................................................53

Chapter 2 CONFIGURING YOUR WORKING ENVIRONMENT 55
Overview ..........................................................................................55
Enabling PVCS Support .................................................................55
Setting Your User Name .................................................................56
PVCS Administration .........................................................................57
  Setting up a Master Configuration File ......................................58
    Identifying the Master Configuration File ...............................58
  Master Configuration File Directives .......................................58
Setting Up Local Configuration Files ..........................................60
PVCS Capabilities that Conflict with
  Visual Smalltalk Enterprise ......................................................60
Configuring the Access Database ................................................61
  Creating an Access Database .......................................................61
<table>
<thead>
<tr>
<th>Chapter 3</th>
<th>TUTORIAL</th>
<th>73</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connecting to the Tutorial Repository</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>Loading the Tutorial Cluster</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>Viewing the Specifications in a Cluster</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>Modifying a Package</td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>Adding Category and Methods</td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>Changing Methods</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>Working with Previous Method Versions</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>Testing the Map Browser</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>Creating a New Revision of a Package</td>
<td>88</td>
<td></td>
</tr>
<tr>
<td>Updating Configuration Clusters</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>Specification Tracking</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>Creating a New Revision of a Cluster</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>Comparing Packages</td>
<td>92</td>
<td></td>
</tr>
<tr>
<td>Making a New Package</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>Updating the Map Browser Configuration Cluster</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>Distributing Your Changes</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Migrating to a New Revision</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Loading a New Revision</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Delivering the Map Browser as a Smalltalk Link Library</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 4</th>
<th>BROWSERS AND TOOLS</th>
<th>101</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>Locating Modules with the Browsers</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>Additions to the Visual Smalltalk Menus</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>Text Display Conventions</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>Using Drag and Drop</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>Package Browser</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>Package Browser Panes</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>Package Browser Contents Pane</td>
<td>109</td>
<td></td>
</tr>
<tr>
<td>Package Browser Module Editor</td>
<td>109</td>
<td></td>
</tr>
<tr>
<td>Package Browser Global Variable Editors</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>History Browser</td>
<td>157</td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>History Browser Version Menu</td>
<td>158</td>
<td></td>
</tr>
<tr>
<td>Browsing Specific Definitions</td>
<td>158</td>
<td></td>
</tr>
<tr>
<td>Definition Group Browser</td>
<td>158</td>
<td></td>
</tr>
<tr>
<td>Definition Group Browser Definition Menu</td>
<td>159</td>
<td></td>
</tr>
<tr>
<td>Message Browser</td>
<td>160</td>
<td></td>
</tr>
<tr>
<td>Comparison Browsers</td>
<td>161</td>
<td></td>
</tr>
<tr>
<td>Version Comparison Browser</td>
<td>161</td>
<td></td>
</tr>
<tr>
<td>Version Comparison Browser Left and Right Menus</td>
<td>162</td>
<td></td>
</tr>
<tr>
<td>Version Comparison Browser Compare Menu</td>
<td>163</td>
<td></td>
</tr>
<tr>
<td>Definition Comparison Browser</td>
<td>163</td>
<td></td>
</tr>
<tr>
<td>Definition Comparison Browser Compare Menu</td>
<td>165</td>
<td></td>
</tr>
<tr>
<td>Definition Comparison Browser Left and Right Menus</td>
<td>165</td>
<td></td>
</tr>
<tr>
<td>Package Comparison Browser</td>
<td>166</td>
<td></td>
</tr>
<tr>
<td>Package Comparison Browser Buttons</td>
<td>167</td>
<td></td>
</tr>
<tr>
<td>Package Comparison Browser Menus</td>
<td>168</td>
<td></td>
</tr>
<tr>
<td>Build Library Dialog</td>
<td>169</td>
<td></td>
</tr>
<tr>
<td>Unresolved References Browser</td>
<td>170</td>
<td></td>
</tr>
<tr>
<td>Unresolved References Browser Variable Menu</td>
<td>171</td>
<td></td>
</tr>
<tr>
<td>Access Control Editors</td>
<td>172</td>
<td></td>
</tr>
<tr>
<td>Repository Access Editor</td>
<td>172</td>
<td></td>
</tr>
<tr>
<td>Repository Entity Access Editor</td>
<td>173</td>
<td></td>
</tr>
</tbody>
</table>

Chapter 5  TEAM PROGRAMMING TASKS 175

A Development Process Scenario | 176
Installing Unpackaged Code | 178
Importing File-in Format Code | 180
Differences Between Installing and Loading File-Ins | 180
Procedures for Importing File-In Code | 180
Installing Code from a Smalltalk Library | 181
Organizing Imported Code into Packages | 182
Using Packages | 182
Dividing an Application into Packages | 182
Creating a Package | 185
Reorganizing Packages | 185
Adding Code to a Package | 187
Reordering Definitions | 187
Using Method Categories | 188
Restructuring a Class Hierarchy with Packages | 190
Initializing Classes | 190
Using ad hoc Initializers | 192
Extending the Base Class Library | 194
System Packages | 194
Problems with Modified Base Classes .......................195
Modifying Base Classes Safely ................................. 195
Using Clusters ............................................................. 196
  Using Conditional Inclusion Flags ......................... 198
  Using Optional Inclusion Flags .............................. 199
Creating a Cluster ..................................................... 200
Adding a Specification to a Cluster .......................... 200
Making a Specification Optional ............................ 201
Adding a Conditional Inclusion Flag to a Module ........ 202
Setting Conditional Inclusion Flags in the Environment ... 202
Interpreting Module Specifications ............................ 203
  Expanding and Contracting Cluster Specifications 204
Converting Between Clusters, Groups, and
  Configurations ..................................................... 204
  Converting a Cluster to a Group ............................ 205
  Converting a Cluster to a Configuration .................. 206
  Minimizing a Complete Specification .................... 206
Completing Partial Specifications ............................ 207
Using Revisions .......................................................... 207
  Keeping Track of Your Work ................................. 207
  Version History .................................................... 208
Modifying a Module .................................................. 211
Making a New Revision of a Module ......................... 211
Identifying the Revision of a Module ....................... 213
Viewing the Revisions of a Module .......................... 213
Examining an Older Revision ..................................... 214
Comparing Versions of Source-Based Definitions ....... 215
Reverting to an Older Version of a Definition .......... 216
Comparing Revisions of Packages ............................ 216
Reverting to Another Revision of a Module ............... 218
Working with Repositories ......................................... 219
  Determining What Kind of Repository to Use .......... 219
  Creating a Repository .......................................... 220
  Backing up a Repository ....................................... 220
  Controlling Repository Access ............................. 221
  Assigning Repository Access Rights ....................... 222
  Assigning Repository Entity Access Rights ............... 224
  Locking ............................................................ 225
  Optimistic Versioning ......................................... 225
  Pessimistic Versioning ........................................ 226
Teamwork ............................................................... 227
  Sharing Code Without Access to a Common Repository . 227
  Detaching from a Repository to Work Off-Site .......... 228
  Editing a Package Without Loading it into the Image ... 229
  Resolving Conflicts ............................................. 230
Reusing Code from Another Package ................................ 232
Loading Multiple Packages to Resolve Circularities .......... 232
Extending Classes Defined in Other Packages ............... 234
Integrating Revisions and Packages ......................... 235
Rebuilding an Image .................................................... 236
Delivering An Application .............................................. 237
Distributing Code in Packages .................................. 237
Distributing Code without Packages .......................... 238
Building Smalltalk Libraries ................................. 239
   Build from a Clean Image .......................................... 239
   Building a Smalltalk Library From a Module ............ 239
Customizing a Build Script ........................................ 241
Filing Out a Module Source Code .............................. 243

Chapter 6   EXTENDING THE TOOL INTERFACE 245
Semantic Interface ....................................................... 246
TeamVInterface Service ............................................... 247
Subsystem ................................................................. 248
   Enumerating over Subsystem Components .............. 249
   Subsystem Searching .............................................. 249
   Migration ............................................................. 251
   1.4 Initialization .................................................. 254
   Classification and Testing .................................... 254
Modules ......................................................................... 255
   Module Components .............................................. 255
   Searching through Modules .................................. 256
   Enumerating over Module Components .................... 257
   Editing Cluster Contents ....................................... 260
Definitions ....................................................................... 261
   Executables and Values ......................................... 262
   Source-Based Definitions ....................................... 263
   Source-Based Searching: The Grep Example ............ 264
   Editing Source ....................................................... 265
   Version History ...................................................... 265
Variable Definitions ..................................................... 266
   Pool Searching and Enumeration ......................... 266
   Constant and Variable Designation ..................... 266
   Creating Variables ................................................ 266
   Creating a Pool ....................................................... 267
Classes and Methods .................................................... 268
   Classes ................................................................. 268
   Searching through Classes .................................. 269
   Enumerating over Class Components .................... 270
   Editing ................................................................. 273
Chapter 7 SOM TOOLS FOR OS/2 297

Installing the SOM Tools ..............................................................297
Installing the SOM Tools from Smalltalk Libraries ..................297
Loading the SOM Tools from Clusters ........................................298
The SOM Browsers .......................................................................298
SOM Class Browser .....................................................................299
SOM Type Definition Browser ..................................................300
Building Smalltalk Proxy Classes ...............................................301
Proxies for SOM Typedefs .........................................................301
Mirroring the SOM Class Hierarchy ...........................................302
Mapping SOM Names to Smalltalk Class Names .......................302
Building Smalltalk Classes Into Unloaded Packages ..................303
Automatically Building Referenced Classes ...............................303
Proxy Methods for SOM Classes ..............................................304
The Build Options Menu ..............................................................304
SOM Class Browser Class Menu ...............................................306
SOM Type Definition Browser TypeDef Menu ...........................306
SOM Implementation Packages .......................................................... 307
SOM Base Cluster ........................................................................... 307
SOM Tool Cluster ........................................................................... 308
SOM Proxy Map ........................................................................... 308

Appendix A  PROBLEM SOLVING ......................................................... 311
Conflicts and How to Resolve Them .................................................. 311
Conflict Notification Dialog .............................................................. 312
Resolving Conflicts ........................................................................... 312
Existing Definition Conflict ............................................................. 313
Multiple Definition Conflict ............................................................ 313
Type Mismatch Conflict ................................................................ 313
Existing Subclasses Conflict ............................................................ 314
Undeclared Superclass Conflict ......................................................... 314
Undeclared Global Conflict ............................................................... 314
Variable Name Conflict ................................................................ 315
Indexable Type Conflict ................................................................ 316
Pool Reference Conflict ................................................................ 317
Undeclared Pool Conflict ................................................................. 317
Existing Instances Conflict ............................................................... 317
Multiple Specification Conflict ........................................................ 318
Unresolved Specification Conflict ..................................................... 318
Recovering From an Image Crash ..................................................... 319
The Change Log ............................................................................. 319
The Recovery Log ........................................................................... 319
The Recovery Log Workspace .......................................................... 320
Recovery Log Workspace Recover menu ......................................... 320
Recovery Log Format ...................................................................... 321
Using the Recovery Log Workspace to Recover Changes .............. 321
Working with Inaccessible Repositories ........................................... 323
The --WORK-- Directory ................................................................. 323
Detaching from Repositories ........................................................... 323
Working Without a Repository ......................................................... 324

Appendix B  METHOD FORMATTING .................................................... 325
Programmatic Interface to the Formatter ........................................... 325
Methods ........................................................................................... 326
The Method Header .......................................................................... 326
Method Comments ........................................................................... 327
Temporary Declarations ................................................................. 327
Primitive Specifications ................................................................. 327
The Code Block .............................................................................. 327
Code Blocks ..................................................................................... 328
Statements ...................................................................................... 328
Blocks .............................................................................................. 328
Preface

Overview

Visual Smalltalk Enterprise provides extended capabilities beyond those provided by Visual Smalltalk. Most of the extensions consist of the expanded tool set and support for team development. These capabilities in turn depend on an enhanced code management system. Rather than writing your Smalltalk code and saving it in the Smalltalk image, in Visual Smalltalk Enterprise you save your code in storage modules called “packages.” From the code in these modules, you or another programmer can reconstruct the application.

About This Manual

This manual provides comprehensive information about the tools in the extended Visual Smalltalk Enterprise development environment, together with instructions on how to use them to make your development efforts more efficient.

The manual is organized into the following chapters:

- Chapter 1, Key Concepts, discusses the main ideas underlying the Visual Smalltalk Enterprise environment: clusters, packages, definitions, and revisions. The following chapters explain the tools and tasks, but you should read this chapter in order to make the most effective use of the tools.

- Chapter 2, Configuring Visual Smalltalk Enterprise, explains how to set up the environment, especially how to configure Intersolv’s PVCS.

- Chapter 3, Tutorial, provides a tutorial to help you become familiar with the basic operations involved in using Visual Smalltalk Enterprise tools: making clusters, packages, defining items, and using definitions. Running this tutorial is recommended, especially if you are the sort of person who learns best from hands-on exercises.

- Chapter 4, Browsers and Tools, provides a thorough reference to the user interface features of all browsers and other tools.
specific to the Visual Smalltalk Enterprise development environment. This reference assumes familiarity with the basic Visual Smalltalk development environment.

- Chapter 5, *Team Programming Tasks*, explains how to perform a variety of development procedures in step-by-step fashion. In addition to *how*, it also provides guidance on *when* and *why* you might need to perform these procedures. You do not need to read this chapter from beginning to end. Instead, consult it as a reference for the answer to a particular question.

- Chapter 6, *Extending the Tool Set*, describes the underlying programmatic interface. The programmatic interface maps the semantic components of the Visual Smalltalk Enterprise tools into classes that you can use to create your own custom tools and browsers.

- Chapter 7, *SOM Tools for OS/2*, describes the optional tools that are available to support the interaction with the OS/2 System Object Model (SOM). This information is only applicable for OS/2.

- Appendix A, *Problem Solving*, describes the conflicts that can occur between packages (or between a package and the base class library under certain circumstances), and tells how you can resolve them. It also describes the Recovery Log workspace, a tool used to recover from crashes, and how to work with inaccessible repositories.

- Appendix B, *Formatting Methods*, describes the code formatting conventions used when you choose *Method / Format* in the Package Browser and the programmatic interface to the formatter.
## Suggested Learning Paths

<table>
<thead>
<tr>
<th>If you:</th>
<th>Then:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Want an overview of the main concepts of Visual Smalltalk Enterprise</td>
<td>Read Chapter 1</td>
</tr>
<tr>
<td>Want to modify the environment configuration</td>
<td>Read Chapter 2</td>
</tr>
<tr>
<td>Want a hands-on exercise</td>
<td>Read Chapter 3 and run the tutorial</td>
</tr>
<tr>
<td>Want information on specific browsers, menus, and other user interface features of the team tools</td>
<td>Consult Chapter 4</td>
</tr>
<tr>
<td>Want guidance on how, when, and why to perform specific programming tasks</td>
<td>Consult Chapter 5</td>
</tr>
<tr>
<td>Want to interact with SOM objects in OS/2</td>
<td>Consult Chapter 7</td>
</tr>
<tr>
<td>Want to see a complete list of conflicts between packages, and how to resolve them</td>
<td>Consult Appendix A</td>
</tr>
<tr>
<td>Want to learn how the tools format methods when you choose Method / Format</td>
<td>Consult Appendix B</td>
</tr>
<tr>
<td>Want to learn how to build your own tools and browsers using the programmatic interface</td>
<td>Consult Chapter 6</td>
</tr>
</tbody>
</table>

## Notation Conventions

We use the following notation conventions in this *User's Guide* to help you understand what you are reading:

- Proper names of actual classes, variables, methods, instances, and other Visual Smalltalk structures are printed in **boldface**, with appropriate capitalization, as required. For example, contrast “class **Dictionary**” with “a dictionary class.”

- Examples of Visual Smalltalk code are printed in **boldface** in the **standard** font. Spaces may be inserted between arguments for easier reading, though they are not necessary. For example:

  ```plaintext
  myLongitude degrees: 124 minutes: 22 seconds: 12.
  myLatitude degrees: 46 minutes: 55 seconds: 8.
  MapBrowser initialize.
  MapBrowser open
  ```
• Titles of chapters or sections of text are printed in *italics* when their proper name is quoted.

• Placeholder names that are meant to be substituted with actual names appear in italics. For example:

  The class *className* has no superclass.

• Menu names and menu items are printed in *italics* and are separated by a slash when both appear together. For example, *Method / Format* refers to the pull-down menu path activated by clicking with the selection button, normally the left mouse button, when the cursor is over the keyword *Method* in the menu bar (the area in the window just below the title bar) followed by a click on the *Format* menu item.

• Steps of instruction are printed in **boldface** in an alternate font. They are numbered, or if only one instruction is shown, prefixed by a right angle bracket (>).
Visual Smalltalk is a free-form programming environment that imposes few organizational restrictions on the developer. However, as development projects and teams grow, managing the source code becomes an increasingly difficult problem.

To help manage large projects, Visual Smalltalk Enterprise includes additional tools and facilities, providing a greater ability to manage and structure large development projects. New *browsers* and *tools* extend the development environment. A set of *revision control* mechanisms allow you to manage and coordinate the process of change and growth.

The enhancements provided by Visual Smalltalk Enterprise center around the use of *packages* to store application code and provide an organizational structure for the contributions of individual team members. Packages can be collected and organized into *clusters*, providing additional organizational structure. The packages included in a cluster can be specified with more or less detail, making the mechanism a very flexible way to specify an application.

Packages explicitly store the definitions that make up your application. Because the definitions are stored in a format independent of the image in which they are created, packages can be loaded into other Visual Smalltalk Enterprise images, making them an ideal code source format for group development.

Each change you make to a definition results in a new *version* of that definition. This change is recorded locally in your image files. When you commit the package containing that changed to a repository, a new *revision* of that package is created with an identifying *revision number*.

These two mechanisms, versions and revisions, record a complete history of the revision process.

Integrating the contributions of individual programmers into an application is simplified by the following benefits:
Forward and external references are resolved before loading application components.
- Conflicting items are reported.
- Applications are stored separately from a virtual image.
- A clearly traceable history of the evolution of the application is kept.
- Applications can more easily be delivered as discrete, executable programs.

In this chapter, we discuss the concepts underlying the Visual Smalltalk Enterprise development environment. This should help keep you oriented in the later discussions of the tools and procedures as they are introduced.

Packages

A package is a unit of Smalltalk source code consisting of definitions in a specified order. A package can define classes, new methods for classes that are defined in other packages (called *loose methods*), global variables, pool variables, and other items (discussed in detail below). Packages are the units team members use to manage and share their code.

In addition to the clusters and packages (jointly referred to as *modules*) containing the extended tool set itself, two system modules are always present in your development environment:

- The cluster named **Visual Smalltalk Enterprise** contains the definitions that make up the base image.
- The package named **unpackaged** contains any definitions you create outside of a package. For example, new methods created using the Class Hierarchy Browser, or globals created with *Smalltalk at:put:*, are added to this default package.

You will learn later how to use these packages in conjunction with your own packages to develop and deploy your applications.

Units of Functionality

A package often encapsulates a single unit of functionality. A simple application can be contained in a single package. For example, suppose you want to create an application that lets users view a map, select any two points, and measure the distance between them. The functionality comprised by this simple utility can be encapsulated in the Cartography package. This package defines **CartographicUnit**, a three-part number consisting of
degrees, minutes, and seconds, and two subclasses \texttt{Latitude} and \texttt{Longitude}. \texttt{CartographicUnit} includes methods that define how to add and subtract geographic numbers. The package also defines a new method for class \texttt{Time}, \texttt{timeAtLatitude:Longitude:} that determines the current time at a given latitude and longitude.

The figure below shows the relationship between the Cartography package and the rest of the system.

\textit{Figure 1-1: The Cartography Package.}

A more complex application might consist of several packages. For example, suppose you were programming an application for linguists who want to track the movement of idioms across regions over time. Such an application could certainly make use of the Cartography package, but it would also need a Linguistics package that captured knowledge about phonemes, morphemes, dialects, populations of speakers, and other linguistic concepts. The user interface of such an application might also grow complex enough to merit its own package.
Clusters provide a means of grouping related packages into larger organizational units. The idiom tracking application is now a cluster containing three packages, as shown in the figure below. You’ll learn more about clusters later in this chapter.

Figure 1-2: Packages in the Idiom Tracking Cluster

**Simple Packages**

The simplest possible package defines only a single method or pool variable. While such a simple package is unlikely, a package need not be large or complex. For example, a package that defines a single class and a few related items if the class is an important one used by many different applications. Perhaps the package defines a specialized radio button that is reused in many applications.

**Using a Package to Hold Classes Related by Inheritance**

A package can define a class hierarchy. For example, a drawing program could use a package that defines the classes of objects that can appear in a drawing. These classes might be arranged in a hierarchy, with a superclass `DrawingObject` defining the functionality common to all such objects, and subclasses such as `Line` and `Polygon` adding their own unique characteristics to that functionality.

**Using a Package to Hold Cooperating Classes**

A package can define a group of collaborating classes. For example, a package to support printing might consist of a class `PrintServer`, classes for each of the different kinds of printers supported by the application, and the methods that define how
each of the objects in the application can print themselves. A package to provide a file system interface might define the classes **File**, **FileHandle**, and **FileStream**.

A package can also define an application framework. For example, a drawing editor might have the classes **Drawing**, **Figure** (the items in a drawing), **Handle** (of a figure), and **Tool** in a single package. Subclasses that use the framework to define specific figures and tools would be placed in other packages.

Packages can also be divided for future reusability. For example, a bus scheduling application might define the classes **Bus**, **BusStop**, and a class used by **BusStop** — **OrderedSet** — a subclass of **OrderedCollection** that does not allow duplicates. While the classes **Bus** and **BusStop** are clearly specific to the bus scheduling application, the class **OrderedSet** might have many future uses. It could be placed in its own package along with a method added to the base class library class **Collection**. This method, called **asOrderedSet** could convert an instance of one class to an instance of the other.

### Dividing Work Among Team Members

In addition to providing units of functionality, packages are also used to divide up work on an application among the team members and to distribute code.

If large, complex applications are to be finished in a timely manner, the work must be done by a team. Yet a team is in some sense an abstract concept; when you come right down to it, programming is done by individuals. In order for the team to get anything done, each member must be responsible for one (or more) well-defined units. Packages and clusters provide the primary method for organizing an individual's work within a team.

When a team begins work on an application, the members must decide how the application is to be structured. The structure of the application and the abilities of the team members determine the logical way to divide the work. The work is divided into packages, and each individual is responsible for one or more of these packages.

Since only packages contain code, the application is ultimately a collection of packages. When it comes time to integrate the application, packages are the units that each programmer shares with his or her teammates. When it comes time to deliver the product, the packages are integrated to create an executable application.
Definitions in loaded packages cannot overlap, which guarantees the separation of individual effort and detects accidental conflicts, such as two different programmers modifying the same method. These conflicts are reported when the packages from both programmers are loaded.

The team can use clusters to organize packages into larger functional units and to record where a package is reused in different functional areas. Clusters are also used to record how the packages fit together to make the application. Ultimately, the entire application can be recorded in a single cluster. If a family of applications is used together, this can be recorded by a cluster containing the application clusters.

**Characteristics of Packages**

A package has the following characteristics:

- **Name**
  A package can have any name, useful for identifying the functionality it provides. The name is a string that can be any length and contain any characters except that leading and trailing spaces are removed. The name may not be the same as that of any other package or cluster loaded in the image.

- **Annotations**
  Annotations are a formal mechanism to communicate certain key features of a package. An annotation has a name and a value, both arbitrary strings. A package can have any number of annotations, and each is defined to specify one key attribute.

  For example, one annotation might be named “testing status.” Its value might be any one of: “untested,” “functional but buggy,” “beta,” or “released.”

  Another annotation could be named “To Do” and would specify the functionality remaining to be implemented for the package under construction.

  Annotations are like comments, but they let a team establish formal conventions for describing particular attributes of a package. The Visual Smalltalk Enterprise tool interface can be used to automatically process packages based on their annotations, such as opening a browser on all packages that have a “To Do” annotation.
Annotations that begin with a period, like .Author, are automatically maintained by Visual Smalltalk Enterprise and are read-only.

- **Comment**
  A package comment is an arbitrary string. Like comments attached to any unit of code, it is useful for telling other programmers what they need to know about the package.

- **Revisions**
  When you commit a package to a repository, you create a new revision of that package, identified by its revision number. This mechanism lets you keep track of changes through a long development process. Revisions are discussed in greater detail below.

- **List of Definitions**
  The main feature of a package is the list of definitions that make up the program source code. Some definitions, such as classes and methods, are already familiar to you. Other definitions let you declare program items that you are probably used to creating by executing Smalltalk expressions. For example, in Visual Smalltalk Enterprise you use a global variable definition to create a global variable, instead of by evaluating a Smalltalk `at:put:` expression.

  For most definitions, order is not important. Those definitions that initialize certain kinds of items (known as initializers) are processed in the order they appear in the package. This order determines the sequence in which items are initialized.

### Package Operations

You can use the Visual Smalltalk Enterprise tools described in chapter 4 to perform operations on packages. These operations are:

- open
- close
- load
- initialize
- unload
- commit
- migrate
CHAPTER 1

Key Concepts

Opening a Package

When a package is open, you can view and edit its definitions, but they are not installed into the image. You can see what the definitions are, but nothing executable is actually added to the image.

Loading a Package

Loading a package causes the definitions to be installed into the development environment. Unlike the file-in process, loading is an atomic operation that either succeeds in its entirety or fails completely. Either all definitions in a package are loaded or none of the definitions in the package are loaded. A common cause of failure to load is a conflict with another definition already present in the development environment.

After the definitions in a package are loaded, the package can be initialized. Initialization involves evaluating certain expressions, called initializers. These initializers can be re-evaluated at any time by initializing the package.

A package is loaded as follows:

1. All definitions in the package are analyzed for conflicts with definitions in other loaded packages. If any problems are found, the load fails.

2. References to variable pools and superclasses by class definitions are resolved against the other definitions in the package and those already loaded in the image. If a reference cannot be resolved, the load fails.

3. All global definitions (classes, globals, pools) in the package are processed in an indeterminate order. Processing a class definition includes processing any instance variable, class instance variable, or class variable definitions for the class.

4. All method definitions are compiled and installed in their respective classes.

5. If you have requested initialization when you load or migrate, all initializers (class, pool variable, global variable and ad-hoc) are executed in the order that they appear in the package. This is the only place where the order of definitions in a package is significant.

Each of the kinds of definitions mentioned in the load algorithm above are discussed in detail in the Kinds of Definitions section below.
Committing a Package

The definitions in a package can be changed in either a loaded or opened package. Because these changes are image-based, they are not visible to other team members until the package is committed. If you want to create a new revision of the package reflecting the changes, you commit it to a repository. A repository, usually located on a shared file system, contains revisions of packages and clusters you want to store and share.

For example, when you make a change to a method, a version of the method is stored in your local development environment and the package in your image that contains the method is marked modified. In order to release your changes to your team members, or just record the current versions of the definitions in the package for your own reference, commit the package to a repository shared by the team. This creates a new revision of the package containing the current definitions and it is then available to other team members.

Migrating to a Different Revision of a Package

If you have changed definitions but don’t want to commit your changes, you can discard them by either migrating the modified package back to the original revision or unloading the package. You can also migrate to other revisions, either newer or older, of the package.

Unloading a Package

When a loaded package is unloaded, all its definitions and objects associated with those definitions are removed from the image.

Closing a Package

Closing an opened package is similar to closing an editor on a text file. Any changes you have made are cancelled, unless you save (commit) it, and there is no effect on the definitions in the development environment.

Definitions

A definition defines a single Smalltalk language element in a package, such as a method or a global variable.

All definitions must be unique within the following contexts:

- A package
- The development image in which it is loaded
In most cases, if you try to load a package that defines an item already defined in another loaded package, the package will not load. Instead, you are notified that an item definition conflict occurred so you can correct it. Handling options for such load conflicts are available using the Preferences Editor. For help in resolving conflicts, refer to appendix A.

**Kinds of Definitions**

The following language elements can be defined in a package:

- Classes
- Methods
- Instance variables
- Class variables
- Class instance variables
- Class initializers
- Global variables
- Pools
- Pool variables
- *Ad hoc* initializers

If you are already a Visual Smalltalk developer, most of these items will already be familiar to you. Each of these kinds of definition are discussed in detail later in this chapter.

**Initializers**

Visual Smalltalk Enterprise adds the concept of defining an initializer. An initializer is either a *class initializer*—an expression compiled in the scope of the class so it can access class pools and set the class variables and class instance variables—or an *ad hoc initializer*—an expression such as those you would evaluate in a workspace. In addition, global and pool variable definitions may include an initialization expression that sets the initial value of the variable. Initializers are evaluated when the package containing them is initialized, and can also be individually evaluated.

**Versions of Definitions**

The development environment uses your change log to keep track of the modification history for certain kinds of definitions, creating a new *version* every time you save a definition. Only definitions that have source code that is stored in the change log are versioned. The kinds of definitions that have versions are:

- Methods
- Class initializers
- Global variables
- Pool variables
- Ad hoc initializers

You can review previous versions with the History Browser. In the inadvertent case of an image crash, recovery information is stored in a recovery log file. For more information about the Recovery Log Workspace, see Appendix B.

Annotating Definitions

Definitions can be annotated, just as packages can. In this way, an individual programmer can keep track of the testing status or any other attribute of a definition for which an annotation has been defined.

Class-Related Definitions

A class definition stored in a package specifies the class name, its superclass, variables, and methods. Visual Smalltalk Enterprise allows you to add a class comment (which is optional, but highly recommended), and to organize the methods into categories. You can supply comments for each of the variables in the class to describe what they contain and how they are used.

These parts of a class are described below.

Methods

A method specifies a message selector, a comment within the method source (which is optional, but recommended), and a set of statements that are executed when an instance of the class receives a message matching the selector. Each method can also be assigned to a category.

A method need not be defined in the same package as its class. Visual Smalltalk Enterprise lets you define a method that belongs to a class defined in another package, or a class within the base class library. For example, the classes Bus and BusStop in the Bus Scheduling package could have methods to let them draw icons on a display for animation purposes. The icon-drawing methods could be defined in an Bus Animation package instead of the Bus Scheduling package.

Instance Variables

An instance variable is a variable that has independent storage for each instance of a class. Instances of the class share the name of the variable, but not its value. Instances of any subclasses inherit the name of the variable, but not its value, either. The value of an instance variable is typically initialized when an instance is created and can be different for each instance.
Instance methods of the class and its subclasses can refer to an instance variable. Class methods cannot.

For example, each instance of the class `Bus` has an assigned driver and an assigned route. Each instance of `Bus` therefore has instance variables `driver` and `route`. The value of these variables for one bus might be the strings 'Smith' and 'college park'. For another instance, the values might be the strings 'Jones' and 'pill hill'.

**Class Variables**

A class variable is a single variable whose name and value are shared by a class, its instances, its subclasses, and the instances of the subclasses. Whenever a class variable is initialized, no matter which class performs the initialization, its reinitialized value is available to the class, its subclasses, and all their instances. By convention, class variable names are usually capitalized.

Both class and instance methods of the class and its subclasses can refer to a class variable.

For example, to schedule buses, it is important to know how many people a bus can carry. Therefore, the class `Bus` defines the class variable `Capacity` to specify the critical information that a bus holds 40 people. When the class is initialized, the value of this variable is set to 40.

**Class Instance Variables**

A class instance variable is a variable with independent storage for every class. Subclasses inherit the name of the variable, but not its value. The value of a class instance variable is typically bound when the class is initialized.

Only class methods of the class and its subclasses can refer to a class instance variable.

Figure 1-3 below contrasts class instance variables with class variables, showing in each case what subclasses inherit and what methods can access.
citizens and the handicapped, the town also buys vans that can hold 12 people. The class Bus is now defined as the superclass of three new subclasses: StandardBus, ArticulatedBus, and Van.

The variable capacity is now a class instance variable of the class Bus. When the classes Bus and StandardBus initialize it, its value is set to 40. But ArticulatedBus initializes the value to 75, and Van to 12.

Pool Usage

A class definition in Visual Smalltalk Enterprise can specify that the class can access one or more pools. A pool is a group of variables that are collectively useful to a variety of classes, even though the classes are not in the same inheritance hierarchy. If a class has access to a pool, it can directly access any variable defined in that pool.

For example, the pool RGBColorConstants contains variables that define a set of colors according to red, green, and blue values. Classes in a graphics applications will very likely use this pool to draw icons or backgrounds on a display.

Class Initializers

A class initializer in Visual Smalltalk Enterprise is not the same as the initialize methods you may be familiar with from your Visual Smalltalk programming. A class initializer is a sequence of Smalltalk statements that executes when the package containing it is initialized. A class initializer might simply call a class initialize method, but it can also be more complex than that.

The initializer compiles within the scope of the class defining it, and therefore can refer to class variables or class instance variables. Indeed, it is a good way to initialize class variables, whose values are shared by its subclasses.

For example, an initializer can simply be:

```smalltalk
self initialize
```

A more complex initializer for the map in the Map Browser package might be:

```smalltalk
"initialize MapBrowser"

self initializeMap. "Read map from disk"
self initializeLocationMark. "Indicator of location"
self initializeReferenceMark. "Indicator of ref point"
RefPoint := (Longitude fromSeconds: 0) @
(Latitude fromSeconds: 0)
```
Global Variable Definitions

A package can define a global variable, which is then available to all classes within the development environment, including those defined in other packages. A global variable definition consists of a name plus an initialization expression that sets the value of the variable.

For example, a package might define a global variable `AllLetters`, a set that contains all uppercase and lowercase letters of the alphabet. The initialization expression for this global would be:

```
((A to: Z), (a to: z)) asSet
```

In Visual Smalltalk Enterprise, use initializers to define and set globals, to ensure that the globals exist and have appropriate values when their package is loaded.

*Do not* define globals in the image using `Smalltalk at: aKey put: aValue`, as is common in Smalltalk. Globals defined this way are not recorded in an initialization expression, and so are not recreated when the package is loaded.

Global variables can be designated as a constant during their initialization. Doing so prevents reassignment of the global’s value.

Pools and Pool Variable Definitions

A *pool* contains a group of related variables that are potentially useful to a wide variety of classes, regardless of whether they belong to the same inheritance hierarchy. The variables defined in the pool are available to any class whose definition specifies that it accesses the pool. Classes that access the pool need not be defined within the same package as the pool itself.

For example, the pool `ColorConstants` in the base library package contains pool variables that define colors such as those shown in Figure 1-4.
The classes in a **Printing** package might access this pool to print things in colored inks on paper.

Defining a pool consists of naming the pool and defining its variables. A pool variable definition consists of a name plus an initialization expression that assigns its value. Pool and pool variable names are capitalized and consist only of alphanumeric characters and underscores.

In Visual Smalltalk Enterprise, use initializers to define and set pools and pool variables, to ensure that theglobals exist and have appropriate values when their package is loaded. *Do not* define pools and pool variables in the image using expressions like *Smalltalk at:* `aKey put: aValue`, as is common in Smalltalk, since these definitions are not managed by the package structure.

Each pool variable can be designated as either constant or variable during their initialization. Pool variables that are designated as constant cannot have their values reassigned.

### Ad Hoc Initializer Definitions

In addition to these familiar items, a package can also include a definition for an arbitrary Visual Smalltalk expression whose purpose is to perform special initialization functions for an application. Such *ad hoc* initializers are not methods. They execute when the package is initialized.

For example, the expression **MapBrowser open** might be included towards the end of the **MapBrowser** package, to open the window for the user automatically, as shown in Figure 1-5.
Comments can be included with the expression in an ad hoc initializer. If you make the first line of a multi-line initializer a comment that describes the initializer, it is displayed in the definition list as shown in the figure above.

Do not use ad hoc initializers to declare global or pool variables. Globals should be described in a package by global or pool definitions. Using ad hoc initializers will cause conflict checking to be inaccurate.

Since ad hoc initializers are executed after all global references in the loaded package have been resolved, globals created by ad hoc initializers are not visible to the other definitions in the package when they are loaded. See the Loading a Package for more information on when ad hoc initializers are processed.

Clusters

Packages provide a single level of program decomposition. As you build larger programs, you will want to group related packages together by functional area, just as packages group related definitions. Clusters support the grouping of packages.

A cluster contains specifications, which identify the packages and clusters that are included in the cluster. Since a cluster’s specification can refer to other clusters, arbitrarily complex hierarchies of packages and clusters can be composed.

Clusters let you:
Key Concepts

• group dependent packages and clusters together, in order to record dependencies between them
• group packages and clusters so that they can be manipulated as a unit
• record combinations of packages and clusters that work together, creating new, larger, functional units
• record specific revisions of packages that work together
• specify packages and clusters to be substituted for one another under specific conditions
• specify the packages and clusters that are used to construct an SLL

A specific package or cluster can be included by more than one cluster. This lets existing clusters that contain overlapping specifications be combined into still larger clusters.

For example, the Idiom Tracking cluster, introduced earlier in this chapter, contains a specification for the Linguistics package. Another cluster containing a Dialect Analyzer can also refer to the same Linguistics package. If desired, both the Idiom Tracker and the Dialect Analyzer could be combined into a still larger cluster. The figure below schematically shows the overlapping nature of the specifications in the Idiom Tracking and Dialect Analysis clusters.

Figure 1-6: Schematic View of Clusters Containing Overlapping Specifications
Characteristics of Clusters

A cluster has several characteristics:

- **Collection of Specifications**
  The primary purpose of a cluster is to contain a collection of specifications that identify other packages and clusters. By combining clusters, you can organize and group your packages.

- **Name**
  Each cluster has a name, which is usually selected to describe its functionality. The name is a string of any length, and can contain any characters, except that leading and trailing spaces are removed. The name may not be the same as that of any other package or cluster loaded in the image.

- **Annotations**
  Annotations are a formal mechanism to communicate certain key features of a cluster. An annotation has a name and a value, both of which can be arbitrary strings. Like a package, a cluster can have any number of annotations, and each is defined to specify one key attribute. An annotation that begins with a period is read-only and cannot be changed after it is initially set.

- **Comment**
  A comment is an arbitrary string, and is useful for telling other programmers what they need to know about the cluster.

Specifications

A cluster identifies modules (packages and other clusters) with specifications. Each specification describes a module in terms of:

- repository name
- module type (package or cluster)
- module name
- revision number

A specification may be partial or complete, depending on how precisely the module is specified.

- A partial specification includes at least the module name (such as “the module named Map Browser”), and may include any of the other items. A minimal specification contains only the module name.
A complete specification includes all specification information (such as “revision 3.0 of the package named Cartography in the repository Tutorial”).

Figure 1-7: **Complete and Partial Module Specifications**

Figure 1-7 shows a cluster organizer with an open cluster selected that contains a complete specification for the **Cartography** package and a partial specification for the **Map Browser** package, specifying only its name and type.

All the module description fields in a specification are strings. The name is required, the other fields are optional.

Figure 1-8: **Entering a New Specification for a Cluster**

The repository field contains a repository name. The development environment maps a repository name to physical data on the disk.

The module type is a string that describes the kind of module. Modules are either of type “package” or “cluster.”
The module name is a string that contains the name of the package or cluster described by the specification. The name string can be any length and can contain spaces and punctuation. A name is required in all specifications.

The revision number describes the revision of a module located in a repository. The revision number must be in the form described in the Revision Control section later in this chapter.

Specifications also have other properties:

- conditional inclusion flags
- an optional inclusion flag, and,
- annotations

Conditional inclusion flags are compared against the development environment’s conditional inclusion context when the cluster is loaded. If all the flags match, the specification is included, and if any of the flags don’t match, the specification is ignored. This lets you vary the effective contents of a cluster by changing conditional flags on the affected specifications instead using multiple clusters, each containing a subset of the specifications. Aspects of your application that you would configure with conditional inclusion flags might include the host operating system, graphics ability, and optional features that are not included in the base application.

The optional inclusion flag indicates that the specification recommends a particular module without requiring it. For example, if an optional specification for

**Basic Functions: package rev 3.2 in NewRelease**

is loaded, it essentially means “if anyone asks for a package named Basic Functions, either now or in the future, give them revision 3.2 of the package from the NewRelease repository.”

Annotations are the same as package annotations. You can add arbitrary annotations to any specification.

Specifications can be created automatically by dragging loaded packages onto cluster, or by hand using the Specification Editor.
Configurations and Groups

Configurations and groups represent particular uses of clusters.

A cluster that contains only complete specifications is called a configuration. Configurations are useful, especially near the end of a development project, when it becomes necessary to exercise precise control over the exact revision of any given module is included in a release.

A configuration specifies the exact contents of its cluster, down to the revisions of the packages, so that the set of packages defined by the configuration is fixed. A configuration records a very specific relationship between revisions of modules.

In contrast, a cluster that contains only minimal specifications is called a group. Groups are useful for presenting logical groupings of modules independently of revision information. Depending on the task, different organizations of modules may make sense.

Since each minimal specification records only the name of a module, any revision of the module satisfies the specification. A group records a general relationship between modules.

Clusters make multiple organizations of modules possible, so that a module may occur in several clusters. When configurations are used, conflicts may occur between clusters if the constituent modules get out of synchronization. This is useful when you need to verify that a specific set of modules are included in your application. During development, however, you don't want these conflicts. Using groups instead makes it easy to avoid this kind of conflict.

There are tools available to create configurations and groups, and for converting clusters from groups to configurations and from configurations to groups. These are all described in chapter 4.

Cluster Operations

Clusters operations are similar to those for packages:

- **Open**
  Opening a cluster lets you view and edit the specifications it contains, without modifying the contents of the development environment.

- **Load**
  Loading a cluster processes the specifications it contains, identifies which ones should be included, and locates and loads packages or other clusters that satisfy the specifications.
• **Commit**
  Committing a cluster saves a revision of the cluster in a repository. Committing a cluster stores the specifications in the cluster, but does not commit any of the modules that the cluster references.

• **Migrate**
  You can migrate from any cluster to another revision of the cluster, to another cluster, or to a package. Migration is a quick way to replace the current cluster with another cluster or package.

• **Close**
  You close an open cluster to remove its specifications from the browsing tool where you were editing it.

• **Unload**
  You unload a loaded cluster. This removes the definitions for all packages it contains from the image if no other loaded clusters reference them.

### The Top-Level Cluster

The cluster named **top level** is predefined. This cluster contains specifications for all modules that are loaded in the development environment, and are not contained in another loaded cluster. Specifications for these modules are added and deleted as modules are loaded and unloaded.

![Figure 1-9: The Top Level Cluster Displayed in a Cluster Organizer](image)

The top level cluster does not appear in the Package Browser, although the modules that appear at the first level of indentation in the module list are those contained by the top level cluster. The top-level cluster does appear in the Cluster Organizer on the first line of the cluster list, as shown in figure 1-9. You can drag any module specification to the **top level** cluster to move it outside of any other clusters.
Resolving Specifications and Loading Clusters

Your development activities with clusters are centered around creating new clusters, adding specifications to clusters, editing those specifications, and committing and loading clusters.

You use the Cluster Organizer to create and modify clusters and their specifications. The Package Browser also has some capabilities for viewing and modifying clusters.

This section discusses what it means to load a cluster. The concepts are expanded on in the following sections.

A cluster is loaded in three phases:

1. In the inclusion phase, the specifications in the cluster and all included clusters that have conditional inclusion flags have their flags checked against those set in the environment. Specifications that have conditional inclusion flags that do not match those set in the environment are excluded from further consideration.

2. In the resolution phase, all the included specifications in the cluster and in all contained clusters are processed to determine a set of packages that matches all the specifications. This process takes into consideration the specifications and packages already loaded in the development environment.

3. In the load phase, the packages identified in the resolution phase that are not already loaded in the development environment are loaded.

Resolution can be thought of as the process of mapping a specification to a module that satisfies that specification. If the specification is for a cluster, resolution means to map each specification in the referenced cluster to a module, and so on, until all specifications have been resolved to packages.

Resolution only occurs when a cluster is loaded. Resolution determines which packages satisfy the specifications in the cluster.

When a cluster is loaded, the resolution process takes into account all currently loaded clusters and packages to determine which of the loaded modules satisfy specifications in the new cluster and which additional modules need to be loaded.

Unlike definitions in a package, having specifications in different clusters that resolve to the same package is not a conflict. This allows clusters to overlap.
Using Groups and Configurations

Using groups and configurations to identify your application and application components is a very powerful organizational tool. The following sections describe some ways to use them to solve organizational problems.

**Using Groups to Record Relationships Between Modules**

Groups record *structural* and *organizational* information about how modules can be combined to form larger functional units. In real applications, the modules that satisfy the partial specifications in a group may come from many different configurations, and groups can be used to combine the modules in separate configurations in new ways. Groups are used to record structural information because structure usually changes less frequently than configuration information does.

Another reason for using groups is separation of responsibilities. A group that describes the structure of an application contains partial specifications that name the modules but omit the revision numbers because revisions are not part of the application structure, but of a particular revision of the application.

Groups aid in understanding how the parts of an application fit together, since they contain only structural information about the application.

You can use groups to describe different views of an application. For example, one group might be composed of the delivery aspects of the application. Another group might contain development tools such as special browsers and debuggers. Yet another group might contain special application test drivers.

**Using Configurations to Record Revisions**

When deploying an application, you need some way to rigorously record the revisions of the modules in the application cluster in order to recreate the application.

This is accomplished with a configuration cluster that has all modules completely specified.

Depending on the size and complexity of the application, one configuration can be used to record all module revisions making up a specific revision of an application, or you can use a separate configuration for each subsystem in the application, and then combine these “sub-configurations” with another configuration that contains their specifications.
Combining Groups and Configurations

Since each specification in a cluster must resolve to exactly one module when a cluster is loaded, you may be wondering how a group can be loaded. The answer is that when any cluster is loaded, the modules already loaded and the specifications in other clusters being loaded at the same time are all taken into consideration during the resolution process.

Using the Idiom Tracking application as an example, let’s consider both a group and a configuration for this application.

Assume that the cluster named Idiom Tracking Group contains the following minimal specifications:

- A module named Cartography
- A module named Linguistics
- A module named Idiom UI

and the cluster named Idiom Tracking Configuration contains the following complete specifications:

- Revision 5.6 of the Cartography package in the Mapping repository.
- Revision 3.2 of the Linguistics package in the Language Analysis repository.
- Revision 1.1 of the Idiom UI package in the Idiom Tracking repository.

The minimal specifications in the Idiom Tracking Group will be satisfied by the same packages that satisfy the complete specifications in the Idiom Tracking Configuration. You can also see that many different Idiom Tracking configurations would satisfy the partial specifications in the Idiom Tracking Group.

There are at least five different ways you can ensure that the minimal specifications in the Idiom Tracking Group are successfully resolved when the cluster is loaded:

1. Add a complete specification for the Idiom Tracking Group to the Idiom Tracking Configuration so that the group is actually loaded by the configuration. Note that many different configurations might include the same group.
2. Load an Idiom Tracking Configuration before you load the Idiom Tracking Group. Since loading the configuration will load packages that satisfy both sets of specifications, the group can be loaded and all of its specifications will be resolved to match the modules already loaded.
3. Load the Idiom Tracking Group and an Idiom Tracking Configuration at the same time by specifying both of them
in another cluster and loading that cluster. Since all specifications in all included clusters are resolved to a set of modules to be loaded, the specifications in the group are resolved to the same modules specified by the complete specifications in the configuration.

4. Make all of the specifications in the **Idiom Tracking Configuration** optional. Then load the **Idiom Tracking Configuration**. No modules are loaded, but the optional specifications are remembered. Now load the **Idiom Tracking Group**. The specifications in the group are resolved against the optional specifications already loaded, causing the modules completely specified by the optional specifications to be loaded since they are now required by another cluster.

The important concept to remember is that a partial specification cannot be resolved by itself; it must be resolved in either the context of the modules already loaded in the development environment, or in combination with other, more complete specifications being loaded at the same time.

The difference between resolving a partial specification and a complete specification is that several different modules (or revisions of a module) may match a partial specification, but a complete specification must be satisfied by exactly one revision of a single module. When you combine the set of modules identified by each of the specifications, each specification must match exactly one module.

**Modifying Clusters and Specifications**

The Cluster Organizer helps you maintain clusters and specifications.

One useful operation is to convert clusters, groups, and configurations, using for example the items in the Module / Convert to menu in the Package Browser. Conversion operations are available in other menus and tools as well. The conversion is a quick way to maximize or minimize a cluster specification, preparing it for a new use.

For loaded clusters, you can make a partial specification be complete, or minimize a complete specification, discarding all information except the module name.

Of course, you can also edit individual specifications in any way you find suitable to your needs. Figure 1-10 shows the specification operations available in the Cluster Organizer.
Optional Specifications

If the optional inclusion flag is set in a specification, it creates a "weak reference" to a module. The module that the optional specification resolves to is not included when the cluster is loaded unless another cluster specifies a matching module.

This provides a mechanism to specify a preferred revision of a module, but not require its use. Optional specifications become part of the context in your development environment against which other specifications are resolved.

Using Optional Specifications with Configurations

Suppose you load a cluster named Latest Verified Revisions that completely specifies the latest verified revisions of all modules in a repository and has the optional inclusion flag set for each of the specifications. When you load this cluster, no modules are actually loaded. All that happens is that a "preference" for a module matching each optional specification is recorded.

If you later load the Idiom Tracking group that contains minimal specifications for the modules, the optional specifications from the Latest Verified Revisions cluster are used in the resolution of the group specifications to identify a set of packages that satisfies both sets of specifications, optional and not. After the set of packages that satisfy the specifications has been identified, they are loaded because they are required by the Idiom Tracking cluster.
Conflicts when Loading Optional Specifications

Optional specifications are resolved when they are loaded, and any resolution conflicts, such as two optional specifications that identify two different revisions of the same module, are detected and reported. The modules identified by optional specifications are not loaded until and unless another non-optional specification resolves to the optional module.

Conditional Specifications

You can use another property of specifications, the conditional inclusion flags, to automatically load different modules depending on conditions set in the development environment.

For example, suppose you are developing an application for two operating system platforms, say Windows and OS/2. Storing the application for both platforms in a single cluster, rather than in separate clusters for each platform, has the advantage that all related data is in one place and can be maintained and versioned together. Using the conditional inclusion feature, you can include both platform specific and platform independent modules in the single cluster.

To set up conditional inclusion, you would use the string “Win32” to Windows specific modules and “OS/2” to OS/2 specific modules. You would not specify a conditional inclusion flag for modules applicable to both platforms. Then, to load the platform independent and the Windows specific modules, you simply add the “Win32” string to the development environment’s inclusion context. When you load the application package, the Windows specific modules will load, but the OS/2 modules are left unloaded.

To specify an inclusion condition string for a cluster, you select the cluster in a Cluster Organizer and click the Add button in the bottom right of the organizer. A dialog allows you to enter the string, as shown in figure 1-11. Similarly, to specify the
development environment inclusion context, select Smalltalk/Open/Inclusion Context Editor and enter the context string in the dialog.

*Figure 1-11: Cluster Organizer, with context string editor*

**Specification Tracking**

Whenever the revision of the loaded module corresponding to a cluster specification is changed, either because the developer committed a new revision or migrated to a different revision, the cluster specification is updated to match the loaded revision. This change to the cluster causes the cluster to be marked as modified.

Tracking keeps specifications in a loaded cluster synchronized with the loaded modules. For example, when a loaded module is modified and committed, a new revision of the module is created and stored in the repository. In addition, the development environment is updated to reference the new revision, and the specifications in clusters that refer to the module are updated to
reference the new revision. If the specification for that module
includes the revision number field, the cluster itself is now marked
as modified, because the tracking mechanism modified its revision
number.

During tracking, only the parts of a specification that must be
updated for consistency are changed. For example if a cluster
contains a partial specification for

```plaintext
package 'Database Interface'
```

and the `Database Interface` package is migrated from revision 3.1
to revision 3.2, specification tracking will not change the partial
specification, since it is still correct.

If the specification had included the revision number,

```plaintext
revision 3.1 of package 'Database Interface',
```

the specification would have been automatically modified to be

```plaintext
revision 3.2 of package 'Database Interface'
```

and both the specification and the cluster containing it would
have been marked as modified. Commit the cluster to
permanently record the updated specification.

Specifications in open clusters are not affected by tracking.

---

**Revision Control**

The Visual Smalltalk Enterprise programming environment uses
revision control to let you keep track of your changes during
development, and of different revisions of your product after
release. Different revisions of packages and clusters are stored
within repositories.

Revisions of source code have long been a useful aid for
individual programmers who may need to remind themselves
what they’ve done before, or to revert to a previous revision if an
experiment doesn’t work. But revisions of packages and clusters
are a useful team programming aid as well. They let managers,
team leaders, or system integrators trace the history of a change,
and to determine accountability. Other members of the team can
consult previous revisions in order to determine who made the
change.
CHAPTER 1

Revisions

When you look at a module, you are looking at a revision of that module. When you initially load a revision of a module, your local image is updated to include the definitions in the module.

When you edit the definitions in a module, the local version in your image is changed. The changes remain local until you commit the module to a repository.

A committed revision of a module is the state of the module when it was last committed. Revisions can be committed in a simple linear hierarchy—for example, Revision 1.0, Revision 1.1, Revision 1.2, Revision 2.0, and so on. But Visual Smalltalk Enterprise also lets you commit revisions in a branching tree structure. For example, you may have a product with released Revisions 1.0, 1.1, and 1.2. You are presently working on Revision 1.3. However, you can fix a bug in Revision 1.2 and create Revision 1.2.0.1 while continuing your work on Revision 1.3.

A committed revision of a package or cluster can be thought of as a read-only module. Once committed, the code in a specific revision in the repository is frozen. You can modify it in your image and commit it with a new revision number, but there is no way to change the contents of a specific revision once it is committed.

You can examine and edit a revision without making it part of your development environment. If a revision is open, it is visible inside the window in which you organize definitions or specifications. You can look at it, but it is not executable. The source code is parsed, but not compiled into your image.

To execute source in the revision, you must load it into the application you are developing. When a revision is loaded, it is part of your executable development environment.

Package Revisions

When you change a definition in any way (including even a change to a comment), the package that includes it is now modified. Committing a modified package to a repository makes a new revision of the package.

Deleting a definition from a package also modifies that package. So does renaming it, adding an annotation to the package or any one of its definitions, or reordering its definitions. These rules are the same regardless of whether a revision is open or loaded.
You can compare two revisions of a package using the comparison tool provided.

**Cluster Revisions**

A cluster is considered modified whenever one of its specifications is changed, a specification is added or deleted, the annotations or comment for the cluster are changed, or the cluster is renamed. Committing a cluster to a repository makes a new revision of the cluster.

**Revision Number Restrictions**

Revision numbers must be composed of an even number of integer components, separated by periods. The revision numbers form a tree, indicating an informal derivation. Parent revisions are located higher in the tree. For example:

- Revision 0.1 is the parent of revision 0.2
- Revision 0.2 is the parent of revision 0.2.0.1

*Figure 1-12: Revision Tree*

A parent revision number must exist in order to create a branch revision number. For example:

- Revision 3.5 must exist before revision 3.5.0.1 can be created.
- Revision 3.5.0.1 must exist before revision 3.5.0.1.0.1 can be created.
Visual Smalltalk Enterprise lets the user specify revision numbers as long as they are “increasing” numbers. Revision numbers increase by adding number components, or by increasing the value of existing components. For example:

- 1.2 is bigger than 1.1
- 2.0 is bigger than 1.2
- 3.5.0.1 is bigger than 3.5
- 3.5.0.3 is bigger than 3.5.0.1
- 3.5.0.3.1.1 is bigger than 3.5.0.3

All branches sort after the parent revision.

Graphically, larger revision numbers are lower in the tree with the root of the tree at the top.

**Revision Attributes**

Revisions have the following attributes:

- **Revision number**
  When a new revision is made, it has an associated revision number. Depending upon the kind of repository you are using to store revisions, this number may be assigned automatically.

- **Author**
  New revisions automatically include the name or user ID of the developer who created them. This name is usually derived from your login name or specified by the VCSID value in your VCS.CFG file. See *Setting Your User Name* in chapter 2 for more information.

- **Time Stamp**
  New revisions are automatically stamped with the time and date of the commit operation.

- **Comment**
  When a new revision is created, the programmer is given the opportunity to associate a comment with the revision. Comments can be quite useful to communicate to other team members what has changed from the previous revision.
Repositories

Repositories provide permanent storage for revisions of packages and clusters. Repositories are identified by a name that identifies the repository and a file system directory that contains the data files used to store the repository information.

In order to share the data in the repository among team members, the directory containing the repository information is usually placed on a shared file system where it is available to all users on a network.

You can copy or back up all the information in a repository by copying or backing up the repository directory with standard operating system utilities.

Repository Operations

You will normally be using several different repositories. Some may be private, containing your work in progress. Others can be shared with team members. Still other repositories can be used for different projects.

To create new repositories and connect your development environment to existing repositories, use the following operations:

• You can create a new repository by supplying a name and the path to the file system directory that will contain the repository data.
• You can connect to an existing repository by supplying the path to the file system directory containing the repository data.
• You can reconnect to a repository that you were previously connected to, but the drive letter mapping or directory path changed.
• You can disconnect from a repository. The repository is not available to load or commit revisions until you connect to it again.

Repositories are only accessed to open, load, or commit revisions of modules. You can continue to use and modify loaded packages in your development environment even if the repository is temporarily inaccessible. See appendix A, Problem Solving, for more information on working with inaccessible repositories.
Kinds of Repositories

Visual Smalltalk Enterprise supports two different kinds of repository implementations: PVCS or the File-based repository. When you create a repository, you select one of these repository types. The advantages and disadvantages of each are discussed below.

**PVCS Repository**

PVCS, the Professional Version Control System, saves revisions in an efficient format that uses less disk space. All of the revisions of a package in a repository are stored in one file. It provides access control mechanisms to limit who can make a new revision of a package. It has mechanisms to deal with concurrent commitment.

**File-Based Repository**

Using the file-based repository, no access control mechanisms are available to limit who can make a new revision of a package. Superceded revisions in a file-based repository use more disk space than in a PVCS repository, since old revisions are stored in their entirety instead of as a set of differences. File based repositories are faster, since there is no encoding used to reduce disk space.

**Access Control**

Repositories are subject to two different kinds of access control. One kind of support is provided by the directory system and network software. These systems provide limited read and write on access control to directories and files in shared directories. You must have read access to repository directories and files to load the files, and write access to commit modules.

The second kind of access control is provided for PVCS repositories, and gives much more extensive access control. Access control restricts access to repositories and repository entities by assigning access rights to individuals and to groups. See *Controlling Repository Access* in chapter 5 for more information.
Unresolved References

Visual Smalltalk Enterprise allows you to have global names in the system that are not bound to specific objects. These temporarily undefined names would ordinarily produce a compiler error. This feature permits a more flexible coding style. It also permits forward references, giving you flexibility in the order in which you load your packages.

When you are defining code in the Package Browser, the compiler responds to an unbound name by giving you the option of declaring it as a global variable or an unresolved reference. When you load a package with an unbound name in it, the system automatically treats the name as an unresolved reference.

Unresolved references do not belong to any package. Therefore, when you commit a package, you are not saving any definition for the unresolved references. For this reason, it is a good idea to check for and resolve unresolved references before you commit your packages or generate a build script for your application. It is also a good idea to check for unresolved references after you load an unfamiliar package.

To determine what unresolved references exist in your image, choose Smalltalk / Browse / Unresolved. If any unresolved variables exist in your image, an inspector appears. Otherwise, a message appears saying “No unresolved references.” For further information, see the section entitled Loading and Migrating Modules in chapter 4.
CHAPTER 2

Configuring Your Working Environment

Overview

Visual Smalltalk Enterprise works with (but does not include) Intersolv’s PVCS Version Control Manager, which provides capabilities for controlling source code versions during development, as well as for controlling access to your source files. If you already own and will be using PVCS with Visual Smalltalk Enterprise, then follow the procedures in this chapter to configure your environment.

Another level of configuration is provided by the Preference Editor. This provides an easy method for modifying a variety of features, from color selections to package conflict resolution settings. These settings can be changed at will during development, so you do not have to set them before you begin to work.

Enabling PVCS Support

If you already have PVCS installed on your computer, Visual Smalltalk Enterprise can be configured to bind to the PVCS DLL file (VMWFDTK.DLL). This enables support when starting up the first time, as long as the DLL is on your system PATH.

Since different versions of PVCS set up different directory structures, you’ll need to find the location of VMWFDTK.DLL and verify that the directory that contains it is in your PATH settings. Note that in PVCS versions 6.0 and 6.6, the correct directory is added to the path during installation.

If you purchase and install PVCS after you have already installed and begun using Visual Smalltalk Enterprise, there are a few steps needed to enable PVCS support.

1 Open up a Repository Browser, by selecting the menu item Smalltalk / Open / Repository Browser.

2 In the resulting window, choose the menu item Repository / Reset Repository Types.
This causes VSE to look for and bind the PVCS DLL. If it is successful, you will be able to create, connect to, and use PVCS based Repositories. As explained above, the DLL must be on your system PATH in order to be found.

### Setting Your User Name

Visual Smalltalk Enterprise determines your user name when you start the development image. As first installed, it determines your user name differently, depending on the operating system platform:

- **On Windows NT**, your login name is used as your user ID.
- **On other Windows platforms and OS/2**, your user ID is read from the VCS.CFG file in the current directory.

The user ID is used in the author and version annotations that are attached to cluster, package, class, method, and other definitions at the time they are created. This is the user name shown in the version button at the bottom of many browsers. For example, when a method is selected, the version button might show that the method was created on:

**09/04/94 11:43:03 AM by Michael**

When you change your user ID, new definitions that you create reflect your user ID. Existing definitions, however, retain their original author.

On Windows or OS/2, set your user ID by setting VCSID environment variable in your local VCS.CFG file, located in the Visual Smalltalk root directory. Open the file using any ASCII text editor or a Visual Smalltalk workspace. Originally, the VCSID line says:

```
VCSID=editThisName
```

Edit this line, replacing “editThisName” with your desired user ID, such as:

```
VCSID=Michael
```

Save the file, then restart Visual Smalltalk Enterprise.

**NOTE:** If you edit VCS.CFG while Visual Smalltalk is running, you must exit and restart the image for the new values to take effect.

On Windows NT, user names are controlled by the master configuration file, as described below.
PVCS Administration

This section describes the configuration options affecting team development. The previous section described setting your user ID. The configuration options described here affect your user ID and other options.

There are two main configuration files: the local VCS.CFG file already introduced, and the master configuration file. The following sections describe how to set up a master configuration file, PVCS capabilities that conflict with Visual Smalltalk Enterprise, and how to control access to repositories with PVCS.
Setting up a Master Configuration File

The Master Configuration File is a configuration file that Visual Smalltalk Enterprise reads before any other configuration file, such as VCS.CFG. The master file is used to ensure that all users share the same default settings for certain PVCS parameters. Among other things, the master configuration file can override certain parameters set in other configuration files, preventing users from resetting them.

Identifying the Master Configuration File

In order to access the master configuration file when starting Visual Smalltalk Enterprise, you must set the file location. The master file is usually located on a shared network drive.

The location of the master configuration file must be set in a local PVCS file by sending the following messages to RepositoryServices. In a Visual Smalltalk Enterprise session, evaluate:

```smalltalk
RepositoryServices current
embedMasterConfiguration: 'master_file_location'
```

providing the drive, directory path, and file name of the master configuration file.

The local Visual Smalltalk Enterprise administrator should perform the above step and save the image. This image can then be distributed to other local users.

The PVCS administrator can then control configurations by editing the specified shared file.

Master Configuration File Directives

The most important setting contained in the master configuration file is set using the LOGIN directive, which allows the administrator to control the source of the user ID. For example, you might override the default login source to use the Netware login name, if it is available. No change to this file is necessary to use the Host default on Windows NT.

The syntax is:

```smalltalk
LOGIN source[, source...]
```

Values for `source` are:
The Version Manager attempts to assign the user ID from the sources in the order listed, from left to right. The default list is **VCSID**, **UNKNOWN** for Windows and OS/2, and **HOST** for Windows NT.

If **UNKNOWN** is used, it should be placed last on the list.

**VCSID** is *not a secure source*, because it can be changed in the local configuration file. For maximum security, use the master configuration file to set the source of user identification and disallow users from changing it. For example, in the master configuration file, put:

```
LOGIN NETWARE, VCSID
DISALLOW LOGIN
```

This directive tells the PVCS Version Manager first to try to find the user identification from Novell NetWare, then from the local VCS.CFG file. The **LOGIN** directive is then disallowed so users cannot change the source of user identification.

Other directives can also be included in the master configuration file.

Intersolv recommends that you disallow the following directives in a VCS.CFG:

- **Semaphore**
  - This directive controls how Version Manager uses semaphores.

- **SemaphoreDir**
  - This directive specifies the directory for semaphore files.

To disallow an options, use the syntax:

```
Disallow directive
```

<table>
<thead>
<tr>
<th>HOST</th>
<th>Host operating system. Use this source with systems that provide a user identification mechanism, such as Windows NT.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LANMAN</td>
<td>Microsoft LAN Manager for DOS and OS/2</td>
</tr>
<tr>
<td>NETWARE</td>
<td>Novell NetWare for DOS and OS/2</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>No source; if no user ID is found, <strong>UNKNOWN</strong> is used.</td>
</tr>
<tr>
<td>VCSID</td>
<td>Default user ID for Windows and OS/2, should be placed first in the list to read user ID from local VCS.CFG file.</td>
</tr>
<tr>
<td>VLOGIN</td>
<td>Don’t use. This entry conflicts with Visual Smalltalk Enterprise</td>
</tr>
</tbody>
</table>
It is important that all users have the same settings for these directives. For instance, if your environment allows each user to customize the semaphore mechanism, different mechanisms could be used to control updates to an archive. The result would be simultaneous uncontrolled updates to the archive file.

**Setting Up Local Configuration Files**

The configuration file, VCS.CFG, is a text file containing PVCS configuration directives and parameters. You can edit this file with any text editor, including a Smalltalk workspace.

The local configuration file is only required on Windows and OS/2 platforms that take the user ID from the VSCID line in the local file. Windows NT, for example, uses the master configuration file directives.

Visual Smalltalk Enterprise installs a default VCS.CFG file into the root installation directory with defaults appropriate for your platform. If you use more than one working directory, you should copy the file VCS.CFG to each directory.

On Windows and OS/2, the default value of the LogIn directive references the value of the environment variable VCSID for user identification. The initial default VCS.CFG file contains the user name “editThisName” as the parameter for the VCSID directive:

If you do not change the user name in the VCS.CFG file, the default name “editThisName” is attached to your revisions. If you see “unknown,” it means that PVCS was not running at the time the version was made or that VCS.CFG wasn’t found.

**PVCS Capabilities that Conflict with Visual Smalltalk Enterprise**

If you are familiar with additional PVCS capabilities, or have use of the full PVCS package, note that the following PVCS capabilities are incompatible with Visual Smalltalk Enterprise, and should not be used:

- ExpandKeyword directive
- Promote directive and the vpromote command
- vmrg command
- BranchWarn directive
- ArchiveSuffix directive
- Semaphore directive

Multiple semaphore mechanisms for the same archive break the controlled update of archives for PVCS.
Configuring the Access Database

Access control provided with Visual Smalltalk Enterprise allows you to control access to repositories and repository entities, by assigning access rights to users and groups of users. By assigning access rights you can control, for example, who can commit a new revision of a given module. Users without access permission can still save their work locally, but not to the controlled repository.

Creating an Initial Access Database

Access control requires an access database. If the database has not been defined, you can create one from a text file.

To create an access database, create a text file with at least one user entry. User entries are in this form:

user Bruce/Bruce ()

Only one is necessary, since additional user and group definitions will be added using the Access Database Editor. The file should include your user name, however, so you can edit the database.

Next, build the Access Database by evaluating this expression in a workspace:

```smalltalk
RepositoryServices current
createAccessDatabase: 's:\reposadm\access.db'
fromTextFile: 's:\reposadm\access.txt'.
```

The access database file should be located on a shared drive that is readable by all users but writable only by the administrator.

The location of the access database file must be recorded in the PVCS files. To do so, evaluate this expression in Visual Smalltalk Enterprise:

```smalltalk
RepositoryServices current
 embedAccessDatabase: 's:\reposadm\access.db'
```

The Visual Smalltalk Enterprise administrator can evaluate this expression and save the image, and then distribute the image to other users.

Editing the Access Database

The access database records users and groups. Once the database is defined and embedded, it can be edited using the Access Database Editor.
To open the Access Database Editor, select Smalltalk / Open / Access Database Editor in any Visual Smalltalk Enterprise menu. The editor opens displaying lists of users and groups. Initially, it will only display entries included in the text file.

**Figure 2-1: Edit Users Dialog**

To add a *user* to the Users list, select *User / New...*. A dialog opens allowing you to enter a new user name.

To add a *group* to the Groups list, select *Group / New...*. A dialog opens allowing you to enter a new group name.

To add users to a group, highlight the target group and select *Group / Edit*, or double-click on the group name. The following dialog opens:

**Figure 2-2: Edit Group Dialog**
Select a user or group name in the Non-members list, then click the Add button. If you include a group in another group, all members of the original group will share access rights granted to the target group.

Similarly, you can delete a user or group from the group by selecting the user or group ID in the Members list and clicking the Delete button.

**Assigning Access Rights**

Access rights are assigned to users and groups on an individual basis for repositories and repository entities. Refer to *Controlling Repository Access* in chapter 5 for more information.

**Setting Environment Preferences**

There are a variety of settings you can change to control your Visual Smalltalk Enterprise working environment. Some, such as changing various color settings, can add to your comfort while working. Others, such as conflict resolution settings, you may need to change to better handle specific working situations or problems.

These settings are controlled by the Preferences Editor. There are actually several separate editors, which you access from a central dialog.

To open the Preferences Editor, select Options / Preferences in any Visual Smalltalk Enterprise window. This opens the Preferences Editor shown below, and lets you specify the preference group you want to modify.
Preferences have been organized into groups, as shown in the left pane above. The right pane is the preference group editor, which allows you to set options appropriate for the selected group. Each group editor is described in the following sections.

### Category Order Preference Editor

When you select *Category Order* in the Preferences Editor, the preference group editor contains a list used for ordering categories in the Package Browser. Click on the *Add* button to add a new entry in the list, or *Delete* to delete the selected entry. The *Up* and *Down* buttons can be used to reorder the selected entry.

Wildcards can be used. See the defaults and examples below for further information.

The default value for Method Category Order of "*, subclass*, private*" indicates that all categories are sorted together, and appear in strict alphabetical order, except those that begin with "subclass" or "private," which are placed at the end of the list.

Another method category order could be

- accessing
- *
- test*
- init*
- private*
In this case the categories appear in the following order:
1. `accessing` always appears first, followed by
2. most other categories in alphabetical order, followed by
3. all categories that begin with the string `test`, followed by
4. all categories that begin with the string `init`, and finally
5. all categories that begin with the string `private`.

The wildcard character * stands for any number of other characters, including none.

**Color Preference Editor**

The Color Preference Editor, shown in Figure 2-4, lets you specify the colors used for the following in tools:
- modified definitions (blue, by default)
- highlighted text for selections (defaults to system highlight)
- highlighted background color of selected list items in a browser (defaults to system background highlight)
- comparison differences (red, by default) used to highlight differences in the definition list and contents panes of the comparison tools.
- new (green, by default) used to highlight items in one definition list of the comparison browsers that have no counterparts in the other definition list.

Figure 2-4: Color Preference Group Editor
Configuring Your Working Environment

Click on any of the colored buttons to open a Color dialog to select a different color. The color of each button reflects the current color setting for the specified item.

Values for the highlighted text and highlighted background color reflect your current settings established for your system default desktop colors. The highlight colors change as you change your system desktop highlight colors unless you select a highlight color with the Color Preference Editor. Reset the colors back to their defaults with the Options / Defaults menu choice to assume the system desktop highlight colors if you change your mind.

When you change the color settings, you may need to update other currently open tools to reflect the current color settings.

Compilation Preferences Editor

The Compilation Preferences editor allows you to set a few compiler options related to support for the new ANSI Smalltalk standards introduced into Visual Smalltalk 3.1. Based on the option settings, the compiler handles certain expressions differently, either using the new ANSI syntax or the syntax and semantics of earlier versions of Visual Smalltalk.

Figure 2-5: Compilation Preferences Editor
The options are presented in two blocks.

**Language Syntax** (only one can be selected):

- **Backward Compatible.** Use Visual Smalltalk 3.0 syntax.
- **Strict Standard.** Strictly adhere to the draft X3J20 ANSI standard syntax.
- **Extended Standard.** Support ANSI syntax, but include extended features of the Cincom Smalltalk language and backward compatibility with Visual Smalltalk 3.0.

Visual Smalltalk syntax is described in the Visual Smalltalk Enterprise Language Reference.

**Compiler Warnings**:

- **Warn Obsolete.** Issues a warning of obsolete usages. For example, a warning is issued if you reference a block argument outside the block when it was not declared as a temporary variable for the method.

- **Warn Questionable.** Issues a warning if a block argument or temporary usage is questionable. For example, using a block argument with the same name as a method temporary variable is a questionable, but permitted, usage.

**WARNING:** The backward compatibility option is supported in Visual Smalltalk and Visual Smalltalk Enterprise release 3.1 only. Future releases will not support this option.

**Conflict Preference Editor**

The Conflict Preference group editor allows you to specify the way conflict checking is done when loading, unloading, and migrating revisions of modules.

If “Base Image” is not checked, you are notified of any conflicts that are detected between the definitions in a module and the original base image, when you are loading, deleting, or migrating from one revision to another. If it is checked, the definition being loaded overwrites the definition in the base image.

If “Existing Instances” is not checked, removing or migrating a module checks to see if a class you are removing has instances present in the image. It also checks that the revision to which you are migrating defines the class for currently existing instances. If it is checked, the instances remain in your image and become instances of DeletedClass.
See Appendix A for further details on these conflicts and how to resolve them.

**Lists Preferences Editor**

The List Preferences control how list panes are displayed. Currently the only preference option determines whether horizontal scroll bars are displayed.

*Figure 2-7: Lists Preferences Editor*
Repository Preference Editor

The Repository Preference Editor lets you set the default repository type to be either File-Based or PVCS. When you create a new repository, you can choose either type. This preference specifies the default value, and is initially set to “PVCS.”

Figure 2-8: Repository Preferences Editor

Saving and Loading Preference Settings

The current preferences settings are saved with the current image whenever you save your Visual Smalltalk image. If you do not typically save the image, but still want to use modified preference settings, you can save your settings to a file and reload it during a session.

Preference files are simply text files. There is no restriction on file naming or location, though the default file name extension is “.PRF”.

The Preferences editor Preferences menu contains three items:

- **Load...** allows you to select a previously defined preferences file, and load the settings from it.
- **Save...** allows you to save the current preferences settings to a file.
- **Defaults** resets the preference items in the current preference group to their default values. If you want to reset all preference values, select each group in turn and issue this command.
CHAPTER 2  Configuring Your Working Environment

Installing Optional Tool Support

A few extensions can be added to Visual Smalltalk Enterprise tools by including additional binding instructions in the development system bind file.

The development system bind file is named VDEVp.BND, where p is “W” on Windows systems, or “O” on OS/2 systems. This bind file and the other bind files referred to in this section are all located in the Visual Smalltalk Enterprise root directory.

Bind files are plain text files, and can be edited using any standard editor, or by opening them in a workspace.

Tool Source Code

Source code for the Visual Smalltalk Enterprise tools is not visible in the standard configuration. You can, however, install the source for the tools by modifying the development system bind file. This is particularly useful if you are using the Team Tool API to create your own tools, and want to use our tools as examples.

Note that you cannot convert an image built with the standard configuration to an image with the visible tool configuration, or vice versa. To change configurations, you must start with a clean V.EXE image file.

The standard configuration is installed by referencing the TEAMVSTD.BND in the main bind file, VDEVp.BND:

@TEAMVSTD.BND

To configure Visual Smalltalk Enterprise with tool source visible:

1. Back up your current image file and its log files: V.EXE, CHANGE.LOG, and RECOVER.LOG.
2. Copy a clean V.EXE image file into your working directory.
   The default installation copies a clean V.EXE into the VBACKUP directory under the Visual Smalltalk root directory.
3. Edit VDEVp.BND and replace TEAMVSTD.BND with TEAMVTOL.BND:
   @TEAMVTOL.BND
   Only one of these bind files may be referenced at a time.
4 Start Visual Smalltalk Enterprise.

The configuration for the visible tool source will automatically bind.

You can return the system to the standard configuration by repeating this procedure, but replacing TEAMVTOL.BND with TEAMVSTD.BND in VDEVp.BND.

After changing the tool set, you can restore your development environment by following the instructions in Rebuilding an Image in chapter 5.
In this tutorial you will create a simple application, the Map Browser. The Map Browser displays a map of the world and lets you click anywhere on the map. The latitude and longitude of the point on which you clicked is displayed in two fields below the map. Alternatively, you can type a latitude and longitude into those fields, and the corresponding point is displayed on the map.

The Map Browser appears as shown in figure 3-1.

Figure 3-1: Map Browser

The Map Browser configuration information is stored in the Map Browser Configuration cluster in the Team/V Tutorial repository. This cluster completely specifies the two packages that contain the actual application code—the Map Browser package and the Cartography package.

In this tutorial, you will perform these tasks:

1. Connect to the tutorial repository.
2. Load a cluster from the tutorial repository.
3. View the specifications in a cluster.
4. Modify one of the packages making up the application.
Chapter 3

5 Create a new revision of a package.
6 Update the cluster to reflect the new configuration and create a new revision of it.
7 Compare the new revision of a package with the original.
8 Split the contents of one package into two packages.
9 Add a new package to the cluster.
10 Distribute your changes.

Chapter 5, *Team Programming Tasks*, also provides step-by-step procedures, but for a wider variety of smaller, generic tasks you need to perform while using Visual Smalltalk Enterprise.

### Connecting to the Tutorial Repository

The clusters and packages that make up the Map Browser application (the **Map Browser Configuration** cluster and the **Cartography** and **Map Browser** packages) are stored in the tutorial repository. A repository is stored in a file system directory. You connect to a repository by the directory the repository is stored in.

**To connect to the tutorial repository:**

1 Choose the **Browse Packages** menu item from the **Smalltalk** menu on the Visual Smalltalk Transcript window to open a Package Browser.

Your screen appears as shown in figure 3-2.
2 In the Package Browser, choose Smalltalk / Open / Repository Browser.

The Repository Browser opens.

3 Select Repository / Connect… to open the Connect Repository dialog.

Figure 3-3: The Repository Browser with Connect Dialog
4 Use the directory browser in the Connect Repository dialog to select the directory containing the tutorial repository.

The tutorial repository is in the TUTORIAL\TEAMV\TMVTUTRL subdirectory of the Visual Smalltalk Enterprise root directory.

5 After selecting the file system directory containing the repository path, click on the OK button.

A progress indicator opens showing you that the repository information in the directory is being read and verified.

**NOTE:** You will get an error if you select a file system directory that does not contain a valid repository. Press OK to dismiss the dialog and try again.

After successfully connecting, the tutorial repository is added to the list of connected repositories in the Repositories Browser.

The tutorial repository is now connected and available for use.

6 Close the Repository Browser by double clicking on the system menu icon in the upper left hand corner.

### Loading the Tutorial Cluster

You next load the **Map Browser Configuration** cluster in the Tutorial Repository into your development environment. Loading the cluster loads the two packages referenced by the cluster. After loading, you can browse the code that has been loaded.

**To load the Map Browser Configuration cluster:**

1 In the Package Browser, choose *Module / Load*.

The Load dialog that appears lets you select the repository in which the package resides, and then to select the cluster and revision to load. Note that the *Load* button on the dialog is disabled until these selections have been made.

2 Select *Team/V Tutorial* in the Repositories pane of the Load dialog.

The clusters and packages in the repository appear in the Modules list, as shown in figure 3-4.
3 Select each of the names listed in the Modules list in turn.

Note that as each is selected, information about it is shown in the text pane at the bottom of the load dialog, including whether it is a cluster or package.

Also note that a list of available revisions for each entity appears in the Revisions list.

4 Select the cluster named Map Browser Configuration.

A list of revisions appears in the Revisions pane.

5 Select the revision of the cluster to load.

When a revision is selected, the Load button is enabled. Your screen now looks like figure 3-5.
Click **Load** to load the cluster.

Visual Smalltalk Enterprise loads the cluster (and the packages it specifies) for you, and after several seconds, it appears selected in the Package Browser, as shown in figure 3-6.

![Package Browser Open on the Map Browser Configuration Cluster](image-url)
7 Take a minute to browse the code in the cluster you just loaded.

View the class definitions it contains in the second pane from the left—the global list. Select the class names and view their comments, definitions, initializations, descriptions, and hierarchies, if you want, by selecting the appropriate item from the Global menu.

Select the categories displayed in the third pane from the left, and then view their methods in the rightmost pane. Select a method to view its source.

8 Click on the expansion indicator with the plus sign to the left of the **Map Browser Configuration**.

Click on an expansion indicator to open it a single level. Double-click to fully open all levels. The packages and clusters contained in the **Map Browser Configuration** cluster are displayed indented below it. You can now see that the **Map Browser Configuration** cluster contains both the **Map Browser** and **Cartography** packages, as shown in figure 3-7.

**Figure 3-7: Showing the Packages in the Map Browser Configuration Cluster**

When a cluster is selected, all the classes in all the packages in that cluster are displayed in the globals list.

Select the **Cartography** and **Map Browser** packages in turn and note which classes are defined in each package. You can also review the comment and revision information for each package.
If you want to hide the packages in the **Map Browser Configuration** cluster, click on the minus sign to the left of the cluster name.

### Viewing the Specifications in a Cluster

You can examine the specifications contained in the **Map Browser Configuration** cluster if you’re curious about what the cluster actually contains.

**To view the specifications in a cluster:**

1. Choose _Smalltalk / Open -> Cluster Organizer_ in the Package Browser.

   The Cluster Organizer opens.

2. Select **Map Browser Configuration** from the cluster list in the Cluster Organizer.

   The specifications for the **Cartography** and **Map Browser** packages are shown in the upper right-hand pane of the Cluster organizer. Annotations and other information about the cluster is shown in the lower panes, as shown in figure 3-8.

---

*Figure 3-8: Viewing the Specifications in a Cluster*
You can browse some of the other clusters in Visual Smalltalk Enterprise for more interesting examples of how clusters are used to organize packages and other clusters.

3 Double-click on the system menu icon to close the Cluster Organizer.

Modifying a Package

Much of your work in the Visual Smalltalk Enterprise development environment consists of modifying packages that exist already, by adding new definitions and changing existing ones. In this part of the tutorial, you will perform these tasks:

- Add a new method category and two new methods to a class.
- Change two methods in another class.
- Look at previous method versions.
- Test the Map Browser.

Adding Category and Methods

1 Select the package named Cartography in the Package Browser.

If the Cartography package is not visible in the module list, expand the Map Browser Configuration cluster by double-clicking on the plus sign to its left.

2 If you have not already done so, take a minute to browse the code in the Cartography package.

It consists of three classes: the superclass CartographicUnit, and its two subclasses Latitude and Longitude.

Notice that Longitude has method categories for accessing and for testing among others. Browse these categories, and you'll notice that “accessing” includes the methods east and west, and “testing” includes the methods isEast and isWest.

Latitude includes the category “accessing,” which contains the methods north and south. However, it lacks the category “testing” and the methods isNorth and isSouth. Latitude needs to be queried as well, so let's create the category and the methods.

3 Select the class Latitude, if it is not already selected. Make sure the instance radio button is selected.
4 Choose Method / New Category...

A window pops up, with the prompt *Create category named*:

5 Type the name "testing" in the field. Then, click OK or press Enter.

The new category appears selected in the category list—the second pane from the right. Although it is new, it is not yet italicized (which would indicate that it had been changed) because it contains no new methods yet. Because it belongs to the selected package (*Cartography*), it is boldfaced.

The new method template appears in the contents pane, as shown in figure 3-9.

Figure 3-9: *Adding a New Method Category to Class Latitude*

6 Now let's browse the code to find out how to determine if a point is in the northern or the southern hemisphere.

How is the data stored? Browse the code inherited from the superclass *CartographicUnit*.

The inheritance box above the method list lets you examine code inherited from all superclasses.
7 Click the drop-down button to the right of the inheritance box to cause a list of superclasses to appear. Select CartographicUnit, and all of its code is available to you in the category and method lists.

You will find the methods **positive** and **negative** in the category “sign.” The method **positive** uses the instance variable `sign`, which you discover is a Boolean that captures the information about the hemisphere where a point resides. This seems to be what is needed.

8 Select the category testing in the class Latitude again.

9 Select the method template and replace it with the following text:

```smalltalk
isNorth
    "Return true if the receiver is in the northern hemisphere."
^self positive
```

10 Choose File / Save to save your text.

The new method appears in the method list—the rightmost pane. The Package Browser now appears as shown in figure 3-10.

![Figure 3-10: Cartography Package after Adding a New Method](image)

Because the method is new, its name is italicized. Because it belongs to the selected package (Cartography), it is boldfaced. The package, class, and category names in the three leftmost panes now appear italicized also, because...
they have been modified. The cluster name is italicized and blue, because a package it specifies has been modified.

11 To redisplay the method template in the contents pane, click on the selected category in the category pane. The method is deselected, and the method template appears below.

Clicking on a selected item in any list pane of the Package Browser or Definition Organizer will deselect the selected items in the list panes to the right.

12 Select the method template and replace it with the following text (or use the isNorth method as a template):

```
isSouth
   "Return true if the receiver is in the southern hemisphere."

^self negative
```

13 Choose File / Save to save your text.

The new method appears, italicized and boldfaced, in the method list.

### Changing Methods

While you were browsing the methods in class **CartographicUnit**, you may have noticed that in the category **sign**, the two message selectors **negate** and **negated** are quite similar. Because the methods are uncommented, it isn't easy to determine what makes them different. After examining each, you determine that the method **negate** changes the receiver, while the method **negated** changes a copy of the receiver. This difference is subtle. Let's add comments so that others need not be puzzled.

1 In the globals pane, select the class **CartographicUnit**.

2 Select the category **sign**.

3 Select the method **negate**.

4 In the contents pane, add the following comment on a line after the message selector:

```
"Change the value of the receiver so that it is in the opposite hemisphere."
```

5 Choose File / Save to save your text.

Note that if you selected **Latitude** as the class rather than **CartographicalUnit**, a dialog opens listing both classes.
You can select **CartographicalUnit** to update the correct method.

6 Select the method `negated`.

7 In the contents pane below, add the following comment on a line after the message selector:

"Return a new instance in the opposite hemisphere in a position analogous to that of the receiver."

8 Save your text.

**Working with Previous Method Versions**

You have just created new versions of the `negate` and `negated` methods. You can view the previous version of a method you have changed.

**To view the previous version of a method:**

1 Select the `negated` method in the Package Browser.

2 Press the version button at the bottom right of the browser.

A History Browser opens, listing the versions of the `negated` method in class CartographicUnit, as shown in figure 3-11.

---

![Figure 3-11: History Browser on Versions of a Method](image)

---

Visual Smalltalk Enterprise Tools Reference 85
You can view the previous version (labelled “Map Demo 1.0”) of the negated method by selecting it in the top list of the History Browser.

You can also revert to the previous version of the method if you made a mistake changing it.

**To restore a previous version of the negated method:**

1. Select the version “Map Demo 1.0” to be restored in the History Browser.
2. Choose Version / Install.

Three versions of the method now appear in the history browser. Effectively, the oldest version was copied to make it be the newest. Since both the oldest and newest versions are labelled “Map Demo 1.0,” you know they are the same method.

If you forget what changes you made to the negated method, you can compare your version with the original version.

**To compare versions of a method:**

1. Choose Version / Compare All in the History Browser.

   The Version Comparison tool opens, as shown in figure 3-12.

---

**Figure 3-12: Comparing Versions of the negated Method**
The versions of the method appear in the top lists on both sides of the comparison tool.

2. **Select the original version on the left and the version you created on the right.**

The differences (the comment you added) are shown between the two methods. Added text is underlined and in red. To change the colors used to highlight differences, use the Color Preferences Editor.

3. **Close the Version Comparison tool.**

4. In the History Browser, select the version of the **negated** you created in this tutorial.

That’s the version with the new comment.

5. **Choose Version / Install to reinstall your new method.**

Note that four versions are now listed in the History Browser, even though there are only two distinct versions of the method.

6. Close the History Browser.

### Testing the Map Browser

Before you make a new revision of the **Cartography** package to record your changes, you should test the **Map Browser** to make sure it still works.

**To test the Map Browser:**

1. In the Package Browser, select in order: the **Map Browser** package, the **MapBrowser** class, the class “opening” category, and the class method **open**.

2. Select the expression **MapBrowser open** in the method comment (do not select the quotation marks).

3. **Choose Smalltalk / Do It.**

The Map Browser window should open, as shown in figure 3-1 at the beginning of this chapter.
Creating a New Revision of a Package

You are now ready to create a new revision of the Cartography package that reflects your changes.

To commit the package and create a new revision:

1. In the Package Browser, select the Cartography package, if it is not already selected.

2. Choose Module / Commit....

The Commit dialog appears, showing you the name, the revision number the current version was derived from, and the proposed revision number of the selected package. In this case, it shows Rev. 3.2 of the Cartography package.

3. In the Comment box, type:

   Added testing methods to Latitude. Added comments to CartographicUnit “sign” methods.

4. Make sure the Revision radio button is selected to increment the revision number.

   The Revision radio button increments the minor revision number. For example, Revision 3.0 becomes 3.1, Revision 3.1 becomes 3.2, and so on.

   The Branch radio button adds a new pair of numbers to the revision number. For example, Revision 1.0 becomes 1.0.0.1, Revision 2.1 becomes 2.1.0.1, Revision 1.0.0.1 becomes 1.0.0.1.0.1.
Creating a New Revision of a Package

The Release radio button increments the major revision number and changes the minor number to zero. For example, Revision 1.0 becomes 2.0, Revision 2.1 becomes 3.0, and so on.

If none of these choices is what you want, you can type the desired revision number directly in the Revision number field.

Because the package originally came from the Team /V Tutorial repository, the Repository field is already filled in with the correct name.

5) Click OK.

The new revision of the Cartography package is committed to the tutorial repository. The Package Browser updates as shown in figure 3-14, removing italics from the package, class, category, and method names that indicated that they had been modified.

Figure 3-14: The Package Browser After Committing the Cartography Package
Tutorial

CHAPTER 3

Updating Configuration Clusters

Notice that when you created the new revision of the Cartography package, the Map Browser Configuration cluster was modified, and now appears in italics in the Package Browser.

Specification Tracking

The cluster was modified because it was loaded and one of the loaded packages it specified (the Cartography package) now has a revision (3.1) that didn’t match what was specified (3.0). Visual Smalltalk Enterprise automatically updates the specifications in loaded clusters to match the revisions of the loaded packages.

The Specification Tracking in chapter 1 contains more information about the automatic updating of clusters when modules they reference are updated.

You can view the updated specification with the Cluster Organizer as follows.

To view the specifications in the Map Browser Configuration Cluster:

1. Choose Smalltalk / Open -> Cluster Organizer.
   
   The Cluster Organizer window appears.

2. Select Map Browser Configuration in the Cluster Organizer.
   
   The specifications in the cluster are listed in the top right-hand pane, as shown in figure 3-15. Notice that the specification for the Cartography package is italicized, indicating it has been changed, and now specifies revision 3.1, since that is the currently loaded revision.
Double click on the system menu icon to close the Cluster Organizer.

Creating a New Revision of a Cluster

You now need to create a new revision of the Map Browser Configuration cluster by committing it to the tutorial repository.

To commit a cluster:

1. Select the Map Browser Configuration cluster in the Package Browser, if it isn’t already.

2. Choose Module / Commit...

The Commit dialog appears, showing you the name and proposed revision number of the cluster. In this case, it shows Rev. 3.1 of the Map Browser Configuration cluster. It appears as shown in figure 3-16 (but with an empty comment initially).
3 In the Comment box, type:

**Updated the specification for the Cartography package from 3.1 to 3.2.**

4 Make sure the Revision radio button is selected to increment the revision number.

   Because the package originally came from the Team /V Tutorial repository, the Repository field is already filled in with the correct name.

5 Click OK.

   The new revision of the Map Browser Configuration cluster is committed to the tutorial repository.

### Comparing Packages

You can compare two revisions of a package. For example, at some point in the future, you may not remember what you changed between revisions 3.1 and 3.2 of the Cartography package.

**To compare two revisions of a package.**

1 Choose Smalltalk / Browse Packages to open a Package Browser, if one is not already open.

2 Select the Cartography package in the Package Browser.

3 Choose Module / Compare.
Comparing Packages

A Package Comparison Browser opens, as shown in figure 3-17. The Cartography package appears selected in the top left pane.

Figure 3-17: Package Comparison Browser

The top panes contain lists of packages that are currently open or loaded in your image. When a package is selected, the middle pane displays a list of definitions in that package. When a definition is selected, the bottom pane displays the source code associated with the definition.

Because you committed a new revision, the original revision of the Cartography package is no longer in your image. Before it can appear in the package list, you must reopen it.

4 Choose Right / Open....

The Open dialog appears letting you specify which module and revision to open.
Figure 3-18: Opening a Previous Revision of a Package

5 Select Team/V Tutorial from the Repositories list.

6 Select Cartography from the Modules list.

7 Select the original revision, 3.1, from the Revisions list.

8 Press Open.

9 In the Package Comparison Browser, select (Cartography 3.1).

   The parentheses indicate that the package is open, not loaded.

   The Compare button is enabled and the collapsed definitions for the two revisions appear in the middle panes—the definition lists. They currently contain the definitions for the classes that make up the Cartography package.

10 Press the Compare button.

   The differences between the packages are computed, and the definitions that are different are indicated by underlining. The Latitude class in the old revision is not underlined, while it is in the new revision. This indicates that the only differences are additions to the new package.

11 Press the Next button on the left-hand side.

   The first difference between the two packages is displayed in the bottom pane, as shown in figure 3-19.

   Differences are highlighted in one of two ways. Definitions that appear in one revision but not the other appear in boldfaced text (by default, green boldface on a color display). Definitions that appear in both revisions but are different appear in italics (by default, red italics on a color display).
As you can see, the differences in the method definitions are highlighted as well—they are underlined (by default, in red on a color display). In this case, one method has a comment that the other lacks.

12 Continue pressing the **Next** button to view all of the differences between the packages.

You can compare the text of any two definitions by selecting one on each side. Selecting a definition causes its text to appear in the contents pane with differences from whichever definition is selected on the other side highlighted. Try comparing the **negate** method against the **negated** method as an example.

The **Next** button performs the following steps for you automatically:

1. Expands the list of definitions for each package.
2. Selects the next definition that is different for whichever package you pressed the **Next** button.
3. Selects the matching definition in the other package.

Close the Comparison Browser when you are finished.
Making a New Package

So far we’ve been working with the Cartography package, but we’ve hardly glanced at the Map Browser package. Let’s take a look at it.

1. In a Package Browser, select the Map Browser package. Examine its contents.

   The package contains two classes: MapBrowser and MapDatum. MapBrowser displays the map and lets the user choose a point. MapDatum represents the point the user has chosen.

2. Examine MapDatum further. Select it and choose Global / Definition.

   The definition appears in the class definition editor at the bottom on the Package Browser, as shown in figure 3-20.

   ![Figure 3-20: Viewing the Definition of a Class in a Package Browser](image)

   MapDatum has two instance variables: latitude and longitude. It has accessing methods to determine the latitude and longitude, and comparison methods to determine if two points are the same.

   None of these characteristics are specific to the Map Browser application. The class MapDatum could be useful to a number of applications, such as an astronomy application that uses a map of Mars to track the location of a robot, or a business application that uses a map of France to show the locations of various sales offices, or an
epidemiological application that uses a map of New York City to track the incidences of tuberculosis. The class \texttt{MapDatum}, in short, does not belong in the \texttt{Map Browser} package. As a generally useful class, it belongs in a package of its own.

3 \textbf{Choose Module / New -> Package...}

A dialog opens with the prompt “Create a new package named:”.

4 Type the name “\texttt{Mapping}” in the field below the prompt (do not type the quotation marks).

5 Click \textit{OK} or press Enter.

The new package appears selected in the module list.

6 Select the package \texttt{Map Browser} again.

7 Select the class \texttt{MapDatum} in the Globals list. Drag it over to the module list and drop it on the new \texttt{Mapping} package you created.

\textbf{Figure 3-21: Dragging Class MapDatum to Package Mapping}

Both the \texttt{Map Browser} package and the \texttt{Map Browser Configuration} are marked as modified—the package because a class was removed from it, and the cluster because a package it completely specified has been modified.

8 Select the \texttt{Mapping} package to verify that the class really has moved.

9 Choose \textit{Module / Commit...} to bring up the Commit dialog.
10 In the Comment box, type:

MapDatum is a class generally useful for any application that requires a map to contain a point identified by longitude and latitude.

Because this package is new, the Revision field displays 0.1.

11 Because this package is new, the Repository field is empty. Click the drop-down button in the Repository field to view a list of the available repositories.

12 Select the tutorial repository and press OK.

### Updating the Map Browser Configuration Cluster

Because MapDatum is now in a separate package, we need to create a new revision of the **Map Browser** package, since the MapDatum class was removed from it, and add the **Mapping** package to the **Map Browser Configuration** cluster, so when the cluster is loaded, all of the packages required by the cluster are loaded too.

**To commit the Map Browser package:**

1 Select **Map Browser** in the module list of the Package Browser.

2 Choose Module / Commit...

3 In the Comment box, type:

   **Moved MapDatum class to the Mapping package.**

4 Check that the revision is set to 3.2 and the repository is set to Team/V Tutorial.

5 Click OK or press Enter.

**To add the Mapping package to the Map Browser Configuration Cluster:**

1 Select the **Mapping** package in the Module list.

2 Drag the package name and drop it on the **Map Browser Configuration**, as shown in figure 3-22.
This adds a specification for the **Mapping** package to the **Map Browser Configuration** cluster. If the cluster is not expanded to show that it now contains three packages, click on the plus sign.

The final step is to commit the **Map Browser Configuration** cluster to record the configuration for the **Map Browser** application.

**To commit the configuration cluster:**

1. Select **Map Browser Configuration** in the module list.
2. Choose **Module / Commit**...
3. Enter the following comment into the Commit dialog.
   
   **Added the Mapping package and updated the Map Browser revision.**
4. Check that the revision is set to 3.2 and the repository is set to **Team/V Tutorial**.
5. Click **OK** or press Enter.

All revisions of all modules in the Map Browser application are now committed.
Distributing Your Changes

In this tutorial, you have taken an existing application and modified it, saving your changes as new revisions of packages and clusters in the tutorial repository. What happens next? It depends on who the customers for the application are.

Migrating to a New Revision

If a colleague has the old version of the Map Browser application in his development image, you can tell him to update (migrate) to your new version by selecting his revision of the Map Browser Configuration cluster and using Module / Migrate... to migrate to your revision.

Loading a New Revision

If another associate hasn’t been using the Map Browser application, she can load your revision into her development image by connecting to the tutorial repository and using Module / Load... to load the current revision of the configuration cluster, just as you did at the beginning of this tutorial.

Delivering the Map Browser as a Smalltalk Link Library

You can also write the Map Browser application into a Smalltalk Link Library (.SLL) file for delivery to customers that don’t have access to the tutorial repository by selecting the Map Browser Configuration cluster and choosing Module / Build Library...

For more information about using these facilities for distributing an application, see chapters 4 and 5.
Overview

This chapter describes the tools and other enhancements to the base Visual Smalltalk development environment provided by Visual Smalltalk Enterprise. Chapter 5, *Team Programming Tasks* describes how to use these tools, menus items, and dialogs in the context of normal programming tasks.

The primary tools described are:

- **Package Browser**
  This is the central tool in the enhanced Visual Smalltalk Enterprise tool set. This browser provides a view of your application in terms of the packages and clusters containing the application code.

- **Repository Browser**
  This browser provides all the tools you need to maintain a repository structure and the modules stored in your repositories.

- **Package Organizer**
  This tool allows you to examine and reorganize definitions in open or loaded packages. In this tool you can also add and order initialization expressions, controlling the way the program is initialized.

- **Cluster Organizer**
  This tool allows you to view, edit, reorganize, and add new specifications for open and loaded clusters.

- **History Browser**
  This browser allows you to examine the different versions of a definition.

- **Definition Group Browser** and **Message Browser**
  These browsers let you examine all definitions that satisfy a particular query, such as senders or implementors of a message or set of messages.

- **Comparison Browsers**
  These browsers allow you to compare versions of definitions and revisions of packages and clusters.
• **Specification Editor**
  This editor lets you edit information associated with open or loaded specifications to complete or minimize a specification for a module.

• **Access Control Editors**
  There are editors for assigning access rights to users and groups, allowing access to repositories and repository entities.

### Figure 4-1: Visual Smalltalk Enterprise Tools and Browsers

#### Locating Modules with the Browsers

Open and loaded modules are treated differently. The following table indicates which browser to use to locate a particular module or one of the special packages or clusters.
Table 4-1: The Right Tool for the Right Module:

<table>
<thead>
<tr>
<th>Module</th>
<th>Package Browser</th>
<th>Package Organizer</th>
<th>Cluster Organizer</th>
</tr>
</thead>
<tbody>
<tr>
<td>loaded packages</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>loaded clusters</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>open packages</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>open clusters</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>module (cluster-package) hierarchies</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>** top level cluster **</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>** unpackaged package **</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Additions to the Visual Smalltalk Menus

Visual Smalltalk Enterprise tools typically have menus and menu options not present in Visual Smalltalk tools. Several of these are described in connection with the descriptions of the tools in this chapter and the tasks in the next chapter. For details about any particular menu item, see the help text for that item.

Text Display Conventions

Visual Smalltalk Enterprise uses several type face and color conventions in list panes in many tools to provide additional information about the current state of a module, definition, or specification. Details about the conventions are described with the tool descriptions, but are summarized in the following table.

Note that the color indicators mentioned below are the defaults, and can be changed in the Color Preference Editor (refer to Color Preference Editor in chapter 2).

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Italic Blue Text</td>
<td>In most tools, italics indicate that the item shown has been modified since its containing module was last committed, or that the module has never been committed. Modified items are also shown in blue (by default) for additional emphasis.</td>
</tr>
<tr>
<td>Italic Red Text</td>
<td>In the Package Comparison Browser, red italic text indicates that the definitions differ between the selected packages. In the Cluster Comparison Browser, it indicates the selected specification differ.</td>
</tr>
</tbody>
</table>
Using Drag and Drop

Most of the tools support the following drag and drop operations.

Full drag and drop (both move and copy operations) are supported as follows:

- Annotations can be dropped on definitions, modules, or specifications.
- Method definitions can be dropped on class definitions.
- Pool variable definitions can be dropped on pool definitions.
- A definition or group of definitions can be dropped on a package.
If the move or copy causes a conflict, a Conflict Notification Dialog is opened.

- Specifications can be dropped on clusters.

Partial drag and drop operations are supported for the following:

- Definitions can be copied to package-independent lists of definitions, as in the definition lists used to browse senders and implementors.

- Modules can be copied to clusters. A new specification of the module is added to the target. If the cluster has its usage set to “group,” a minimal specification is created; otherwise, a complete specification is created.

- Class and method definitions can be dropped on text panes, which appends a file-out representation of them to the text.

- A method definition can be moved to a different method category.

**Package Browser**

The Package Browser lets you examine all loaded packages and clusters in your image and all the definitions within each package. You can modify existing packages or definitions and define new ones.

Items that have been modified since you last committed them appear in italics and in the modified color (blue, by default) for additional emphasis. Items that belong to the module you are examining (the name selected in the module list) appear boldfaced.

**To open a Package Browser:**

> Select Smalltalk / Browse Packages.

or

> Press Ctrl + P.

**Package Browser Panes**

A Package Browser is shown in Figure 4-2.

The **module list** displays the names of all the clusters and packages currently installed in the system, shown in alphabetical order.
The **global list** displays all global definitions (classes, global variables, and pools) contained in the currently selected module. A cluster contains all the definitions contained by packages and clusters specified by it.

The **category list** displays the method category names defined for the selected class. If the class contains methods that have not been organized into categories, it shows the special category **unclassified**. The special category **all** shows all methods in the class, regardless of their category.

Above the category list are two radio buttons, **instance** and **class**. The selected button selects whether instance method categories or class method categories are displayed.

When you select a category, the **method list** displays the names of all the methods in the category, listed in alphabetical order.

Above the method list, the **inheritance box** lets you view methods inherited from superclasses. The drop-down list contains the entire hierarchy above the class selected in the global list, with superclasses on top, as shown in figure 4-3.
When you select a superclass from the inheritance list, the method list includes all methods inherited by the current class from its superclasses, up to and including the selected superclass.

When a subclass overrides a superclass, it may not be clear which implementation you are viewing. The *implementor box*, below the method list, shows the class that actually implements the selected method. Only classes that implement the method appear, listed in hierarchical order. Select any class to view its implementation of the selected method.
At the bottom left of the Package Browser and many other tools is the `context` button, which displays the name of the package containing the definition you are viewing. This is not always the same as the module selected in the module list. If you click on the context button, the package shown on the button becomes selected in the module list, and the Package Browser resets the other panes to correspond to that new selection.
The *version* button is at the bottom right of the Package Browser and several other tools. This button contains information that identifies the version of the selected definition. If versions are not maintained for that type of definition, the button is blank. A string such as

**Visual Smalltalk Enterprise 3.1**

indicates that the definition is part of the Visual Smalltalk Enterprise product.

Otherwise, you see a creation date, time, and author for the selected definition and you can click on the version button to open a History Browser on the selected definition, letting you view its previous versions, if any.

**Package Browser Contents Pane**

The *contents pane* shows a wide variety of information depending on settings in the *Global* menu and which item is currently selected. In the preceding figure, the contents pane shows the text for the selected method.

The information displayed depends on the selected item:

- module
- global
- pool, or
- class.

For each of these for types of items, the contents pane can be modified by one of the selections as appropriate to the context from the Package Browser’s *Global* menu item:

- Comment
- Definition
- Initialization
- Description
- Hierarchy

The *File / Save* menu item typically saves any changes made in the contents pane.

**Package Browser Module Editor**

As shown in the following figure, the module editor is visible in the contents pane of the Package Browser when a module is selected but no global is selected. It contains a text pane for editing the module comment and a read-only text information
pane for the module that documents its repository, revision number, author, whether it's a cluster or package, and the latest revision comment.

Figure 4-6: Package Browser Module Editor

Package Browser Global Variable Editors

The global variable editors are visible when a global variable is selected. The contents pane can show a comment for the global variable, a definition pane, the initialization expression, or a descriptive phrase.

The figure below shows the contents pane when a global variable is selected and the Global / Definition item is toggled on. This lets you edit both the initialization expression in a pane on the left and the global variable comment on the right.
Figure 4-7: Package Browser Global Variable Definition

Package Browser Pool Editors

The pool editors are visible when a pool is selected. The contents pane can show a comment for the pool, a definition pane, or a descriptive phrase.

The figure below shows the contents pane when a pool is selected and the Global / Definition item is toggled on. This lets you view any pool variables defined in this pool in the pane on the left. If you select a pool variable on the left, you can edit its initialization expression and its comment in other panes. See Working With Variables in chapter 4 for information on using menu selections from the Variables menu to alter the pool variables in the list.
Figure 4-8: Package Browser Pool Definition

Package Browser Class Editors

When a class is selected in the Package Browser, the contents pane shows information about the category or method, if either are selected:

- If a category is selected but a method is not, the contents pane shows a method template.
- If a method is selected, the contents pane shows the method source code.

If a method category is not selected, the contents pane shows information about the class, controlled by which item from the Global menu is toggled on. This information can show the class comment, definition, initialization expression, description, or hierarchy. When a Package Browser is first opened, the default is the class comment.

- **Global / Comment** displays an editable class comment.
- **Global / Definition** displays a class definition pane that lets you edit or create a class definition as described below.
- **Global / Initialization** displays an editable text pane for the class initialization definition.
- **Global / Description** displays a description of the class and its components, similar to the class creation message used by the Class Hierarchy Browser. (Although you can edit in this pane, you cannot save the changes, and should not edit in it.)
Figure 4-9: **Package Browser with Class Comment**

The class `MapDatum` represents the point on a map in a geographic application.

Figure 4-10: **Package Browser with Class Description**

- `Global / Hierarchy` shows the superclasses and subclasses of the selected class.
Figure 4-11: **Package Browser with Class Hierarchy**

The class definition editor lets you edit the definition of a new or existing class. It is visible when only a class is selected, without a method category, and the **Global / Definition** item is toggled on, or when you select **Global / New / Class**.

**Package Browser Class Definition Editor**

The class definition editor lets you edit the definition of a new or existing class. It is visible when only a class is selected, without a method category, and the **Global / Definition** item is toggled on, or when you select **Global / New / Class**.

Figure 4-12: **Package Class Definition Editor**
Defining a Package or Cluster in the Package Browser

To define a package or cluster:

1. From the Module menu, choose New and then either Package..., Cluster..., Group..., or Configuration... from the submenu.

2. In the prompter, type the module name and click OK.

   The module definition editor appears in the contents pane as shown in figure 4-13.

Figure 4-13: Module Definition Editor

3. In the Module Comment field, describe the module.

   It's a good idea to include the purpose of the module, an overview of the classes and other definitions it will contain, and any other information that would be valuable for other users. If you don’t know some of these things now, you can update the comment later.

   The Module Information field is read-only. For a new module, it is empty except for the module name. It is filled in after you have committed a new revision of the module.

4. When you are done, select File / Save, if you edited the comment.
Defining a Class

To define a class:

1. Select the package in which you want the class to be defined.

2. From the Global menu, choose New -> Class.

   The class definition editor appears in the contents pane as shown figure 4-14.

Figure 4-14: Class Definition Editor

![Class Definition Editor](image)

3. In the Class field, type the name of the new class.

4. In the Superclass field, type the name of the superclass.

   The Indexables combo box lets you specify the type of indexable instance variables, if any. The initial selection is none. Refer to the Visual Smalltalk Enterprise Language Reference for information about the options.

5. Change the selection in the Indexables combo box if you want to specify object or byte indexables.

   Under Variables, four radio buttons let you specify instance, class, class instance, or pool. The initial selection is instance.

6. To define an instance variable, select the instance radio button, type the new instance variable name in the pane to the right of the radio button and above the variables list pane, and press Enter.

   The instance variable appears in the list below.
7 To define a class or class instance variable, click on the appropriate radio button and enter the name in the pane to the right as indicated in the previous step.

8 To specify that your class needs to use a pool, click on the pool radio button. Then, type the name of the pool in the top pane of the combo box and press Enter.

**NOTE:** This does not define the pool. You must define the pool before referencing it here in the class definition editor.

The pool appears in the list below.

When the class definition editor is visible, the Variables menu can be used to edit the variables list.

9 In the Comment field, comment your class and its variables.

If a variable is selected in the list of variables, this field lets you comment the selected variable. If no variable is selected, this field lets you comment the class.

10 When you are finished with the class definition, choose File / Save to save all variables and comments belonging to the class.

Your new class appears selected in the global list.

**Working With a Module**

Operations on modules are accessed through the Package Browser’s **Module** menu.

**The Package Browser Module Menu**

The **Module** menu provides operations applicable to the currently selected module, or those that cross module boundaries, such as locating a module or creating a new one. This menu is very similar to the Package Organizer’s **Package** menu and the Cluster Organizer’s **Cluster** menu.

The **Module** menu items are:

- **New** displays a submenu containing **Package…**, **Group…**, **Configuration…**, and **Cluster…**. Choosing any of these opens a prompter that lets you specify the name of a new loaded module. Type the name and click **OK**, and your new module appears selected in the module list.

- **Rename…** pops up a dialog letting you specify the new name of the selected module. It is enabled when a module is selected.
NOTE: Renaming a module only renames the copy currently in memory, in effect creating a new module. The revision history is still accessible under the original module name.

- **Unload** deletes the selected module from the image, after confirmation, but does not remove it from its repository. If unloading the module would cause a conflict, the module is not deleted, and a Conflict Notification Dialog appears. See Appendix A for instructions on resolving these conflicts before you delete the module. This menu item is enabled when a module is selected.

NOTE: Unloading a loaded package deletes the definitions in that package from the image. Unloading a cluster does not delete the modules contained in it unless the cluster contains the last reference to those modules.

- **Convert To > Group, Configuration, Cluster** automates the process of either maximizing or minimizing cluster specifications. For more information, see *Converting Between Clusters, Groups, and Configurations* in chapter 5.

- **Load**... opens the Load dialog that lets you specify the repository and revision of the module you want to load into your image. See the section entitled *Loading and Migrating Modules* on page 131 for more information.
• *Migrate*... opens the Migrate dialog, similar to the Load dialog, letting you select a replacement for the currently selected module in the Package Browser’s module list. Use this to update to newer (or older) revisions of a module or swap it for a different module entirely. The current revision of the currently selected module from the Package Browser is automatically selected in the Migrate dialog. (If other revisions appear above the one that is selected, you do not have the newest revision loaded.)

• *Commit*... pops up the Commit dialog, letting you type in a revision comment, increment the revision number, and specify the repository to which you want to commit the currently selected module. These operations are discussed further in *The Commit Dialog* section below.

• *Commit All* does a bulk commit on all modified modules in the currently selected cluster. A dialog opens listing the modified modules, and allowing you to enter a single revision comment.

• *File Out*... pops up a file dialog, letting you specify the file pathname to which to file out the selected module. The file out format is compatible with Visual Smalltalk and VisualWorks. It does not contain package and cluster organizational information, so this information will be lost. For further information about turning packages into file-ins, see the section entitled *Installing Unpackaged Code* in chapter 5.

• *Export*... lets you distribute a module to a teammate that has Visual Smalltalk Enterprise but does not have access to your repository. It prompts you for the name of a file and then writes the current contents of the selected module to that file in export format. See *Teamwork* in chapter 5 for more information.

• *Copy*... lets you copy the currently loaded revision of the selected module to another repository. The menu item is enabled if the selected module has been committed at least once. The copy preserves the revision number, author, and revision comment. To copy a revision to a PVCS repository, there must be a valid parent revision in the target repository.

• *Build Library*... opens a Build Library Dialog to create a Smalltalk Link Library from the selected module. For more information, see the section entitled *Build Library Dialog* on page 169.
• *Find Module*... prompts for a module name or pattern with wild cards. If the search is successful, it selects that module in the Package Browser, expanding the module list if necessary to show the (previously) hidden module.

• *Edit Lock*... opens a dialog allowing you to lock or unlock the selected module. If you lock a module, others cannot create new revisions of it until you have unlocked it.

• *Compare* opens a Package Comparison Browser on the selected module if it is a package, or a Cluster Comparison Browser if it is a cluster.

• *Initialize* initializes the selected module. All class, pool, global variable, and *ad hoc* initializers are executed in the sequence specified in the Package Organizer.

• *Update* tells the Package Browser to update its list and contents panes. You must do this if you have modified a definition in another tool; otherwise, the Package Browser may not reflect the current state of your image.

• *Filter*... displays the module filter dialog. By selecting the appropriate check boxes, you can limit the modules shown in the modules list to those of interest to you.

### The Commit Dialog

The Commit Dialog opened by the *Module / Commit*... menu item lets you enter a revision comment, increment the revision number, and specify the repository to which you want to commit. It appears as shown below.

*Figure 4-16: Commit Dialog*
The information at the top of this dialog shows the name of the module and the revision number of the base revision, the revision from which it was derived.

The revision number is automatically incremented to reflect the next valid revision number for the specified repository. However, you can specify a number other than the default by clicking on one of the radio buttons provided, or by typing the desired revision number directly in the Revision text entry field.

The Branch radio button adds a new pair of numbers to the revision number. For example, Revision 1.0 becomes 1.0.0.1, Revision 2.1 becomes 2.1.0.1, Revision 1.0.0.1 becomes 1.0.0.1.0.1.

The Release radio button increments the major revision number and changes the minor number to zero. For example, Revision 1.0 becomes 2.0, Revision 2.1 becomes 3.0, Revision 2.1.0.1 becomes 3.0, and so on.

See the section entitled Revisions in chapter 1 for more information about revision numbering.

If the package has been committed before, the Repository combo box is already filled in with the name of the repository to which it was previously committed. If you want to commit to a different repository, select from the drop-down list, and update the revision number as necessary.

To commit to a repository that is not in the list, click the Repository... button to bring up the Repository Browser to modify the list of repositories shown. This browser is described in Repository Browser on page 136.

OK is enabled if a repository has been selected. When the Repository combo box shows the name of the correct repository, click on OK to commit the revision as specified.

Click on Cancel to close the dialog without performing the commit.

**Working With Globals**

The Global menu lets you define, modify or delete global definitions (classes, global variables, and pools) within the selected module. The menu items describe operations that are appropriate for all globals unless otherwise noted.
• **Global / New** has three submenu items: **Class**, **Global Variable**, and **Pool**. **New** is enabled when a package is selected in the module list (clusters can only hold packages or other clusters).

  **Class** causes the class definition editor to appear in the contents pane. See the section entitled **Package Browser Class Definition Editor** on page 114 for more information about this editor.

  **Global Variable** prompts you for the name of the new global variable. Type the name and click **OK**. Your new variable appears selected in the global list. See the section entitled **Package Browser Global Variable Editors** on page 110 for more information about this editor.

  **Pool** prompts you to enter the name of the new pool. Type the name and click **OK**. Your new pool appears selected in the global list. See the section entitled **Package Browser Pool Editors** on page 111 for more information about this editor.

• **Rename...** pops up a dialog letting you specify the new name of the selected global. It is enabled if a global definition is selected. References to the global are renamed as well.

• **Delete** pops up a confirmation dialog. Clicking **OK** deletes the selected global from the image.

**NOTE:** Use **Smalltalk / Browse / Unresolved** to find references that need to be fixed.
The next five items are toggles that control the information in the contents pane of the Package Browser—only one of the five can be toggled on at a time.

- **Comment** displays the comment for the selected global in the contents pane. You can edit the comment and save your changes. This item is enabled if a global definition is selected.

- **Definition** displays the definition editor for the selected global in the contents pane. You can edit the definition and save your changes. This item is enabled when a global definition is selected.

- **Initialization** displays the initialization expression for the selected global in the contents pane. You can edit and save the expression. This item is not available for pools.

- **Description** displays a textual description of the selected global definition.

- **Hierarchy** displays the hierarchy for the selected class in the contents pane. The hierarchy is not editable in the contents pane. To change the superclass structure, redefine the affected classes using the class definition editor. See the section entitled Defining a Class. This item is enabled only when a class is selected.

The next four items on the Package Browser’s Global menu let you search for specific references.

- **Browse References** pops up a Definition Group Browser containing a list of all definitions in the image that refer to the selected global. See the section entitled Browsing Specific Definitions on page 158 for information on using the Definition Group Browser.

- **Find Class (or Global)...** pops up a dialog that lets you specify the name of the class, pool, or global variable you are looking for. Click OK, and the global appears selected in the Package Browser. You can use wildcards for pattern-matching.

- **Inspect** opens an inspector on the current value of the selected global.

- **Initialize** is enabled if a global definition is selected. It initializes the selected global. If the selected global is a class, the class initializer is executed. If the selected global is a global variable, the value of the global variable is set to the result of the execution of the initialization expression. If the selected global is a pool, the initialization expressions defined
for each of the pool variables are executed in an arbitrary order.

**NOTE:** If the selected package does not include the entire pool, this menu item initializes only the portion in the package.

- *Filter...* displays the global filter dialog. By checking selected check boxes, you can limit the contents of the globals list to any combination of classes, globals, and pools, modified and or unmodified.

**Using Drag and Drop with Global Definitions**

If the selected global is contained in the selected module, the drag operation manipulates the class and all methods in the selected module. If the definition of the selected global is *not* contained in the selected module (that is, the item is displayed with an underline rather than a bold font), the drag object is the group of definitions that are defined in the selected module.

**Working With Categories and Methods**

The Package Browser’s *Method* menu operates on categories and methods for the selected class.

Categories provide a way to group related methods. They do not affect execution at all: any given method behaves the same no matter which category it is in. When you use them consistently, category names enhance program understandability and maintenance.

**NOTE:** Consult the *Programming Guidelines* appendix in the *Visual Smalltalk Enterprise Language Reference* for recommendations about category names.
Categories can be hierarchical. For example, you can have subcategories within a category as shown below. This figure shows two hierarchical categories: `accessing` is collapsed, but `private` is expanded to show its subcategories.

Figure 4-18: Hierarchical Categories

A hierarchical category has an expansion indicator to its left. When the category is collapsed, the expansion indicator is a rectangle with a plus inside. If you select a top-level category, you see all the methods in that category. If you click on the expansion indicator with the plus sign, the category expands to show its subcategories and the plus sign changes to a minus.

When you create a hierarchical category, use a hyphen (-) between the category names. For example, the category names above were created with the names

- `private-accessing`
- `private-displaying`
- `private-initialization`
Figure 4-19: Package Browser's Method Menu

- **New...** is enabled if a package is selected in the module list. If a class and category are selected, the text in the contents pane is changed to the method definition editor. If no class is selected, this menu choice opens a Method Definition dialog as shown in figure 4-20. This is useful for adding loose methods to a package.

Figure 4-20: Method Definition Dialog

- **Delete** removes the selected method from the image after confirmation. You cannot delete methods that belong to other than the selected module. Use the **Context** button to select the appropriate package if necessary.
NOTE: Browse senders of the message before you delete the method, so that you can be sure that its deletion will not cause a problem.

- Senders opens a Definition Group Browser on all senders of the message represented by the selected method.
- Implementors is used to open a Definition Group Browser on all implementors of the message represented by the selected method.
- Local Senders and Local Implementors have the same effect as their namesakes in this menu, described above, except that the scope of the search is limited to the selected class and its subclasses.

See the section entitled Browsing Specific Definitions on page 158 for further information on these browsers.

- Messages opens a Messages Browser on all senders or implementors of methods represented by the messages sent in the selected method.
- Compare Versions opens a Version Comparison Browser on the selected method. See the section entitled Comparing Methods for a complete description of this process.
- Format is used to format the selected method according to the conventions specified in Appendix B.
- New Category... prompts for the name of a new category to be added to the selected class. Hierarchical category relationships are indicated with a dash between the components, such as private-initialization.

NOTE: A category name must not include a comma.

- Rename Category... prompts for the path of a new or existing category. All methods in the currently selected category are moved to the specified category and the selected category is removed.
- Delete Category removes the selected category, including every method in it, from the image. A confirmation dialog is displayed if the category is not empty.
- Find Category... is enabled if a method is selected. If you have either a category with subcategories or the special category
** all ** selected, and want to know which category contains the selected method, use this menu item to quickly select the category or the special category ** unclassified ** if the method has not been placed in any category.

- *Recategorize...* moves the selected method to another category that you choose from a pop up list of current categories. The new category appears selected in the category list, and the method remains selected in the method list.

  You can also use drag and drop to move a method to another category or another package.

- *Filter...* displays the method filter dialog. By checking selected check boxes, you can limit the methods shown in the methods list.

### Working With Methods

**To define a method:**

1. Select a package, a class, and a category in which to define the method.

   If the class and category aren’t listed, you can still create the method as a loose method in the selected package.

2. Select *Method / New*.

   A method template is displayed in the contents pane.

   You can also simply select an existing method, and change the selector to a new selector. When you save the method, it will be created as a new method.

3. Select the text in the method editor and type the message selector, local variables, and the body of the method.

   Be sure to comment your method.

4. When you are finished, save your text.

   The new method belongs to the selected package.

**To modify a method:**

1. In the class list, select the class in which the method is defined.

2. In the category list, select the category in which the method is defined.

3. In the method list, select the method you want to modify.
TIP: If you don’t know what category the method is in, you can also select the special category **all** to show all methods in alphabetical order.

The method source appears in the contents pane.

4 Select the text you want to modify and type your change.

5 When you are finished, save your text.

Using the implementor box, you can modify a method implemented in a class other than the class selected in the class list. For example, you select Class B in Package B, and use the implementor box to view the methods it overrides. You can then edit a method having the same name that was implemented by Class B’s superclass Class A belonging to Package A. When you save your change, a dialog pops up asking whether you want your new method to be part of Class A or Class B—in other words, whether you want to edit or override the method. The edited method belongs to the package it belonged to before.

**Working With Variables**

The Package Browser’s *Variable* menu lets you create new variables or work with existing ones. It is shown in the following figure.

**Figure 4-21: Package Browser Variables Menu**

The *Variable* menu items are:
• *New...* pops up a dialog letting you specify the name of the variable you want to add to the current definition. The new variable is then added to the selected list. This item is enabled when a class or pool is selected and its definition is being edited.

If a class is selected, the type of variable (instance, class, class instance, or pool) is determined by the radio buttons in the class definition editor. This menu item creates new instance, class, and class instance variables, but you must specify the name of an existing pool definition. (Use *Global / New -> Pool* to create a new pool definition.)

If a pool definition is selected when this item is chosen as shown in the figure above, it lets you add new pool variables to the pool. In the figure, the pool variable **EmeraldGreen** was just added to the list on the left using this menu item.

• *Rename...* pops up a dialog letting you specify the new name of the selected variable. This item is enabled if a pool variable is selected in a pool definition editor or an instance, class, class instance, or pool is selected in the class definition editor. In the case of a selected pool in the class definition editor, rename does not actually change the name of the pool as referenced by the class—it only changes the class definition to refer to a different (pre-existing) pool.

If there are references to the changed variable, a definition group browser opens.

**CAUTION:** To avoid the possibility of eventual system failure, do not rename or delete a pool variable in the pool that contains the system-defined constants for your operating environment, or **CharacterConstants** pool. The system cannot find references to variables in either of these pools.

• *Delete* pops up a confirmation dialog. Clicking *OK* removes the selected variable from the image, or, in the case of a pool, it deletes the reference from the class to the pool. It does not delete the pool itself. This item is enabled if a pool variable is selected in a pool definition editor or an instance, class, class instance, or pool is selected in the class definition editor. Use *Smalltalk / Browser / Unresolved* to update references to the variables you have deleted.
• **Browse References** opens a Definition Group Browser containing a list of all definitions in the image that refer to the selected variable in any manner. See the section entitled *Browsing Specific Definitions* for information on using the Definition Group Browser. This item is enabled if a pool variable is selected in a pool definition editor or an instance, class, class instance, or pool is selected in the class definition editor.

• **Inspect** opens an inspector on the value of the selected variable. It is enabled when a pool variable is selected in a pool definition editor.

• **Initialize** initializes the selected variable. The value of the variable is reset to the result of the execution of the initialization expression. It is enabled when a pool variable is selected in a pool definition editor.

**Loading and Migrating Modules**

There are several ways to load a module from a repository.

One is to simply select a repository, module, and revision in the Repository Browser, then select *Revision / Load* in the browser menu. If another revision of the module has already been loaded, you will be prompted to verify that you want to migrate to the newly selected revision.

If you do not have a Repository Browser open, you can select *Module / Load* or *Module / Migrate* from the Package Browser menu. A dialog opens allowing you to select the repository, module, and revision of the module to load.

A similar dialog lets you open (without loading) a package or cluster. Select *Package / Open...* in a Package Organizer or *Cluster / Open...* in a Cluster Organizer to open a module. The Open dialog looks like this:
Select a repository, module, and a revisions, then click on the *Load*, *Migrate*, or *Open* button (depending on which dialog is open) to access the module contents. You can also double-click on a revision number to load, migrate, or open the module.

The buttons in these dialogs are:

- The name of the button in the lower left corner reflects the current operation: *Open*, *Load*, or *Migrate*.
  
  Select a revision and the *Open/Load/Migrate* button is enabled. When you press this button, the requested operation is performed and the dialog is closed if it is successful. If not, a Conflict Notification dialog opens.

- *Cancel* closes the dialog without performing the operation.

- *File...* opens a file Open dialog. You select a file in file-in format or export format, and the requested operation is performed on this file.

- *Repository...* opens a Repository Browser to modify the list of repositories. For example, you can connect to a new repository. When you return to the *Open/Load/Migrate* dialog, the current list of repositories is updated.

### Loading a Revision of a Module

The Load dialog opens when you choose any of the following:

- *Module / Load...* in the Package Browser
- *Package / Load...* in the Package Organizer
- *Cluster / Load...* in the Cluster Organizer

A Load dialog is shown below.
Figure 4-23: Load Dialog

It looks the same as the Open dialog, except that

- the button in the lower left is labeled Load, and
- a checkbox lets you specify whether to initialize the definitions in the module after you load them.

Press Load and the specified revision of the module is loaded into your image. Visual Smalltalk Enterprise checks for conflicts, and opens a Conflict Notification dialog if any are detected. If a conflict occurs, the specified module is not loaded.

If you click on the File... button, you can specify a file to be loaded and checked for conflicts. This file can either be in export format or in file-in format. See Teamwork in chapter 5 for more information on exporting a file.

A file-in is loaded as a package named From <FileName>, the items it contains are treated as definitions, and it is checked for conflicts before loading. See Installing Unpackaged Code in chapter 5 for further details.

**Migrating to a Different Revision**

The Migrate dialog is the same as the Load dialog except that the lower left button is labeled Migrate. It lets you unload the selected revision in the Package Browser, Cluster Organizer, or Package Organizer and load another in its place. You can migrate to a revision of a different module or even a revision in another repository entirely.
The Migrate dialog also lets you choose whether to execute the initialization expressions after the migrate operation.

When the Migrate dialog opens, the current open or loaded revision is selected.

Migrating revisions is useful for updating to a new revision of a package, perhaps recently committed by another developer, or for reverting to an earlier revision to undo some modifications.

### Steps Performed When Loading and Unloading a Package

When you load a package, the following operations occur in order:

1. All definitions in the package are analyzed for conflicts with definitions in other loaded packages. If any problems are found, the load fails.

2. References to pool variables and superclasses by class definitions are resolved against the other definitions in the package and those already loaded in the image. If a reference cannot be resolved, the load fails.

3. All global definitions (classes, globals, pools) in the package are processed in indeterminate order. Processing a class definition includes processing any instance variable, class instance variable, or class variable definitions for the class.

4. All method definitions are compiled and installed in their respective classes in an indeterminate order.

5. If you have requested initialization when you load or migrate, all initializers (class, pool variable, global variable and ad-hoc) are executed in the order that they appear in the package.
This is the only place where the order of definitions in a package is significant.

The initialization step can be performed independent of loading.

Unloading a package removes the methods first, and then the global classes, variables, and pools.

**Steps Performed When Loading a Cluster**

When you load a cluster, the following operations are performed:

1. The conditional and optional inclusion flags for all the specifications in the cluster being loaded are analyzed to determine which specifications should be resolved.

2. The specifications that need to be resolved are resolved to specific packages, which are added to the list of packages to be loaded.

3. If a specification resolves to a cluster, the specifications in that cluster are recursively resolved until all specifications are resolved to packages to be loaded.

4. Any packages in the load list that are already loaded in the development environment are eliminated from the list of packages to be loaded.

5. The packages in the load list are loaded.
Repositories are a crucial component of Visual Smalltalk Enterprise, providing the organizational structure for storing packages and clusters. The Repository Browser is the central tool for maintaining repositories. All maintenance activities are performed from this browser.

**To open a Repository Browser:**

> Select Smalltalk / Open / Repository Browser

The browser is also opened when you click the *Repositories* button in the Open, Load, and Migrate Module dialogs, or the Commit Dialog.

![Repository Browser Diagram](image)

The Repository Browser displays four panes. The Repositories pane displays a list of currently connected repositories. When you select a repository, the Entity list displays the entities in that repository. When you select an entity, the Revision list displays a list of available revisions. The contents pane displays summary information on each the currently selected repository, entity, and revision.

The main operations provided in the Repository Browser are presented in three menus: *Repository*, *Entity*, and *Revision*, each representing the repository object the menu items operate on.
Repository Operations

The Repository Browser Repository menu provides access to the following repository operations.

These operations are subject to restriction based on access rights.

- New... opens the Create Repository dialog, allowing you to create a repository in a new or existing directory. Instructions for creating a repository are provided in Creating a Repository below.

- Destroy deletes the entire repository, including all data and repository management files. The directory containing the repository is not deleted.

- Connect... opens the Connect Repository dialog, allowing you to select a directory containing the repository you want to connect. Instructions for connecting to a repository are provided in Connecting to a Repository below.

- Reconnect... allows you to connect to a repository that you have previously connected to, but the drive letter mapping or directory path changed.

- Disconnect removes the connection to the selected repository. The repository files remain intact, but you no longer have access to the modules it contains.

- Check... examines the selected repository for errors, and optionally repairs errors.

- Check Directory... examines the repository in a directory you select. A directory dialog is displayed allowing you to select the directory.

- Copy... creates a copy of the selected repository in another directory. A directory dialog is displayed allowing you to select the target directory.

- Copy As... creates a copy of the selected repository in another directory, and with a different repository name. A prompter is displayed asking for the new repository name, followed by a directory dialog allowing you to select the target directory.

- Find Entity... searches all currently loaded or opened repositories for an entity (package or cluster). A prompter is displayed asking for the entity name. If the entity is found, it is selected.

- Edit Access allows you to edit access rights for this repository, if you have sufficient privilege. Editing access rights is
described in Access Control Editors later in this chapter and in Controlling Repository Access in chapter 5.

Entity Operations

The Repository Browser Entity menu provides access to the following operations on repository entities. Repository entities are any file object that can be saved in and revisioned by the repository system. Repository entities are most often packages and clusters, but may be parts or other objects.

These operations are subject to restriction based on access rights.

- **New...** opens a dialog prompting for a name for the new repository entity and the type. The entity is then added to the selected repository.

- **Copy To...** opens a dialog displaying a list of connected repositories, and copies the currently selected repository entity to the target repository you select in the dialog. The complete revision history is copied.

- **Edit Access** allows you to edit access rights for this repository entity, if you have sufficient privilege. Editing access rights is described in Assigning Access Rights in chapter 2.

- **Filter...** displays the entity filter dialog. By checking selected check boxes, you can limit the entities shown to any combination of packages, clusters, and other repository entities.

Revision Operations

The Repository Browser Revision menu provides access to the following operations on entity revisions.

These operations are subject to restriction based on access rights.

- **Open** provides viewing access to the revision, without loading it into the local image.

- **Load** installs the selected revision into the image, making it available for editing and execution.

- **Export** saves the currently selected revision in export format, for distribution to another Visual Smalltalk Enterprise user. Typically this format is only used if the other developer does not have access to your repositories.

- **Edit lock** allows you to lock or unlock access to this revision.
• *Filter...* displays the revision filter dialog. By checking selected check boxes, you can limit the revisions shown based on locking condition.

### Connecting to a Repository

Before you can load or open the modules in a repository, you must connect to it, as follows:

1. **Select Repository / Connect...** in the menu.

   The Connect Repository directory selection dialog opens.

   *Figure 4-26: Directory Dialog*

   ![Directory Dialog](image)

   - **current directory**
   - **drives**
   - **subdirectories**

2. Select the directory containing your repository, then click **OK**.

Repositories have normal directory names, and are not generally recognizable by name alone, so you have to know which directories contain repositories.

When you select a directory containing a repository, it is connected and the repository name is added to the repositories list. Note that the repository name is usually different from the directory name.

If you select a directory which does not contain a repository, an information dialog is displayed notifying you of the fact, and nothing is added to the Repository Browser repositories list.

Also, if the repository name in the selected directory is the same as an already connected repository, an error notifier dialog will inform you, and the repository will not be connected.
NOTE: This is the only dialog that accesses repositories by directory path. In all other Visual Smalltalk Enterprise tools and browsers you access repositories by name.

Creating a Repository

Before you can commit any modules to a repository, you must create the repository. Creating the repository creates its directory and special files required for managing the repository.

To create a repository:

1. Select Repository / New... in a Repository Browser.

   The Create Repository dialog opens.

2. Enter the required information, then click Create.

   The require information is:

   - Name: The repository name can contain spaces and punctuation, and can be any length. The name may not be the same as a connected repository.

   - Path: The directory path name for the new repository. You can type the path name or click SetPath... and select a directory in a directory dialog.

   - Type: The repository type is either PVCS or File-Based, which you choose from the drop-down list.

   - Locking: Choose either Optimistic or Pessimistic from the drop-down list.

See Creating a Repository on page 220 for more detailed information.
Once created, you cannot change the name, type, or locking policy for a repository. You can move it to a different location and then reconnect to it from the Repository Browser.

Buttons in a Create Repository dialog are:

- **Create** is enabled when all fields have been specified. Pressing this button creates the specified repository and closes the dialog. If the repository cannot be created, you are notified with an appropriate message and the dialog is not closed.
- **Cancel** closes the dialog without creating a repository.
- **Set Path...** opens a directory dialog that lets you select a directory by traversing the file system.

**Package Organizer**

The Package Organizer lets you examine the definitions within an open or loaded package in the order in which they will be initialized and reorder them as you require. You can also view the definitions for a loaded package in a Package Browser, but you can only view an open package in a Package Organizer.

Items that have been modified since you last committed them appear in italics.

**Opening a Package Organizer**

Under the **Smalltalk** menu from any window, select **Open -> Package Organizer** to open a Package Organizer. The Package Organizer is divided into four panes as shown in figure 4-28.
The **package list** appears in the upper left pane. In this pane, the names of all the packages loaded or open in the system are presented in alphabetical order. (The special package **unpacked** is always first, however.) The names of open packages are enclosed in parentheses.

The **definition list** appears in the upper right pane. This pane shows all definitions in the selected package, in the order in which they will be initialized when they are installed in an image.

Certain definitions, such as pools and classes, can be expanded or collapsed. These definitions contain other definitions: a class contains its methods or variables, and a pool contains its variables. Each such definition appears with a small square to its left. If the square contains a plus sign, the definition is collapsed, and the items it contains are not visible. If the square contains a minus sign, the definition is expanded, and the items it contains appear indented below it. Click the left mouse button once on these squares to toggle between expanded and collapsed definitions. Double-click to expand all sub items.

The **annotation list** appears in the lower left pane. This pane shows all the annotations defined for the selected definition or package.

**NOTE:** Annotations whose names begin with a period (".") cannot be modified or deleted.
The contents pane appears in the lower right pane. This pane is an editable text pane showing the source code or annotation associated with the selected package or definition.

**Working With Packages in the Package Organizer**

To view the definitions within a package, select a package in the package list, the top left pane of the window. Visual Smalltalk Enterprise displays the definitions within the package in collapsed form in the definition list to the right.

The Package menu contains functions that permit you to install packages into your image or open them for examination without installing them. From this menu, you can also commit revisions of a package, lock them so that others cannot change them, and perform other operations. This menu changes when you switch from an open package to a loaded one. For example, you close an open package, but you unload a loaded package.

**Package Organizer Package Menu**

The Package Organizer Package menu is the same as the Package Browser's Module menu, with the following additions or modifications:

The Package Organizer Package menu for a loaded package is shown below.

![Figure 4-29: Package Organizer Package Menu](image)

The Package Organizer Package menu is the same as the Package Browser's Module menu, with the following additions or modifications:
• \textit{New}... prompts you for the name of a new open package to add to your image. This is a different function from the menu item \textit{Module / New} in a Package Browser, which creates a loaded package or cluster.

• \textit{Close}... is available if the package is open. It prompts for confirmation before closing the package. If the package is loaded, this menu item is \textit{Unload}..., as in the Package Browser.

• \textit{Load} is enabled if an open package is selected. It opens a dialog to ask whether the module should be initialized, then loads the selected package. If a load conflict exists, it opens a Conflict Notification Dialog.

• \textit{Find Package}... is the same as the \textit{Find Module}... item in the Package Browser.

\section*{Organizing Definitions}

To \textit{view the definitions within a package}, select a package from the package list, in the top left pane of the window. Visual Smalltalk Enterprise displays the definitions within the package in collapsed form in the definition list to the right.

To \textit{expand a definition}, click once on the expansion indicator next the definition. The variables in a pool, or the methods and variables in a class, appear in an indented list below it. They can now be selected.

To \textit{collapse a definition}, click the left mouse button once on the expansion indicator next to the definition. The variables in a pool, or the methods and variables in a class, disappear from the list.

\textit{To expand or contract all sub items}, double-click on the expansion indicator next to the definition.

To \textit{view a definition's annotations}, select a definition in the definition list, the top right pane of the window. Visual Smalltalk Enterprise displays its annotations in the annotations list at the lower left, and its associated source code in the contents pane at the lower right. Annotations can now be added, deleted, or modified, and the source code can be edited.

To \textit{reorder definitions within a package}, select the definition you want to move, and move it using the appropriate menu items, as described below.

To \textit{move a definition from one package to another}, drag it from the definition list pane into the package list pane and drop it onto the package in which you want the definition to reside. The target
package appears highlighted with a light rectangular outline. The definition disappears from the definition list. If you select the package in which you dropped it, you see it at the end of the definition list for that package. You can then reorder it as desired.

**Package Organizer Definition Menu**

The *Definition* menu contains functions that permit you to create, remove, reorder, collapse, or expand definitions. The *Definition* menu items are:

- **New** opens a submenu from which you can create definitions for new classes, methods, and assorted types of variables. See the heading entitled *New Definitions* below for a description of the items on the *New* submenu.

- **Rename...** prompts you for a new name for classes, pools, and global variables. It is enabled if a global variable, pool, pool variable, class, instance variable, class variable, or class instance variable selected.

  If the global is part of an open package, you cannot name it the same name as any other global in that package.

  If the global is part of a loaded package, you cannot name it the same name as any other global in any package loaded in your image.

- **Delete** pops up a confirmation dialog. Confirming this operation causes the selected definition to be deleted from its package.

- **File Out...** is enabled if a definition is selected. It opens a file dialog to specify the name of a file in which to store the file-out representation of the selected definition.

The Package Organizer *Definition / Move* item brings up a submenu to change the order of the selected definition. It is enabled if a definition is selected.

- **Move -> Top** moves the selected definition to the top of the definition list.

- **Move -> Up** moves the selected definition above the preceding definition in the definition list.

- **Move -> Down** moves the selected definition below the following definition in the definition list.

- **Move -> Bottom** moves the selected definition to the bottom of the definition list.
• **Move to...** pops up a list of all packages open or loaded in your system, as appropriate. Select the package to which to move the selected definition. Confirming this operation causes the selected definition to be moved from its present package to the one you have specified. This operation has the same effect as dragging the definition and dropping it on the specified package name.

• **Copy to...** pops up a dialog prompting you for the name of the package to which to copy the selected definition. Confirming this operation causes a copy of the selected definition to be added to the package you have specified.

**NOTE:** The copy operation cannot create conflicts. Therefore, either the source or the destination package must be an open (unloaded) package.

• **Browse** brings up a submenu that allows you to browse senders, implementors, local senders, local implementors, and message for the selected definition. These submenu items are enabled when a method definition is selected, except for the Messages submenu item that is enabled if the definition is source-based.

• **Compare Versions** opens a Version Comparison Browser on the selected method.

• **Format** is used to format the selected text according to the conventions specified in Appendix B. It is enabled if a source-based definition is selected.

• **Filter...** displays the definition filter dialog. By checking selected check boxes, you can limit the definitions shown based on their modified state.

**New Definitions**

Selecting **Definition / New** in the Package Organizer opens a submenu from which you can create definitions for new classes, methods, and assorted types of variables. The **New** submenu items are:

• **Class...** opens a Class Definition Dialog to create a new class definition in the selected package.

• **Method...** opens a Method Definition Dialog to create a new method definition in the selected package. The class to which the method belongs does not need to be in the selected package.
• **Instance Variable**... is used to create a new instance variable definition for a class in the selected package. You are prompted for the name of the new instance variable. It is enabled if the selected definition is a class definition or a definition nested within a class definition.

• **Class Variable**... is used to create a new class variable definition for a class in the selected package. You are prompted for the name of the new class variable. It is enabled if the selected definition is a class definition or a definition nested within a class definition.

• **Class Instance Variable**... is used to create a new class instance variable definition for a class in the selected package. You are prompted for the name of the new class instance variable. It is enabled if the selected definition is a class definition or a definition nested within a class definition.

• **Pool Usage** lets you define access to pool dictionaries from a class in the selected package. You are prompted for the name of the pool dictionary. It is enabled if the selected definition is a class definition or a definition nested within a class definition.

• **Class Initializer** lets you define an expression to initialize the class. This expression is evaluated when the package is initialized. This expression can reference class variables and class instance variables directly. The pseudo-variable `self` is bound to the class. The class initializer can be evaluated in a Package Browser using `Module / Initialize` or `Global / Initialize`, or in a Package Organizer using `Package / Initialize`.

This menu item is enabled if the selected definition is a class definition or a definition nested within a class definition that does not already have an initializer.

• **Global Variable** is used to create a new global variable definition in the selected package. You are prompted for the name of the new global variable.

• **Pool** is used to create a new pool definition in the selected package. You are prompted for the name of the new pool. Use the menu item `Definition / New` -> `Pool Variable` to add elements to the new pool.

• **Pool Variable** opens a Pool Variable Definition Dialog to define a new pool variable for an existing pool. The pool need not be defined in the selected package.
• *Initializer* is used to create an ad hoc initializer. The procedure for creating this item is described in the next section of this chapter.

**Creating an Initializer**

In the Package Organizer, you can create an ad hoc initializer to contain code, such as platform-specific initialization, that is not associated with a class or variable. The initializer is executed when the package is initialized.

To create an *ad hoc* initializer:

1. From the Definition menu, execute the item *New -> Initializer*.  
   In the definition list, the words *empty initialization expression* appear.
2. In the contents pane, type the expression you want to execute when this definition is executed.
3. Save your work in the contents pane.

*Figure 4-30: Creating an ad hoc Initializer*

*TIP*: The first line of this expression appears in the definition list when you save it in the contents pane. It is therefore handy to start your expression with a brief comment that expresses its intent. Then you see this comment as you scroll through the definition list.
Annotating a Package

To view the annotations for a package in the Package Organizer, select the package. Its annotation labels appear in the annotation list. Select a label, and the text of the annotation appears in the contents pane, where it can be modified.

To view the annotations for a definition, select the definition. Its annotation labels appear in the annotation list. Select a label, and the text of the annotation appears in the contents pane, where it can be modified.

Package Organizer Annotation Menu

The Annotations menu contains functions that permit you to add, rename or remove an annotation. The Annotations menu items are:

- **New...** prompts you for a new annotation label. Type the label in the field provided and press Enter or click OK. The new annotation appears selected in the annotation list. Type the annotation text in the contents pane, and save the text.

- **Rename...** prompts you for a new name for the selected annotation. If the name you specify already exists, you are notified and no modification occurs. This item is enabled if an editable annotation is selected.

- **Delete** pops up a confirmation dialog. Confirming this operation causes the selected annotation to be deleted from the definition. This item is enabled when an editable annotation is selected.

- **Filter...** displays the annotation filter dialog. By checking selected check boxes, you can limit the annotations shown based on whether or not they are editable.

Annotations beginning with a period cannot be edited. You can drag editable annotations from an annotation list, but you cannot drop items on annotations.
Cluster Organizer

The Cluster Organizer lets you view and edit specifications in both loaded and open clusters.

Opening a Cluster Organizer

Select Smalltalk / Open / Cluster Organizer to open a Cluster Organizer. The Cluster Organizer is divided into five panes as shown below.

Figure 4-31: Cluster Organizer

The upper left pane of a Cluster Organizer is the cluster list pane that shows all clusters known to the system, whether loaded or open. Open cluster names are enclosed by parentheses.

When you select a cluster from the cluster list, the specification list in the upper right pane contains a list of specifications in the selected cluster.

The annotation list in the lower left pane displays the names of annotations defined for the selected cluster (if no specification is selected) or the selected specification.

When an annotation name is selected, the content pane in the bottom middle pane shows the value of the annotation. Otherwise, if a specification is selected, its comment is shown. If only a cluster is selected, the description of the cluster is shown. If there are no selections, the content pane is blank.
The bottom right pane shows the *conditional inclusion flags* associated with the selected specification. This pane also has two buttons:

- *Add* prompts you for the name of a conditional inclusion flag and adds it to the list. It is enabled when a specification is selected.
- *Delete* is enabled when a flag is selected. It removes the selected flag from the list.

**Cluster Organizer Cluster Menu**

The Cluster Organizer *Cluster* menu for a loaded cluster is shown below.

The Cluster Organizer *Cluster* menu is similar to the Package Browser’s *Module* menu and the Package Organizer’s *Package* menu. The Cluster Organizer *Cluster* menu is the same as the *Package* menu documented in *Package Organizer Package Menu* on page 143, with the following exceptions:

- *Convert > Group, Configuration, Cluster* automates the process of either maximizing or minimizing cluster specifications. For more information, see *Converting Between Clusters, Groups, and Configurations in chapter 5.*
- *Commit All* does a bulk commit on all modified modules in the currently selected cluster. A dialog opens listing the
modified modules, and allowing you to enter a single revision comment.

- *Find Cluster...* is the same as the *Find Package...* item in the Package Organizer.

**Cluster Organizer Specification Menu**

The Cluster Organizer *Specification* menu is shown below.

*Figure 4-33: Package Organizer Specification Menu*

The Cluster Organizer *Specification* menu items follow. If any move or delete operation would cause any module to be unloaded, you are asked for confirmation before the operation is performed. For example, deleting a specification unloads the module it specifies if either the specification is the last complete specification or the remaining specifications together are not complete. If an operation cannot be completed because of conflicts, a Conflict Notification dialog is opened.

- *New...* is enabled if a cluster is selected. This creates a new specification, brings up a Specification Editor, and adds the edited specification to the selected cluster.

- *Delete* is enabled if a specification is selected. It removes the selected specification from the selected cluster. If the specification is loaded but its deletion requires unloading the resolved module, you are prompted for confirmation.
• **Uncluster** is enabled if a loaded specification is selected. It moves the selected specification from the selected cluster to the top level cluster.

• **Move To...** is enabled if a specification is selected. It prompts you to select the name of a cluster from a scrollable list, then moves the selected specification to that cluster unless the operation was cancelled. If the move operation would cause any modules to be unloaded, you are asked for confirmation.

• **Copy To...** is similar to **Move To...**, except that the specification is copied instead of moved.

• **Edit...** opens a Specification Editor on the selected specification.

• **Complete** is enabled if a loaded specification is selected that is incomplete. It completes the specification so that it fully specifies the loaded module.

• **Complete All** is enabled if a loaded cluster is selected. It completes all specifications in the selected cluster.

• **Minimize** is enabled if a specification is selected. It minimizes the selected specification, so that it specifies its module by name only.

• **Minimize All** makes all specifications in the selected cluster minimally specify their modules (by name only). It is enabled when a cluster is selected.

• **Optional** is a toggle menu item that is enabled if a specification is selected. It specifies whether the selected specification is optional. If the cluster is loaded, this may require loading or unloading the specified module. If the operation cannot be performed, you are notified.

• **Filter...** displays the specification filter dialog. By checking selected check boxes, you can limit the specifications shown based on their modified state, repository entity type, and whether or not the specifications are resolved.

**Annotating a Cluster**

To view the annotations for a cluster in the Cluster Organizer, select the cluster. Its annotation labels appear in the annotation list. Select a label, and the text of the annotation appears in the content pane, where it can be modified if it is not a read-only annotation.
To view the annotations for a specification, select the specification. Its annotation labels appear in the annotation list. Select a label, and the text of the annotation appears in the content pane, where it can be modified.

**Cluster Organizer Annotation Menu**

The *Annotations* menu contains functions that permit you to add, rename or remove an annotation. The Cluster Organizer Annotation menu is the same as that documented in the section entitled *Package Organizer Annotation Menu* on page 149.

**Specification Editor**

The Specification Editor lets you edit information associated with open or loaded specifications. Using the editor, you can edit the list of conditional inclusion flags and specify whether inclusion is optional. For open specifications, you can change the repository, module type, module name, and revision number.

**Opening a Specification Editor**

To open a Specification Editor, open a Cluster Organizer and select a cluster. Then either

- select a specification in the specification list and select *Specification / Edit...*, or
- select *Specification / New...*

A specification editor will open as shown in figure 4-34. If the specification is a group or configuration, the editor has a “Override Usage” check box. Check this box to gain access to the other fields.
Editing a Specification

As shown above, a Specification Editor shows the repository name, entity type, entity name, and revision number.

Each item is displayed in a drop down list box, which displays the options for that item. You can clear an item from the specification by selecting its text and pressing the Delete key. If you clear all items except Entity Name, you create a minimum specification. If you select entries for each item, you create a maximum specification. Any combination in between is also possible, except that Entity Name must be specified in order to save the specification.

The current conditional inclusion flag list is shown in the pane at the lower left. You can add and delete entries in this list with the Add and Delete buttons to the right. You can also rename a flag by selecting the flag name in the list box, entering a new name in the entry field, then clicking Rename.
The checkbox at the lower left specifies whether the specification is optional.

Click OK to check the edited specification for consistency with all loaded modules. If problems are detected, you are notified. If the specification is consistent, the editor closes and the specification is modified in the original Cluster Organizer window as necessary.

**Inclusion Context Editor**

A Conditional Inclusion Context editor allows you to specify the set of conditional inclusion flags that determine which modules are included when processing conditional specifications. The flags serve as a filter so you can include platform-specific functionality in separate modules or optional behavior such as debugging support.

When you choose Smalltalk / Open -> Inclusion Context Editor, a Conditional Inclusion Context editor opens, showing the current set of flags defined for the image.

By default, no conditional inclusion flags are defined and the Conditional Inclusion Context editor opens with an empty list.

*Figure 4-35: An Example Conditional Inclusion Context Editor*

![Conditional Inclusion Context Editor](image)

The figure above shows an example with the flag **Debug** defined. If a package defined the conditional flag **Testing**, it would not be loaded with these flags in place.

Buttons provide the following functionality:

- **Add** prompts you for the name of a flag to add to the list.
- **Delete** removes the selected flag from the list. It is disabled if no flag is selected.
- **OK** changes the list of flags in the image to match the list. If completing this operation requires modules to be loaded or unloaded, you are prompted for confirmation before the
operation is carried out. If you do not confirm, the Conditional Inclusion Context editor remains open. If you do confirm the operation and there are any conflicts, a Conflict Notification dialog is opened.

- **Cancel** closes the editor without changing the list of flags defined in the image.

See *Using Conditional Inclusion Flags* on page 198 for an example that explains how conditional inclusion flags might be used.

---

**History Browser**

The bottom right button of many development tools specifies the version of the definition you are examining. To browse all versions of the selected definition, click on this button. A History Browser opens on the selected definition. It is divided into two panes as shown in figure 4-36.

**Figure 4-36: History Browser**

The **version list** appears in the top pane. In this pane, the labels of all the versions of the definition available in your image appear in reverse chronological order. The current version is first in the list; it appears in italics if it has been modified since the package it belongs to was last committed.

The **contents** appear in the lower pane. This pane is an editable text pane showing the source code associated with the selected version.
To view the source code associated with a previous version, select the version. Its source code appears in the contents pane.

**History Browser Version Menu**

The *Version* menu contains functions that permit you to install a previous version as the currently active version, or to update the History Browser. The *Version* menu items are:

- **Install** makes the selected version of the definition the currently active version in your image. If you revert to a previous version, a copy of that version is added and appears at the top of the list of versions, indicating that it is the current active one. The previously active version is now in second place, and the version you reverted to can also still be found in the same place in the sequence it occupied before.

- **Compare All** opens a Version Comparison Browser on the versions.

- **Update** tells the History Browser to recompute the versions of the definitions it can show. You must do this if you have modified a definition in another tool. Otherwise, the History Browser does not reflect the current state of your image.

**Browsing Specific Definitions**

Visual Smalltalk Enterprise lets you browse selected groups of definitions. Executing *Senders*, *Implementors*, *Local Senders*, *Local Implementors*, or *Browse References* from a variety of menus; browsing references to unresolved variables; or browsing modified definitions all open a browser, the Definition Group Browser, on the definitions that meet the specified criterion. The title for this window reflects its purpose. For example, if you execute `Smalltalk / Browse -> Modified`, the title of the window is *Modified Definitions*.

**Definition Group Browser**

The Definition Group Browser is divided into two panes as shown in figure 4-37.
The top pane is a definition list, showing the definitions that met the specified search criteria.

The bottom pane is the contents pane, showing the source code associated with the selected definition.

**Definition Group Browser Definition Menu**

The Definition Group Browser has one extra menu, the Definition menu, as shown in the above figure. The Definition menu items are:

- **Delete** removes the selected definition from the image after confirmation from the user. The package that contained the definition is modified. This item is enabled if a definition is selected.

**NOTE:** Before you delete a method, browse senders of the message, so that you can be sure that its deletion will not cause a problem.

Before you delete a global or a variable, browse references to it, so that you can be sure that its deletion will not cause a problem.

If you delete a class initializer, the class will not be initialized when the package is next initialized, unless you make other provisions to do so.
• **Remove From List** removes the selected definition from the list in the browser. It has no effect on the image. It is enabled if a definition is selected.

• **Senders** is used to open another Definition Group Browser on all senders of the message represented by the selected definition. It is enabled if a method definition is selected.

• **Implementors** opens another Definition Group Browser on all implementors of the message represented by the selected definition. It is enabled if a method definition is selected.

• **Local Senders** and **Local Implementors**. Executing either of these has the same effect as their namesakes in this menu, described above, except that the scope of the search is limited to the class in which the selected definition is defined, and its subclasses.

• **Messages** opens a Message Browser on all senders or implementors of methods represented by the messages sent in the selected definition.

• **File Out...** pops up a file dialog letting you specify the file in which to place the definition you have selected in the definition list. The selected definition is then filed out in file-in format.

• **File Out All...** pops up a file dialog letting you specify the file in which to place all definitions in the definition list. They are then filed out in file-in format.

• **Compare Versions** opens a History Browser on the selected method.

• **Compare All** opens a Definition Comparison Browser on the list of definitions.

• **Format** is used to format the selected text according to the conventions specified in Appendix B. It is enabled if a source-based definition is selected.

• **Filter...** displays the definition filter dialog. By checking selected check boxes, you can limit the definitions shown based on their modified state.

**Message Browser**

The Message Browser lets you browse all the messages sent by a selected source definition. It can be accessed from the **Method** menu in the Package Browser, or the **Definition** menu in the Definition Group Browser.
In addition, the *Debugger* menu in a debugger also contains a *Messages* menu item to let you access a Message Browser for the method selected in the debugger.

The Message Browser is divided into three panes as shown in figure 4-38.

*Figure 4-38: Message Browser*

<table>
<thead>
<tr>
<th><a href="#">Return</a></th>
<th><a href="#">Forward</a></th>
<th><a href="#">Next</a></th>
<th><a href="#">Previous</a></th>
</tr>
</thead>
</table>

The top left pane is a *message list* showing the messages sent by the selected definition.

The top right pane is a *message list*, showing either the methods that send or implement the selected message, depending on which radio button is selected.

The bottom pane is the *contents* pane, showing the source code associated with the selected method.

The Message Browser has one extra menu, the *Definition* menu. It is the same as the *Package Organizer Definition Menu* on page 145.

**Comparison Browsers**

Visual Smalltalk Enterprise lets you compare two versions of a definition such as a method, or two revisions of a package.

**Version Comparison Browser**

To compare two versions of a method, open a Version Comparison Browser. From the Package Browser, under the *Method* menu, select *Compare Versions*. The Version Comparison
Browser is divided into four panes as shown in figure 4-39, and is arranged as if two History Browsers were positioned side-by-side in a single window.

Figure 4-39: Version Comparison Browser

The two top panes are both version lists, as in a History Browser. In these panes, the labels of all the versions of the definition appear in reverse chronological order. The current version appears in italics if it has been modified since the package it belongs to was last committed.

The bottom two panes are contents panes, also as in a History Browser. These panes show the source code associated with the selected version, including non-printing characters. They are not editable although you can copy selections from them.

To compare two versions of the method, select one version in the left version list and select another version in the right. The associated source code appears in the corresponding contents pane. Differences between the two versions are underlined (in red on a color display, although it can be changed using the preferences).

Version Comparison Browser Left and Right Menus

The Version Comparison Browser contains the menus Left and Right. These menus operate on the left and right panes, respectively. They each contain the same item, which is enabled if a previous version is selected in the respective pane.

- **Install** causes the selected version of the method to become the currently executable version loaded in the image. The new version is added at the top of the list of versions.
**Version Comparison Browser Compare Menu**

Another menu added by the Version Comparison Browser is the *Compare* menu. It contains the following two items:

- *Preformat* is a toggle menu item. When checked, method source is formatted before computing the source comparison, which effectively ignores formatting differences.

- *Update* tells the Version Comparison Browser to recompute the versions of the definitions it can show. You must do this if you have modified a definition in another tool. Otherwise, the Version Comparison Browser does not reflect the current state of your image.

**Definition Comparison Browser**

A Definition Comparison Browser is very similar to a Version Comparison Browser, but gives you more flexibility to compare a wider range of definition types.

Choose *Smalltalk / Open -> Definition Comparison Browser* to open an empty Definition Comparison Browser. You can copy definitions from the definition list of a Package Organizer, Package Browser, or any other window that contains definitions.

Choose *Definition / Compare All* from a Message Browser to open a Definition Comparison Browser on its list of definitions.
For example, figure 4-40 shows a window in the upper left opened on all method selectors that begin with the string 'add'. In this window, you can select a specific method and then choose Definitions / Compare Versions to open a Version Comparison Browser (shown at the bottom left) on all versions of that method for that particular class. If you choose Definitions / Compare All, you can view all implementations of that message as shown in the Definition Comparison Browser at the right. In this browser, you can compare the implementations of a method from two different classes. You can also copy other similar methods from other classes by dragging them from another window and dropping them in one of the panes at the top, if you want to compare the source for a method that has a different name.

A Definition Comparison Browser is not restricted to method comparisons—you can compare any definitions. For example, figure 4-41 shows a Definition Comparison Browser open on the definitions of two different classes. The class definitions were selected in a Package Browser by dragging a class name and dropping it on one of the upper panes in the Definition Comparison Browser.
Definition Comparison Browser Compare Menu

The *Compare* menu for a Definition Comparison Browser contains the following item:

- **Preformat** is a toggle menu item. When checked, method source is formatted before computing the source comparison, which effectively ignores formatting differences.

Definition Comparison Browser Left and Right Menus

The Definition Comparison Browser’s *Left* and *Right* menus are the same as that documented in *Definition Group Browser Definition Menu* on page 159, except that these menus do not contain the *Compare All* or *Format* items.
Package Comparison Browser

To compare two revisions of a package, or to compare two packages, open a Package Comparison Browser. From the Package Browser, under the Module menu, select Compare. From a Package Organizer, select Package / Compare.

The Package Comparison Browser is divided into six panes as shown in the following figure.

The two top panes are both package lists, showing the names of all packages that are currently loaded or open. The names of open packages are enclosed in parentheses. No differences are computed until a selection is made in both package lists and the Compare button is clicked.

The middle two panes are definition lists. They contain lists of all the definitions in the selected revision of each package.

Differences are highlighted in one of two ways. Definitions that appear in one revision but not the other appear in boldfaced text (green boldfaced on a color display, by default). Definitions that appear in both revisions but are different appear in italics (red italics on a color display, by default). These colors can be changed using the preferences. See the Color Preference Editor on page 65 for more information.
The bottom two panes are contents panes. These panes are text panes showing the source code associated with the selected definitions. Editing text in these panes is not allowed, but selecting and copying text is supported. Differences between the two panes are underlined (in red on a color display, by default).

To compare two revisions of the package, select one revision in the left revision list and select another revision in the right. The associated definitions appear in the corresponding definition lists. Click the Compare button. Differences between the two revisions appear highlighted.

To compare two packages, select one revision of one package in each list. Click the Compare button. Differences between the two versions appear highlighted.

To update both panes, double-click a definition. If the selected item appears in both lists, it is selected and differences are displayed. If the selection does not appear in the second list, the second list is updated so that it has no selection and its contents pane is cleared. In either case, item expansion is supported in the second list so that the displayed hierarchy is kept in synch.

**Package Comparison Browser Buttons**

A row of buttons separates the package lists and the definition lists in the Package Comparison Browser. These buttons modify the appearance or selection in the definition lists. They have the following behavior:

- **Next** finds the next definition that is bold or italic (new or changed) and simulates a double-click to update both panes at the same time. The search begins just after the current selection in the right or left pane, depending on whether you click the right or left button. If nothing is selected, the search starts at the beginning of the list. It proceeds downward.

- **Previous** finds the previous difference. It is like Next, except the search proceeds up in the list.

- **Compare** computes the package comparison. This button is enabled whenever a selection is made in either list that causes the highlighting in the definition lists to be removed. When it is clicked, the comparison is computed and the button disabled.
Package Comparison Browser Menus

The Package Comparison Browser includes Left, Right, and Compare menus. The Left and Right menus are identical, but control the left and right sides of the browser. The options in the menus are:

Left/Right Menus:

- **New...** opens a dialog prompting for the name of a new package.
- **Rename...** opens a dialog prompting for a new name for the package.
- **Close** closes the selected package (option is displayed if the package is open).
- **Unload** unloads the selected package (option is displayed if the package is loaded).
- **Load** loads the selected package (option is available if the package is already open).
- **Load...** opens a load package dialog.
- **Migrate...** opens a migrate package dialog.
- **Open...** opens an open package dialog.
- **Commit...** opens a commit package dialog.
- **Find Package** opens a dialog prompting for a package name, then searches for and selects that package.
- **Lock** allows you to lock or unlock the selected package.
- **Filter...** displays the package filter dialog. By checking selected check boxes, you can limit the packages shown.

Compare Menu:

- **Preformat**, if checked, the methods are formatted before being compared.
- **Update** refreshes the list pane display, if changes occurred somewhere else in the system.
- **Filter...** displays the definition filter dialog. By checking selected check boxes, you can limit the definitions shown to any combination of changed, new, and unmodified.
Build Library Dialog

The Build Library dialog provides a convenient way to build a customized Smalltalk link library for a module.

When you select a module and choose Module / Build Library... in the Package Browser, the Build Library dialog opens. The following figure shows a Build Library dialog for the Map Browser Configuration cluster from the tutorial.

Figure 4-44: Build Library Dialog

A check box lets you specify whether to include source with the SLL file.

The buttons in a Build Library dialog are:

- **Build...** opens a file dialog to specify the name of the SLL file. After a name is chosen, the SLL is written.
- **Customize...** closes the Build Library dialog and opens a Library Build Script workspace with code you edit to customize the build procedure.
- **Cancel** closes the dialog without building a library.

An example of a Library Build Script is shown in figure 4-45.
Figure 4-45: A Build Script for the Map Browser Configuration Cluster

```
<table>
<thead>
<tr>
<th>File Edit Smalltalk Options Help</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Evaluate the following to create a library for Map Browser Configuration&quot;:</td>
</tr>
<tr>
<td>Smalltalk:LibraryBinder bindTo: 'V10c30';</td>
</tr>
<tr>
<td>module := TeamVInterface current: moduleNamed: 'Map Browser Configuration'.</td>
</tr>
<tr>
<td>info := TeamLibraryInformation new.</td>
</tr>
<tr>
<td>info name: 'MapBroCo'.</td>
</tr>
<tr>
<td>info targetDirectory: 'G:TEAMMV'.</td>
</tr>
<tr>
<td>info sourceInFile: 'forSourceSeparate'.</td>
</tr>
<tr>
<td>info includeSourceTest [def] true.</td>
</tr>
<tr>
<td>info module: module.</td>
</tr>
<tr>
<td>info description: 'Repository: TeamV Tutorial (PVCS)'.</td>
</tr>
<tr>
<td>Revision: 3.0 (locked by Roxie)</td>
</tr>
<tr>
<td>Author: Juanita</td>
</tr>
<tr>
<td>Cluster: Map Browser Configuration</td>
</tr>
<tr>
<td>Revision Comment: TeamV Release 3.0'.</td>
</tr>
<tr>
<td>info staticInitializationTest [def] false.</td>
</tr>
<tr>
<td>builder := info makeLibrary.</td>
</tr>
</tbody>
</table>
```

For more information about build scripts, see *Building Smalltalk Libraries* on page 239.

**Unresolved References Browser**

To browse all unresolved references in your system, choose Smalltalk / Browse -> Unresolved. This opens a browser, as shown in figure 4-46.

Figure 4-46: Inspector on Unresolved References

The names of all unresolved variables appear in the left list; the values appear on the right when a variable name is selected.
Unresolved References Browser Variable Menu

The browser has one new menu, Variable, which contains the following items:

- **Variable / New**... opens a dialog that lets you specify the name of the new variable to add. Type in the new variable name and press OK. You can now define methods and initializers that reference the new variable without getting compilation errors. This temporarily allows new code to refer to classes and other globals that you have not yet defined.

**NOTE:** You should be extremely cautious about adding unresolved variables. They do not belong to any package so they are not saved when you commit a package. They are to be used only as a temporary measure when the package defining them is not yet available.

- **Variable / Delete** removes the selected variable from the pool of unresolved variables, as long as no references to it exist in the image. If references exist, the selected variable is not removed and you are asked if you want to open a Definition Group Browser on the definitions that reference the unresolved variable. This menu item is disabled if no variable name is selected.

- **Variable / Browse References** opens a Definition Group Browser on all the definitions in the image that refer to the selected unresolved variable, if any. See the section entitled Browsing Specific Definitions on page 158 for more information on the Definition Group Browser. This menu item is disabled if no variable name is selected.

- **Variable / Inspect** opens an inspector on the value of the selected variable. This menu item is disabled if no variable name is selected.

- **Variable / Delete Unreferenced** removes all unreferenced, unresolved variables from the pool of unresolved variables. This is equivalent to selecting each variable and performing a Delete operation, except that there is no notification for variables that are still referenced.

- **Variable / Update** tells the inspector to recompute the collection of unresolved references when unresolved references are encountered. You must do this when you add or delete unresolved references in another tool. Otherwise, the inspector does not reflect the current state of your image.
Unresolved variables do not belong to any package. Therefore, when you commit a package, you are not saving any definition for the unresolved references. For this reason, it is a good idea to check for and resolve unresolved references before you commit your packages or generate a build script for your application. It is also a good idea to check for unresolved references after you load an unfamiliar package.

**Access Control Editors**

In chapter 2 we described how to set up an initial Access Database and how to add users to the database. There are a few tools for using the database to control access to your repositories and modules. Some typical uses of these tools are described under *Controlling Repository Access* in chapter 5.

These additional tools are all accessible in the Repository Browser.

**Repository Access Editor**

The Repository Access Editor is used to assign access rights to an entire repository. Access rights can be assigned or denied to users or groups for these activities:

- *Read* access is required for a user to be able to connect to the repository.
- *Create* access is required for a user to be able to create entities in the repository.
- *Destroy* access is required for a user to be able to delete an entity from the repository.
• *Administration* access is required for a user to assign access rights, as well as to destroy the repository itself.

*Figure 4-47: Repository Access Editor*

To open the Repository Access Editor, select a repository in the Repository Browser, then select *Repository / Edit Access*. A browser like the one shown above is opened.

When the *With Permissions* radio button is selected, only users and groups for which access rights have been assigned are displayed. To assign rights to a user or group that is not in the list but is in the database, select the *All* radio button.

To assign rights, select a user or group and mark the check boxes for the access rights you want this user or group to have. When you have selected the access rights, click *OK*.

**Repository Entity Access Editor**

The Repository Entity Access Editor is used to assign access rights to a repository entity. Access rights can be assigned or denied to users or groups for these activities:

- *Read* access is required for a user to be able to open or load a revision.
- *Create* access is required for a user to be able to create new revisions of an entity. The right to create revisions can be limited to creating only branch revisions.
- *Destroy* access is required for a user to be able to delete a revision from the module.
• *Administration* access is required for a user to assign access rights to the entity.

**Figure 4-48: Repository Entity Access Editor**

To open the Repository Entity Access Editor, select a repository and entity in the Repository Browser, then select *Entity / Edit Access*. A browser like the one shown above is opened.

When the *With Permissions* radio button is selected, only users and groups for which access rights have been assigned are displayed. To assign rights to a user or group that is not in the list but is in the database, select the *All* radio button.

To assign rights, select a user or group and mark the check boxes for the access rights you want this user or group to have. When you have selected the access rights, click *OK*. 
This chapter explains how to perform a variety of team programming procedures in step-by-step fashion. In addition to how, it also provides guidance on when and why you might need to perform these procedures.

You do not need to read this chapter from beginning to end. Instead, consult it as a reference for the answer to a particular question. Although occasional answers may refer to other answers, in general we have tried to provide all the information you need for a given task in one place. However, for those who want to read this chapter, or sections of it, straight through, the questions have been organized according to the phases of a development project.

The first section, *A Development Process Scenario*, presents an overview of the process of using Visual Smalltalk Enterprise to develop and deliver an application by a team. It outlines a sequence of steps a team might follow, designed to give you a structure for you to develop your own customized software development process.

The next section, *Using Packages*, discusses a variety of design issues and explains how you can use packages, initializers, and other items specific to Visual Smalltalk Enterprise as aids to good design.

*Using Clusters* and *Working with Clusters and Specifications*, explain how clusters can be used to group a variety of packages together and control which packages are loaded based on flags you can set in your environment.

The next two sections cover two main aspects of implementation: managing changes and working as a team.

*Using Revisions to Keep Track of Changes* explains how and when to use the mechanisms available for keeping track of changes: versions of certain definitions, revisions of packages, and repositories. In addition, the section discusses when you should open someone else’s package for examination, as opposed to loading it, and also explains how to integrate code from more than one revision or programmer.
Working with Repositories describes the two kinds of repositories and other repository management issues such as how to create, backup, and choose versioning policies for repositories.

Teamwork describes some of the every-day mechanics of loading packages and clusters, sharing code with colleagues who may or may not have Visual Smalltalk Enterprise, and resolving conflicts between modules.

Finally, Delivering An Application explains what you must do so that others can enjoy the fruits of your labors.

A Development Process Scenario

The goal of Visual Smalltalk Enterprise is to create an environment through which you can reliably create, deliver, and maintain Smalltalk applications over extended periods of time. Visual Smalltalk Enterprise provides

- tools to support programming teams in structuring and coordinating their work, and
- the ability to reliably recreate an application from source code.

Each organization has different goals and procedures, and each team needs the flexibility to work according to their own style. To work most efficiently with these tools, you need to understand how they were designed to operate and how to integrate that process in with your own organization’s software development process.

Here is an overview of the process of developing and delivering an application by a small team (5 members) using Visual Smalltalk Enterprise. It is by no means the only way to use Visual Smalltalk Enterprise, nor the best for your team or your organization. We encourage you to customize this scenario to meet your needs. You may have more repositories, need to restrict access, or have a much more formal process.

Here’s one possible software development process:

1. The system administrator installs and configures Visual Smalltalk Enterprise on each team member’s workstation.
2. The system administrator configures the access database, master configuration file, and local configuration files, making sure each person’s user name is set up correctly.
3. The corporate Smalltalk librarian sets up a shared repository on the network file system with clusters and packages.
containing additions and extensions to the base development environment as appropriate for the company.

4. The team leaders set up a shared repository on the network file system for application development, configuring it as a PVCS repository with optimistic locking.

5. Team members divide the work into a number of small packages. Most packages consist of entire classes, but some cross class boundaries. Each package has a designated owner. A package owner can change over time, and the package boundaries may also change as needed, but at any one time, every one on the team can determine who owns each package.

6. The team establishes a set of style and naming conventions, policies regarding when revisions are committed to repositories, and the information that needs to be included in the module comments.

7. The team integrator sets up a number of common clusters that provide convenient cluster groupings. Team members can also build clusters, but the integrator is responsible for maintaining the common cluster groups.

8. The team establishes guidelines for committing modules to the common repository (whether they can commit code that is untested, ready for review, but not tested, etc.) and commenting modules to reflect the status of the module.

9. Team members develop code. They sometimes choose to commit checkpoints to local repositories, but when they want to share their code, they commit their modules to the common repository.

10. If a team member discovers that a change needs to be made to another person’s package, a request for the change can be made so that the package owner is responsible for making the change and committing it to the repository. If that is not feasible, the requestor makes the proposed change and commits the module to a branch, indicating that this is a requested change, and notifies the owner of the package. The owner can approve the requested change or make other changes as necessary, committing to branches as needed. When the owner of the module and the requestor agree, the owner commits the module to the main line and notifies the integrator.
11. When team members commit a package to the common repository that should be included in the weekly build, they notify the integrator with the revision number of the module.

12. The integrator does daily integrations, loading modules in her current integration image. If a conflict is discovered, team members are notified and packages are modified as necessary. Daily integrations should only take a few minutes each day.

13. The integrator does weekly builds, using the latest modules that team members have submitted for the build. Each weekly build is built in a clean Visual Smalltalk Enterprise image (or perhaps one that has been saved with the base company library package) to ensure that no unpackaged definitions, unresolved variables, or other development artifacts are lurking. A script is run to determine whether there are any revisions that are newer than those in the build. If there are, discrepancies are noted and resolved—just because a newer revision has been committed doesn’t mean that it should necessarily be included in the build, but it does raise a question that the integrator needs to resolve.

14. Each weekly build is tested. When the build passes, the integrator records a complete specification for each revision of each module in the master configuration cluster and commits it. This guarantees that the build can be recreated if necessary.

15. The integrator places the new weekly build image in a common directory.

16. When the weekly build image is available, each team member makes a copy, connects to any local working repositories, loads personal configuration files, and continues work.

17. When the application is ready to deliver, the integrator prepares a Smalltalk library file or a runtime application built from the master configuration cluster.

**Installing Unpackaged Code**

During normal development you load source code from modules, as described throughout this manual. There are two other formats that can be used for sharing Smalltalk code with developers:

- File-in format is a portable code format between Visual Smalltalk and Visual Works development environments
- Smalltalk link libraries are a Visual Smalltalk format, portable across operating system platforms
While you can use file-in format for sharing code with other developers using Visual Smalltalk, it is particularly useful for sharing code with developers using Visual Smalltalk and VisualWorks.

Instructions for creating Smalltalk libraries and file-in format files (using File-out menu items) are provided elsewhere in this manual. In this section we cover a few points necessary for installing these files and bringing their code into your code module structure.

The relationship between file-ins, modules, the various menu items available, and conflict-checking in the image is diagrammed in figure 5-1.

**Figure 5-1: Bringing Code Into Your Image**
Importing File-in Format Code

With file-in format code you have two installation options:

- Installing file-in code installs the code in your image without checking for conflicts. Use either the File / Install... or the Smalltalk / File It In menu item to install file-in code.

- Loading file-in code treats the file like a package, and checks for conflicts while installing the code. Use the Module / Load menu item in the Package Browser or Package / Load in the Package Organizer to load file-in code.

Load or Install... loads the code into a package named “From FileName,” where FileName is the name of the file that was filed in.

File It In loads the code into a package named “From Anonymous Stream.”

Differences Between Installing and Loading File-Ins

When you install file-in format code, using either File / Install or Smalltalk / File It In, the code is installed without checking for conflicts with previously defined items, regardless of the settings in the Conflict Preference Editor. If the code filed in defines an item that is already in the image, the item in the image is replaced.

If the system encounters an error, the file-in process may quit and open a notifier. If possible, it writes a message to the Transcript and continues.

The items in the “From ...” package are treated as definitions. If a definition from the file-in existed in the image prior to being filed in, the new definition is shown as a modification to the original package rather than as a part of the file-in package. You can use filtering to focus on modified definitions.

When you load file-in format code using Module / Load (or Package / Load in the Package Organizer), the code is installed as if it were contained in a package, and it is checked for conflicts.

Procedures for Importing File-In Code

To install file-in code using File / Install:

1. Select File / Install in any Visual Smalltalk Enterprise tool.

   A standard file selection dialog is displayed.

2. Select the file containing the file-in format code, and click OK.

The code is loaded into a package named “From FileName,” where FileName is the name of the source file.
To install file-in code using Smalltalk / File It In:

   
   A standard file selection dialog is displayed.

2. Select the file containing the file-in format code, and click OK.
   
   The file is opened in a workspace.

3. Select the code text in the workspace, then select Smalltalk / File It In.
   
   The code is loaded into a package named “From Anonymous Stream.”

To load file-in code:

1. Select either:
   
   - Module / Load... in the Package Browser.
   - Package / Load... in the Package Organizer.
   
   The Load Package dialog is displayed.

2. Click the File... button in the Load dialog.
   
   A standard file selection dialog is displayed.

3. Select the file containing the file-in format code, and click OK.
   
   The selected file is checked for conflicts and loaded in a new package named “From FileName.”

   If conflicts are discovered, follow the resolution procedure described in Appendix A.

Installing Code from a Smalltalk Library

When you load code from a Smalltalk library, one or more clusters and packages are created. User generated SLLs create only a single package.

Modules loaded from SLLs are placed in the From Libraries cluster. Each SLL produces a single package, named “From Library FileName,” where FileName is the name of the SLL file.

If a conflict occurs with a definition already in the image, an additional package is created that contains the conflicting definitions from the library, plus a containing cluster. Definitions in the original packages are marked as modified.

The procedure for installing Smalltalk libraries is discussed elsewhere in the Visual Smalltalk Enterprise documentation, and so is not repeated here.
Organizing Imported Code into Packages

Once you have installed or loaded code from a file-in or an SLL, you can restructure into modules.

To reorganize the imported code:
1. Open a Package Organizer or Package Browser.
2. Locate the package containing your imported code.
   The package is called “From FileName,” “From Library FileName,” or “From Anonymous Filestream,” depending on the source of the code (refer to the preceding sections for details).
3. Create new packages and reorganize the definitions as you require.
   You may also simply rename the package and commit it to a repository.

For more help in structuring your application, see Dividing an Application into Packages below.

Using Packages

This section discusses the following topics:
- Dividing an application into packages.
- Creating a package.
- Reorganizing packages.
- Adding code to a package.
- Reordering definitions.
- Using categories.
- Restructuring a class hierarchy with packages.
- Initializing classes.
- Using ad hoc initializers.
- Extending the base class library.

Dividing an Application into Packages

In order to deploy teams of developers effectively, large and complex applications must be organized into modular units called packages. Packages are intended to be:
- The basis of a manageable work assignment for one developer
- Units of discrete functionality
- Units to distribute to other developers or users
• Independently maintainable units that isolate the effects of change
• Units whose revisions you can archive independently

Deciding how to divide the work into packages is a design decision. A package is added or removed from an application in its entirety, so those definitions likely to be used independently should not be placed in a package with others.

A package can contain just a single class—even a single method or variable. Often, however, a package consists of more than one class. A variety of heuristics can be used to divide an application. A package can be any of the following:

• A class hierarchy
• An abstract class and its concrete subclasses
• A group of collaborating classes
• A set of standard reusable components
• An application framework
• An application-specific use of such a framework
• Any unit of application-specific functionality

Class hierarchy. A hierarchy of classes can form a package, especially if the classes perform a similar or related function. For example, the class Printer and its subclasses DotMatrixPrinter, LaserPrinter, and ColorPrinter might form a reasonable Printing package.

Abstract class and its subclasses. A special case of a class hierarchy package is one consisting of an abstract superclass and its concrete subclasses. An abstract class is a class that is not intended to have concrete instances. Instead, its purpose is to define an interface and encapsulate functionality common to all subclasses—those classes that model real-world or application entities and therefore do have instances. Common functionality is encapsulated in the abstract superclass so that it can be implemented once and maintained easily. Concrete subclasses inherit the functionality and add the additional code necessary to implement their unique functionality, whatever it may be.

The Printing package above is an example of such a package, the superclass Printer being abstract and its subclasses concrete. Another example comes from a structured drawing application that implements an abstract superclass DrawingElement and its concrete subclasses Line, Text, Ellipse, Rectangle, and so on.

Collaborating classes. A group of classes might be unrelated by hierarchy, but they might collaborate closely to provide their functionality. The degree to which a set of classes collaborate can
be assessed, at least roughly, by the number of messages they can send to each other. Classes that send many messages to each other form a tightly collaborating group. Fewer messages usually mean less collaboration. For example, a PrintServer class would probably collaborate closely with classes in the Printer hierarchy.

Another example comes from an application modeling a more domestic domain: your home water heater. The WaterTank informs the Thermostat of its temperature. The Thermostat turns the HeatingElement on or off. And the HeatingElement, when turned on, causes the temperature in the WaterTank to rise. These three classes collaborate, and they could provide a sound basis for the Heating package in a plumbing application.

Reusable components. Standard reusable classes such as Array or String do not belong in an application-specific package. If, as part of developing your application, you need to create a component (or set of components) that can be useful to many applications, group them in their own package (or packages).

Frameworks. A framework for an application can be thought of as a reusable design. A framework is a set of classes, usually abstract, that provide the skeleton of an application. Their collaborations are not specific to any particular application, but rather delineate the structure that a variety of applications could take. Such collaborations provide a context for many applications. They are usually more important than the class hierarchies of the classes that make them up. A framework, therefore, makes a good basis for a package.

For example, a structured drawing application could use a Figure framework to structure the way users can create and modify elements of the drawing. A drawing is a kind of Figure, and each of its elements is a kind of FigureElement. Specific figures are created by specific Tools and have Handles by which the user can move or modify them. The classes Figure, FigureElement, Handle, and Tool form a relatively simple framework that makes a coherent package.

If a framework is complex, it might form the basis of several packages instead.

Application-specific use of a framework. Once a framework has been created, a logical package within an application comprises those classes that use it. For example, the class StructuredDrawing is a concrete subclass of Figure. Similarly, StructuredDrawingElement is a concrete subclass of FigureElement, StructuredTool is a concrete class of Tool, and
StructuredHandle is a concrete subclass of Handle. Although the drawing application includes other classes as well, these classes form the basis of a package.

Application-specific functionality. Not all package-specific functionality can be neatly partitioned by class. Sometimes a class outside of your package or application needs to be extended for your application. A method added to a class outside the package can nevertheless be part of a package. For example, the plumbing application might need to extend class Object with a method isPlumbingComponent, to determine if a given object is a component in the plumbing application. In this case, the method extending class Object is nevertheless part of the appropriate package in the plumbing application, even though Object clearly is not.

Creating a Package

The mechanical process of making a new package is simple:

1  In the Package Browser, choose Module / New -> Package...
2  In the window that pops up, type the name of the package.
3  Click OK.

The module definition editor appears in the contents pane.

4  In the Module Comment field, comment the package.

Include the purpose of the package, an overview of the classes and other definitions it will contain, and what applications or users will use it. If you don't know some of these things now, you can always update the comment later.

The Module Information field is read-only. For a new package, it is empty. It will be filled in after you have committed a new revision of the package.

5  When you are done, choose File / Save.

Reorganizing Packages

Packages can be made at various stages of the development process. You can start with one empty package (or even none) and blast off in a coding frenzy, worrying about reorganizing later. Then, after you have cooled down, you can organize the definitions you have added or changed into packages.
The “program first, reorganize later” method may have the advantages of simplicity and familiarity, but we do not recommend it. Especially if your application is large or complex (and most applications grow larger and more complex than we first expect), it can be very confusing to view all your new classes, methods, and variables sprinkled throughout the browser.

When you select a package in the leftmost pane of the Package Browser, only those items belonging to that package are visible to you. (A class is visible if even one of its methods belongs to the package, but the class name will not appear in boldface.) This can be a powerful programming aid, letting you focus your thoughts on the problem at hand. It makes your work in progress easier to understand, and it makes it easier to find what you are looking for.

We therefore recommend that you plan your work (design your application) in advance, subdividing the effort and the functionality in a reasonable, helpful, organized fashion. Then, make the packages you need. Before you start work, select the package you want to work on.

You might be wrong—few of us can design something perfectly the first time. But even if you are, you can always make new packages or reorganize as necessary. Your work will still be easier than it will be if you have to reorganize one large, undifferentiated group of classes, methods, and variables.

If such a formal design process does not suit you, another way you can avoid reorganizing a large, undifferentiated group of definitions is to divide the work of programming your application into tasks. Then, create a new package each time you start a new programming task. Place what you create as part of that task into the new package. In that way, you end up with a set of packages containing some logical structure. You may have to reorganize them somewhat, but the job will be easier than it would have been without the structure you gave yourself.

**NOTE:** For more help in structuring your application, see the section *Dividing an Application into Packages* earlier in this chapter.
Adding Code to a Package

You can add code to a package in two ways.

To add code as you define it:
1. In the Package Browser, select the package to which you want to add code.
2. Create the items you want to add to the selected package.

To add code defined in another package:
1. Open a Package Browser or a Package Organizer.
2. Drag the definition you want to add to the package name representing the package you want the definition to belong to.
   
   For example, drag a method name from the method list pane into the new package. You can drag from one browser to another, if you want.
3. Drop the definition on the desired package name.

The context button at the bottom left of a Package Browser shows the package to which the selected definition belongs. If you drop an item into a different package, the context button updates, displaying the name of the target package. If you want to view the contents of the target package, click the context button, and the Package Browser updates to display the definitions it contains.

Reordering Definitions

Two tools are available in the Visual Smalltalk Enterprise development environment to help you organize definitions. The Package Browser lets you move items from one package to another. The Package Organizer also lets you do that, as well as specify the initialization sequence of definitions within a package.

NOTE: You might want to move definitions from one package to another for a variety of reasons having to do with design and maintenance. To learn more about these issues, see the section entitled Dividing an Application into Packages earlier in this chapter.

Initially, definitions appear in the Package Organizer in the order in which they were created. The order in which definitions that include initializers appear in the Package Organizer defines the order in which they are evaluated during package initialization.
The desired initialization order might not be the order in which they were created. Therefore, the definitions might need to be reordered.

To reorder the sequence of definitions within a package:

1. Open a Package Organizer by selecting Smalltalk / Open / Package Organizer.

2. In the top left pane, select the package whose definitions you want to reorder.

   The definitions it contains appear in the top right pane.

3. Select a definition and then pull down the Definition menu and choose Move -> Top to move it to first in the sequence, Move -> Bottom to move it to last place, Move -> Up to move it above of the item it is currently beneath, or Move -> Down to move it below the item it is above.

TIP: When you are doing extensive reorganizing, try zooming in on the pane of interest. For example, select a module in a Package Browser and choose the Options / Zoom item to expand the module list pane so it fills the entire window. Resize it so it is tall and narrow, and you can focus on dragging definitions from other windows to this package browser.

Using Method Categories

A method category is a convenience provided in the Package Browser. It is a way of grouping related methods together within a class so that they are easier to find. A category does not affect execution in any way. It is merely a name or phrase used to describe the purpose of a group of methods. For example, all of the methods that a given class provides for copying objects might be grouped into the category “copying.” All the methods that define the internal implementation of a class, but which are not intended to be part of its external interface, might be grouped into the category “private.”

NOTE: Consult the Programming Guidelines appendix in your Visual Smalltalk Enterprise Language Reference for suggestions about category names.
This mechanism can also enhance communication among developers when used consistently. For example, you can tell the developers on your team that other classes should not send the messages implemented in “private,” but that they are reserved only for the class to send to itself.

**To create a category:**

1. In a Package Browser, select the class for the new category.
2. Choose Method / New Category... to open a dialog prompting for the category name.
3. Type the name of your category and press OK.

Your category appears in the category pane (the second from the right), just to the left of the method pane (the rightmost pane).

**TIP:** To create a hierarchical category, use a hyphen (-) to separate the categories. For example, *accessing-formats* is the name of the subcategory named *formats* within the category *accessing*.

**To move existing methods into the category:**

1. Select the method you want to move.
2. Drag the method over to the category pane until the desired category is highlighted.
3. Release the mouse button to drop the method into the category.

The method is removed from the old category.

**To create new methods in the category:**

1. Select the category.
2. Edit the method template in the contents pane to create the method.

Any method created when the category is selected becomes a part of that category. When you view inherited methods, inherited categories are also visible.

**NOTE:** You can also create a new method by changing the selector of an existing method in the category and editing the method body as necessary.
Restructuring a Class Hierarchy with Packages

The purpose of inheritance is to factor out code common to more than one class, so that it resides in one place where it is easier to maintain. The purposes of a package are different: a package is a single module of functionality, manageable for one programmer to develop, and logical to store, maintain, and distribute as a unit.

A package can contain a complete class hierarchy, but it need not do so, and often it should not. Class hierarchies will often be contained within several packages. An abstract superclass might be defined in one package (such as part of a framework), while one application-specific subclass is defined in one application-specific package, and another such subclass is defined in another package.

For example, a plumbing application might include the abstract superclass `PlumbingComponent`, with subclasses for `Valve`, `Spigot`, and `Pipe`. `Valve` itself has two subclasses (`CheckValve` and `ControlValve`), and `Pipe` has four (`StraightPipe`, `TPipe`, `NibPipe`, and `ElbowPipe`).

However, pipes have different characteristics depending upon what they are made of. Suppose the plumbing application needs to query a pipe about its flow rate, strength, malleability, or suitability for a specific purpose. In that case, a new class, `PipeCharacteristics`, can be made to encapsulate this knowledge. Subclasses such as `CopperPipeCharacteristics` and `ABSPipeCharacteristics` can contain the specific details of each type of material.

If a great deal of code is associated with these components, it could make sense to create four packages for them:

- One package holds the abstract superclass `PlumbingComponent`.
- One package holds `Spigot`.
- One package holds `Pipe` and its subclasses, and `PipeCharacteristics` and its subclasses.
- One package holds `Valve` and its subclasses.

In this way, the plumbing application can be developed, distributed, and maintained in a modular fashion.

Initializing Classes

You can initialize a class at any time as you develop, and you may need to do so. But before you can distribute a usable package, you must define the order in which the class initializers (as well as
pool variable, global variable and ad-hoc initializers) within it will initialize. These two operations are distinct, but they both use a concept that might be unfamiliar to you: class initializers.

A class initializer is an expression (one or more Smalltalk statements) that is executed when a class is initialized. It is not the same as an initialize method. A class initializer could, of course, consist of the simple statement:

```
self initialize
```

In such a case, the class initializer does nothing more than call the initialize method.

However, you may not want to place your initialization code in an initialize method. Code in the initialize method is inherited by all subclasses. Code in the initializer is not. If the code applies only to the class but not its subclasses, put it in an initializer, or put it in a private method that is invoked by the initializer. If the code applies to all subclasses, put it in the initialize method.

If you use a class initializer to initialize a class, you might need to define corresponding class initializers for each of its subclasses.

To create a class initializer:

1. In the global pane of a Package Browser (the second pane from the left), select the class that needs the class initializer.
2. Choose Global / Initialization.
3. In the contents pane, type the initialization expression.
4. Choose File / Save to record the initializer.

Figure 5-2: Making a Class Initializer
As you interactively develop a class, you might need to initialize it before you use it. You can reinitialize it whenever you are unsure that the class is in a valid state.

**To initialize a class while you are developing:**

1. In the Package Browser, select the class you want to initialize in the Globals pane.
2. Choose *Global / Initialize*.

The class executes the code you have defined as the class initializer. Class initializers are also executed when the package containing it is initialized.

A package is initialized when it is loaded when the *Initialize after Load* checkbox is checked on the *Load* dialog.

**NOTE:** To learn about initializing other aspects of your application besides classes, see the section below titled *Using ad hoc initializers*.

---

**Using ad hoc Initializers**

Ad hoc initializers are mechanisms to set up various aspects of the initial state of your application or the system that runs it. An ad hoc initializer is an expression (one or more Smalltalk statements) that, like any other definition, executes when the package containing it is loaded. Many Smalltalk programmers are accustomed to executing expressions with *Do It* to initialize an application. Ad hoc initializers provide a formal mechanism for many, but not all of the same functions for which *Do It* has been used:

- You can provide system resources to your application and connect them.
- You can provide visual feedback to the user while the application is initializing.
- You can set up external hardware devices in a specific sequence. For example:
  ```smalltalk
  Safeguard1 release. 
  Safeguard2 release. 
  Safeguard3 release 
  ```
- You can open the main application window. For example:
  ```smalltalk
  AppWindow open 
  ```
In contrast to traditional Smalltalk development practices, ad hoc initializers and *Do Its* should not be used to create or initialize global or pool variables. If you create global variables or pools through the result of a *Do It* or ad hoc initializer, Visual Smalltalk Enterprise is not able to accurately check for conflicts or composition errors. Global variables or pools created with *ad hoc* initializers are placed in the **unpackaged** package.

If you try to load a package that references a global defined by a *Do It* or initializer the load will fail because the global cannot be resolved as a forward reference. For more information on loading packages, see *Sharing Code Without Access to a Common Repository* on page 227. You should always use global and pool variable definitions to declare these globals.

**NOTE:**Globals created by *ad hoc* initializers may cause composition errors. All globals created in this way are put into the **unpackaged** package.

To create an *ad hoc* initializer:

3 In the Package Organizer select the package you want to add the initializer to.

4 Choose *Definition / New* -> *Initializer*.

In the definition list, an empty initialization expression appears, already selected.

**NOTE:** Unlike method creation, you cannot create a new initializer by editing an existing one. You must use *Definition / New* -> *Initializer* to create an initializer.

5 In the contents pane, enter a comment on the first line and the expression you want to execute when the package initializes on the second line.

6 Choose *File / Save* to record the initialization expression.

The first line of the text in the text pane appears in the definition list (top right pane). If you want this line to be informative, type a comment at the top of the text pane to explain what the ad hoc initializer does. The first line of the comment appears in the definition list, as shown in figure 5-3.
Extending the Base Class Library

Classes in the base library are in the Visual Smalltalk Enterprise cluster. Although two packages loaded in the same image cannot both define an item by the same name, a package can redefine any class, method, or variable in the Visual Smalltalk Enterprise cluster, if you have specified that no conflict checking is set for this cluster (refer to Conflict Preference Editor in chapter 2). A definition in the loaded package can then overwrite the corresponding definition in the Visual Smalltalk Enterprise cluster.

Whenever possible, you should extend the base class library using inheritance or extensions rather than modifying base class methods. This helps insulate your code from future changes to the base class library and makes maintenance easier. For example, you can define a class in your package as a subclass of a class in the base class library. You can also extend a class in the base class library with a new method. For example, your package can add a method that belongs to a class in the base class library.

System Packages

The packages contained in the Visual Smalltalk Enterprise cluster and the package **unpackaged** are special system packages. You cannot make new revisions of **unpackaged**, and may not be able to unload them, or migrate any of these packages.
Problems with Modified Base Classes

When a definition in Visual Smalltalk Enterprise package is modified, it still belongs to the original package unless you move it to another package.

A modified definition in the base class library raises the following problems:

- When overriding definitions in Visual Smalltalk Enterprise packages, the focus of your work is on differences from the original versions. Because most definitions in the base are unmodified, it’s hard to see which class and method were modified unless you use the filter to only show only modified items.

- You can’t migrate back to the original revision of the Visual Smalltalk Enterprise cluster. If you need to restore the original state of the base classes other than those in the Team/V cluster, you must use a History Browser to undo each modification, one at a time.

- You must go to extra work to distribute these changes with your application.

If you move a modified definition of the base class library to your application package so that you can distribute it, an even bigger problem arises:

- If you unload an application package that contains modified base class definitions, you remove those definitions from the image. This can cause the image to fail if the items removed were critical to the functioning of the image.

Modifying Base Classes Safely

If you cannot use inheritance or an extension to extend the base class library, you can modify a base class definition, but you must do so in a disciplined manner. Use the following procedure when modifying a Visual Smalltalk Enterprise package other than a package in the Team/V cluster.

1. Save the original base definitions in a separate package, or commit the package that contains them.

   To ensure the integrity of the base class library, make sure the original definitions are committed. It may be convenient to want to change to a new package.

   To save the original definitions in a new package:

   a. Create a new package to contain the base class modifications.
b. Drag the definitions you intend to change from the base class library package to the new package.

c. Commit the package to store the original revisions of the definitions you are going to change.

2 Modify the base class library definitions.

You can modify the definitions in the new package as necessary and commit the changes.

3 Restore the base class library definitions.

When you want to restore the base class definitions, migrate back to the revision of the package containing the original base class library definitions.

4 Unload the packaging containing base class library definitions.

When you want to unload your application and restore the original base, migrate to the “original definition” package.

Using Clusters

Packages are the modules that store your Smalltalk code, and can represent reusable and maintainable units of functionality. Clusters represent bigger units of functionality. The ability to compose smaller units into larger units is a good way to deal with complexity.

Developers who use another developer’s clusters are protected and isolated from details about the cluster. They do not need to know which revisions of the packages are in the cluster, and which revisions work with other revisions of packages. For example, a cluster can capture the information that revision 1.6 of the application framework is required for the cluster to function properly. The developer who uses the cluster does not need to know that revision 1.5 had a bug in it that will break his application. The developer that maintains the cluster probably does need to know that.

Often, developers will end up with several clusters that overlap because they have common functionality or depend on common packages.

In figure , there are two clusters, one that represents a PERT chart editor, and one that represents a PERT chart report generator.
The specifications in the PERT editor cluster might look like this:

**PERT Editor contents:**
- **Application Framework** (cluster)
  - revision 3.3 in repository **Common User Interface**
- **Pert Chart** (package)
  - revision 5.9 in repository **PERT**
- **Pert Chart Editor** (package)
  - revision 1.3 in repository **PERT**
- **Drawing Subpane** (package)
  - revision 4.1 in repository **Graphics**

Both of these clusters contain the package **Pert Chart**, indicating a dependency on the package **Pert Chart**. The package **Pert Chart** is required for the pert chart editor. They also contain a common cluster, **Application Framework**. Developers can generate a program that contains both of these clusters.
Using Conditional Inclusion Flags

Another common situation is that some module needs to be included only under certain conditions. A condition is often something external to the application itself, such as the operating system or the presence of a certain display type.

For example, if you are building your PERT chart application to run on OS/2, you might need different chart storage code than if it ran on another platform.

Modules can be conditionally included in a cluster by setting a conditional inclusion flag. For example, in the PERT Storage example we use conditional inclusion flags in the PERT Storage cluster:

**PERT Storage contents**

- **Pert Chart** (package)
  - revision 5.9 in repository PERT

- **Pert Chart PVCS Storage** (package)
  - revision 2.3.0.2 in repository PERT
  - include if OS/2
  - include if Windows

- **Pert Chart SCCS Storage** (package)
  - revision 1.1 in repository PERT
  - include if OS/2

- **Pert Chart RCS Storage** (package)
  - revision 1.6 in repository PERT
  - include if Windows 95

The conditions that determine whether a module is included or not are set in the development environment, using the Inclusion Context Editor (**Smalltalk / Open / Inclusion Context Editor**). So, if the Pert program is being built to run under OS/2, you add the OS/2 inclusion flag to the conditional inclusion settings.

With the inclusion context set, the **Pert Storage** cluster will have two entities when the program is generated: the **Pert Chart** package and the **Pert Chart PVCS Storage** package. The other two package specifications, for OS/2 and Windows 95, are ignored because their conditional inclusion flags did not match the program’s flags.

A specification can have multiple inclusion context flags. The module is loaded only if *all* of its flags are set (satisfaction is determined using a logical AND).
You can get the effect of a logical OR by using several specifications in such a way that if any of the specifications are resolved the module will be loaded. You can change the evaluation to a logical OR by evaluating this expression:

```
TeamVInterface or ConditionalInclusionFlags: true
```

Set the flag to `false` to return the behavior to AND.

### Using Optional Inclusion Flags

Optional inclusion can be used to create configurations that cross functional boundaries. Suppose we wanted to build a configuration that represented all verified PERT chart related packages. We would do this by building a cluster with all the PERT chart related packages, and making each specification optional. With the optional flag, the cluster does not cause any of the entities to be included in a subsystem. Another cluster must require a package to cause it to be included. In our example, this cluster is called **Verified PERT Components**. Its contents would look like this:

**Verified PERT Components** contents

- **Pert Chart** (package)
  - v 5.9 in repository **PERT**
  - Optional

- **Pert Chart Editor** (package)
  - v 1.3 in repository **PERT**
  - Optional

- **Pert Chart PVCS Storage** (package)
  - revision 2.3.0.2 in repository **PERT**
  - Optional

- **Pert Chart SCCS Storage** (package)
  - revision 1.1 in repository **PERT**
  - Optional

- **Pert Chart RCS Storage** (package)
  - revision 1.6 in repository **PERT**
  - Optional

- **Pert Chart Report Viewer** (package)
  - revision 1.6 in repository **PERT**
  - Optional
With this cluster to specify revisions, we can now create other clusters that specify structure. For example, the **PERT Storage Cluster** could be modified to provide only a structural description.

**PERT Storage contents**

**Pert Chart** (package)

**Pert Chart PVCS Storage** (package)
- include if OS/2
- include if Windows

**Pert Chart SCCS Storage** (package)
- include if OS/2

**Pert Chart RCS Storage** (package)
- include if Windows 95

### Creating a Cluster

You can create a new cluster using either the Package Browser or the Cluster Organizer.

**To create a new cluster:**

1. Open a Cluster Organizer or Package Browser.
2. Select **Cluster / New... / Cluster** (Cluster Organizer) or **Module / New... / Cluster** (Package Browser).
3. Fill in the name of the new cluster in the dialog that appears.
   - The name can contain spaces and punctuation and can be any length.
4. Press the **OK** button to create the cluster.
   - This creates a new, open cluster, which does not show up in the Package Browser.

### Adding a Specification to a Cluster

You can use drag and drop to add a specification to a cluster or manually create a new specification in a cluster.

**To add a specification to a cluster using drag and drop:**

1. Find the package or cluster whose specification you want to add using either the Package Browser, Package Organizer, or the Cluster Organizer.
2. Drag the desired module from the module list in the Package Browser and drop it on the target cluster, either in the same Package Browser, a different Package Browser, or the Cluster Organizer.
The result of the drag/drop operation varies a little if the target cluster is a group or a configuration.

- If you drag a module to a configuration, it is added as a complete specification. If you drag it from a group, the specification is completed using information from the currently loaded revision satisfying the specification.

- If you drag a module to a group, it is added as a minimized specification. If dragging the last complete specification, you must do a drag copy operation (hold the Ctrl key while dragging), so a new minimal specification is created.

You can manually add a specification to a cluster as follows. Use this method if the entity is not visible in any browsers in your image.

**To manually add a specification to a cluster:**

1. In the Cluster Organizer, select the cluster you want to add the specification to.
2. Choose Specification -> New Specification....
3. Fill in the information in the Specification Editor that appears.
   - See the section entitled Editing a Specification in chapter 4 for more information on this editor.
4. Press the OK button to create the specification.

### Making a Specification Optional

If a specification has the optional inclusion flag set, loading the cluster that contains it does not cause the module to be loaded.

**To make a specification be optional:**

1. Select the specification you want to make optional in the Cluster Organizer.
2. Choose Specification / Optional to turn on the optional inclusion flag.
   - The word “optional” appears before the specification and a check mark appears next to the Optional menu item.

To make an optional specification not be optional, repeat the steps above to toggle the optional flag off.

You can also set the optional flag from the Specification Editor.
Adding a Conditional Inclusion Flag to a Module

To add a conditional inclusion flag to a module:

1. Select the specification you want to make conditional in the Cluster Organizer.
2. In the lower right-hand pane, the conditional flag list, press Add...
3. Enter the name of the conditional inclusion flag you want to add in the Create flag named: dialog that appears and press OK.
4. The new conditional inclusion flag appears in the lower right-hand pane of the Cluster Organizer.

Setting Conditional Inclusion Flags in the Environment

Whenever a cluster is loaded that contains specifications with conditional inclusion flags, the conditional inclusion flags for the specification are checked against the conditional inclusion flags set in the environment. If the flags match, the specification is included in the loaded cluster.

_TIP:_ For a good convention to follow for choosing platform-specific flag names, evaluate the Smalltalk expression Smalltalk product and use the result as a conditional inclusion flag.

To set the conditional inclusion flags in the environment:

1. From a Smalltalk menu, choose Browse -> Inclusion Context...
2. The Conditional Inclusion Context dialog opens. The conditional inclusion context for the environment appears in the list box.
3. To add a conditional inclusion flag to the environment, press Add and enter the flag name in the dialog that appears.
4. To delete a conditional inclusion flag from the environment, select the flag in the list box and press Delete.
5. When you are finished making changes to the conditional inclusion flags, press OK.
6. The conditional inclusion flags you have changed are checked by the environment for any conflicts. If conflicts are detected, you are notified and given another opportunity to modify the flags.
Interpreting Module Specifications

The module specification that appears in specification list of the Cluster Organizer contains several pieces of information.

![Figure 5-5: Specifications in a Cluster Organizer](image)

As shown in Figure 5-5, module specifications are displayed as a number of fields in the form:

```
{ optional } name { [ inclusion flags ] } : type rev revision in repository
```

In this specification:

- The word "optional" indicates that the optional inclusion flag is turned on. If the flag is off, "optional" is not displayed.
- The `name` field contains the module name. The name may be followed by square brackets ([ ]), containing conditional inclusion flags defined for this specification.
- The `type` field gives the type of specification, either a cluster or a package.
- The `revision` field contains the module revision.
- The `repository` field contains the repository name.

The `type`, `revision`, and `repository` fields may be left unspecified.

If a specification is resolved and some fields are unspecified, the actual values of unspecified fields are displayed following the specified fields and enclosed in parentheses. If the module is
marked as fully specified, but a field value is not known (for example, the specification is uncommitted), the unknown field value is left blank.

If a specification is either optional or conditionally loaded and not included, its text is printed in strikeout.

Specifications that have been modified since their cluster was last committed are highlighted using the “modified” color and italicized.

**Expanding and Contracting Cluster Specifications**

If the module specification is preceded by a plus or minus sign, the resolved module is a cluster that contains other modules. Click on the cluster specification icon that displays a plus sign to expand the specification to show the first level of contained modules. Double-click on an icon with a plus sign to fully expand the specification. Click on an icon with a minus sign to conceal the child module specifications.

**Converting Between Clusters, Groups, and Configurations**

Several tools are provided for converting cluster specifications, simplifying the process of creating groups and configurations from clusters.

Groups and configurations, as has been explained earlier, are simply a special kind of cluster. When you make a group or a cluster, the `.ClusterUsage` annotation is added to the cluster definition. This field is displayed in the annotation pane in the lower left of a cluster organizer. Select this annotation name to see whether the cluster is a group or a configuration, as shown in figure 5-6. If a cluster is not identified specifically as a group or a configuration, this field is not shown.
Converting a cluster to a group requires minimizing all of the specifications in the cluster. Similarly, converting a cluster to configuration requires completing, or maximizing, all of its specifications. The conversion tools attempt to minimize or complete the specifications for you.

Converting either a group or a configuration to a cluster involves no particular change to the cluster or its specifications beyond removing the cluster usage annotation.

The conversion tools are available in the Package Browser (Module / Convert To) and the Cluster Organizer (Cluster / Convert To). The resulting menu allows you to select either Group, Configuration, or Cluster.

Converting a Cluster to a Group

When you choose to convert the cluster to a group, the tool attempts to minimize each specification in the cluster. If it succeeds, then the conversion is finished. Commit the cluster to save it as a group.

If one or more specifications cannot be minimized, the tools display a dialog indicating that it could not. The cluster itself is made into a Group, but it still contains more fully defined specifications than it should. Before committing the cluster, make sure you minimize its specifications as described below.
A specification cannot be minimized if doing so would remove information needed to specify an associated loaded module. To create a minimal specification in this case, you must first create another complete specification for the module.

**Converting a Cluster to a Configuration**

When converting to a configuration, the tool completes all specifications using the revision level and other identification from the currently loaded module matching the prior specification.

Before you convert the cluster, *make sure* you have the correct revisions of the specified packages loaded. The tool may not be able to complete specifications in open clusters.

After converting the cluster to a configuration, commit the module.

**Minimizing a Complete Specification**

You may also manually minimize a specification or group of specifications.

**To minimize a single specification:**

1. In a Cluster Organizer, select the cluster and specification you want to minimize.
2. Select *Specification / Minimize*.
   
   If necessary, a confirmation dialog is displayed asking whether to create a new top level specification.
3. Click on *Yes* to create the new specification.
   
   The specification is minimized, and parentheses are placed around the revision and repository information.

**To minimize all specifications in a cluster:**

1. In a Cluster Organizer, select the cluster containing the specifications you want to minimize.
2. Select *Specification / Minimize All*.
   
   If necessary, a confirmation dialog is displayed asking whether to create new top level specifications.
3. Click on *Yes* to create the new specifications.
   
   The specifications are minimized and parentheses are placed around the revision and repository information.
The new specifications, which includes all the information that had been contained in the original specifications, are detached and displayed in the Package Browser module list as top level modules. These are the currently “loaded” modules, and the specifications in the group refer to them.

**Completing Partial Specifications**

A partial specification displayed in the Cluster Organizer that has been resolved to a module displays a complete specification for the module, with the fields filled in from the resolved module in parenthesis. You can use the information from the resolved module to complete (fill in the rest of the fields in) the partial specification.

To make a partial specification that has been resolved to a module complete:

1. Select the cluster and specification in the Cluster Organizer that you want to make complete.
2. Choose Specification / Complete to make the partial specification be complete. The parenthesis are removed.

**Using Revisions**

This section contains the following topics:

- Keeping track of your work.
- Modifying a module.
- Making a new revision of a module.
- Viewing the revisions of a module.
- Examining an older revision.
- Comparing versions of source-based definitions
- Reverting to an older version of a definition.
- Comparing revisions of packages.
- Reverting to another revision of a module.

**Keeping Track of Your Work**

Visual Smalltalk Enterprise provides a variety of mechanisms to help you keep track of your work. Repositories provide a shared team history of development work by keeping track of committed revisions of your packages and clusters. Meanwhile, the development tools show what an individual developer has done within an image by keeping track of changes made in between committing new revisions.
Each revision of a module has a revision comment. You can review these comments to determine what has changed with each revision.

The Visual Smalltalk Enterprise development tools keep track of the current revision of the module loaded or opened in your image, and whether it has been modified since it was loaded or opened. When you change any definition or specification in any way (including changes to a comment), and save your change, Visual Smalltalk Enterprise considers the module to have been modified. If you add or delete a definition or specification, Visual Smalltalk Enterprise also considers the module to have been modified. If a module has been modified since it was last committed, the module name appears in the tools in italics (by default, blue italics on a color monitor).

Modified or new definitions and specifications also appear in the tools in italics, so that skimming names for italics can give you a rapid idea of precisely what changes have occurred since the package was last committed. Once you commit the package again, the italics are replaced by a roman font.

You can also use the filter dialog in the Package and Definition menus in the Package Organizer and the Module, Global, and Method menus of the Package Browser to display only modified modules and definitions.

**Version History**

As you browse a package, the *version* button at the bottom right of the Package Browser and at the bottom of a Message Browser shows the version label of the selected definition. If you modify the selected definition, the version label is replaced by a time stamp showing the time and date of the change and the name of the user who made it.

If the user name is displayed as `editThisName`, the user identification configuration was not completed properly at the time the definition was saved. See *Setting Your User Name* in chapter 2 for more information.

If the user name is *unknown*, PVCS was not available at the time the definition was saved. A required module may not have been available or in the user's path. See *PVCS Administration* in chapter 2 for more help.
If you click on the version button at the lower right of a Package Browser or at the bottom of a Message Browser, a History Browser appears. If no previous versions are available, you are notified with a dialog. If the definition has versions in the change log, the History Browser shows the original version of a definition and a history of all changes made to it that are known to the image.

When you commit a revision to a repository, this version history remains. As long as you save your image and change log and do not compress changes, the history is retained. The following table shows which definitions are versioned.

<table>
<thead>
<tr>
<th>Definition type</th>
<th>has versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>no</td>
</tr>
<tr>
<td>Method</td>
<td>yes</td>
</tr>
<tr>
<td>Class variable</td>
<td>no</td>
</tr>
<tr>
<td>Instance variable</td>
<td>no</td>
</tr>
<tr>
<td>Class instance variable</td>
<td>no</td>
</tr>
<tr>
<td>Pool usage</td>
<td>no</td>
</tr>
<tr>
<td>Class initializer</td>
<td>yes</td>
</tr>
<tr>
<td>Global variable</td>
<td>yes</td>
</tr>
<tr>
<td>Pool</td>
<td>no</td>
</tr>
<tr>
<td>Pool variable</td>
<td>yes</td>
</tr>
<tr>
<td>Ad hoc initializer</td>
<td>yes</td>
</tr>
</tbody>
</table>
Committing a package does not discard the version history for a definition. Visual Smalltalk Enterprise does not support discarding version history on a per package basis.

If you load the required revision of a package into a fresh image, it has no version history unless it overrides definitions from the base class library.

Compressing the change log obliterates the entire version history of all development work in your image. You compress the change log with the expression

```
SourceManager current compressChanges
```

If you have not compressed changes, you can click on the version button to open the History Browser to examine previous versions of a definition and to revert to any previous version. Click on a version of a definition in the version list pane of the History Browser, and the text associated with that version appears in the contents pane. You can copy and paste text between versions as necessary. After you have examined previous versions, if you decide to revert to a previous version:

1. Select the version you want to revert to.
2. Choose Version / Install.

A copy of the selected version becomes the current, active version of the definition, but the original time stamp information is retained. Subsequent versions (that is, those that used to be subsequent) are unaffected.

Figure 5-8 shows the History Browser after making a change to a loaded method.

**Figure 5-8: History Browser Showing Two Versions of a Method**
Figure 5-9 shows the History Browser after the original method has been reinstalled with Version / Install.

Figure 5-9: Reverting to a Previous Version of a Method

Modifying a Module

Visual Smalltalk Enterprise considers a module modified if:

• You have added a definition or specification to it.
• You have deleted a definition or specification it contained.
• You have modified a definition or specification it contains in any way, and you have saved the change. Such modifications include adding, deleting, or modifying a comment or annotation.
• You have added, deleted, or modified the module comment or any annotation.
• You have reordered the definitions in a package.
• You have renamed the module.

When a module is modified, its name appears in italics in the browsers. To save these changes outside the image (on disk), you must commit a new revision of the module to a repository. After the module has been committed, it is marked as unmodified, and the italics are replaced with a roman font.

Making a New Revision of a Module

Committing a module records the current definitions or specification in an external location where it is accessible to others who can access the repository. You should commit a fresh revision of a module whenever such an operation would be useful or convenient.
• When you have finished working on the module and believe it to be fully tested and ready for use by others.
• When others need access to the module.
• Before starting an experiment that you suspect may not work out.
• Prior to building Smalltalk libraries based on modules.
• When your disk is too full to save your image. Modules are typically much smaller than whole images, and committing one can let you save your work immediately while you consider the best way to deal with your disk space problem.

In any event, be sure to include a comment when you commit the revision so you know what the state of the module was at that time. Your team may develop guidelines about when you should or should not commit to a common repository that others can access. Your team policy may state that only fully tested and commented modules can be committed to the common repository.

To **commit a new revision of a module:**

1. In the Package Browser, select the module you want to commit.
2. Choose *Module / Commit*....
   
   The Commit dialog opens.
3. Select the repository to which you want to commit the new revision.
   
   The default is the repository from which the current one was derived.
4. Check the revision number (it is automatically incremented for you) and fill in the comment for the module.
5. Press *OK* to commit the module to the repository.
Identifying the Revision of a Module

In the Package Browser, the revision of the currently loaded module (or base revision if the module has been modified) is displayed along with other information about the module in the module pane. You may need to do a Module / Update to refresh the module pane.

The revision information is also visible in the Module Information pane when the module is selected.

NOTE: Make sure that no class or global is selected in order to view the Module Information pane.

In the Package or Cluster Organizer, the revision of the selected module is available by selecting the .RevisionNumber annotation in the annotation pane.

NOTE: If there are no revisions of the module yet, no revision number will appear.

Viewing the Revisions of a Module

You can view the revisions and revision comments of any module in the Repository Browser.

To view the list of revisions of a module:

1. In any tool, select Smalltalk / Open / Repository Browser.

The Repository Browser opens, as shown in figure 5-10.
Examining an Older Revision

You can examine any revision of a package without loading it and making it part of your executable image. When you open a package to examine it, you can perform a subset of development tasks:

- You can add, delete, and rename definitions.
- You can view the source code in that revision of the package.
- You can edit the source code.
- You can reorganize the sequence of definitions.
- You can commit changes and make a new revision.

However, when a package is open but not loaded, you cannot perform these tasks:
- You cannot execute source code in the package.
- You cannot build an SLL based on the package.
- You cannot check variable references for validity.
You cannot check for conflicts.

You cannot perform any deeper debugging operations that depend upon execution, or depend upon the semantics of the source code.

To open a package for examination:

1. In the Package Organizer, choose Package / Open....

   A module Open dialog appears, letting you select any revision of any package from any repository known by the system. The current revision of the selected package is visible.

   **NOTE:** If you want the system to include new repositories in this dialog, you can add them by pressing the Repository... button and connecting to a new repository with the Repository Browser.

2. Select the revision you want to open.

   You can also open a file-in for examination using the File... button in the module Open dialog. If you specify a file-in instead of a package, Visual Smalltalk Enterprise converts the contents of the file-in to definitions in a package.

   If you cannot load a package or file-in because a conflict has occurred, open the package and edit it to eliminate the conflict. You can then load the modified package.

**Comparing Versions of Source-Based Definitions**

Visual Smalltalk Enterprise lets you directly compare two versions of any method.

To compare methods:

1. In a Package Browser, select the method whose versions you want to compare.

2. Choose Method / Compare Versions.

   If a version history is available for the selected method, a Version Comparison browser opens on the selected method. The two top panes contain lists of the method's versions.
Select one of the versions in one of the panes, and the associated text appears in the text pane below it.

Select one of the versions in the other pane, and the associated text appears in the text pane below.

Differences between the two versions are underlined (in red on color displays).

**Reverting to an Older Version of a Definition**

To revert to an older version of a definition:

1. Select the version to which you want to revert.
2. Pull down the **Left** or **Right** menu, depending on where the version is that you want to restore.
3. Choose **Install**.

The selected version becomes the currently active version in the image.

**Comparing Revisions of Packages**

Visual Smalltalk Enterprise lets you directly compare two revisions of any two packages.

To directly compare any two revisions of any two packages:

1. In a Package Browser or Package Organizer, select one of the packages whose revisions you want to compare.
2 Choose **Module / Compare** in a Package Browser or **Package / Compare** in a Package Organizer.

A comparison browser opens with the package selected in the left pane.

If the revision you want to compare is not in the list in the right pane, choose **Right / Open...** to open it.

3 Select one of the revisions in each of the top panes, and the associated lists of definitions appear in the middle panes below them.

Differences are highlighted in one of two ways. Definitions that appear in one revision but not the other appear in bold green (by default) text; definitions that appear in both revisions but are different appear in italics. By default, these italics are red on a color display. This default can be changed by setting your preferences.

![Comparing Two Revisions of a Package](image)

4 Select one of the definitions in each of the definition lists (the middle panes), and the associated text appears in the contents panes at the bottom. Double-click on a definition to automatically select the corresponding definition from the other side (if one exists).

Differences between the text of two definitions are underlined. By default, this underlining is red on a color display. These defaults can be changed by setting your preferences.
NOTE: For information on specifying your color preferences, see the section entitled Color Preference Editor on page 65.

See the Browsers and Tools chapter for more information about using the Package Comparison browser.

Reverting to Another Revision of a Module

You can change the current revision of a module loaded in your development environment at any time or migrate to an entirely different package.

To change a loaded module from one revision to another:

1. In a Package Browser, select the module whose revision you want to change.
2. Choose Module / Migrate....
   
   A Migrate dialog appears.

   If you want to discard any uncommitted modifications you have made to a module, you can choose the same revision you have been working on.

3. Select one of the revisions in the list.
4. If you want the module to be initialized after the migrate operation, toggle the check box in the lower left.

   It does not reinitialize your module unless you explicitly request that the initialization be performed.

5. Press the Migrate button.

   Visual Smalltalk Enterprise adds, deletes, and modifies definitions to bring your image to a state consistent with the specified revision of the module.

To unload one module and load an entirely different module in one operation:

> Follow the steps above, choosing a different module in the Migrate dialog.

Migrating to a previous revision of a package does not change the version history in your image. If you've been working on a later version and have saved your image, those versions of definitions still appear in the History Browser, earlier in the list, below the newly loaded, older versions.
Working with Repositories

This section covers the following topics:

- Determining what kind of repository to use.
- Creating a repository
- Backing up a repository
- Access control
- Locking
- Optimistic versioning
- Pessimistic versioning

Determining What Kind of Repository to Use

Visual Smalltalk Enterprise makes two kinds of repositories available: PVCS or File-Based.

You are not limited to one repository, nor to one kind of repository. For example, a developer can use a local, private repository to commit revisions frequently, saving small increments of work to provide flexible options for updating or reverting. When the package is ready to be released to the rest of the team, a version can be committed to a public repository to which all team members have access.

Or, a project can keep a master repository of released revisions of an application, while another repository stores working revisions representing current development efforts.

A repository is stored in platform-independent format. You can have a common repository on a networking file system that teammates on the same project access from OS/2, Windows, Windows 95, and/or Windows NT. You can also move its physical location between any supported system if necessary.

PVCS saves revisions in an efficient format that uses less disk space. The File-Based repository uses more disk space to store the revisions than PVCS, since each revision is stored in its entirety. Performance considerations for each depend on your network configuration.

Both kinds of repositories automatically label new revisions with the correct package name and revision number. See the Revisions in chapter 1 for more information on revision numbering.

At the time you create a repository, you specify its name, type and whether to use pessimistic or optimistic locking. Once created, you cannot change any of these attributes.
Creating a Repository

To create a new repository:

1. In a Visual Smalltalk Enterprise window, select Smalltalk / Open / Repository Browser.

2. Select Repository / New...

   The Create Repository dialog opens. You must provide entries for all of the fields, as described in the following steps.

3. In the Name field, enter a name for the new repository.

   The name can contain spaces and punctuation and can be any length.

4. Specify a directory path for the repository in the Path field.

   You can either enter a pathname directly or select a from a directory selection dialog.
Working with Repositories

Figure 5-14: Using the Create Repository to Specify the Repository Directory

To select a directory, click the Set Path... button, then select a drive and directory path in the usual way.

To enter the path name directly, type the full directory path for the directory. If the directory does not exist, it will be created. An error occurs if drives or directories are missing from the path, or you don’t have permission to create files at that location.

5 Choose the type of the new repository from the Type list.

The repository types are explained in Revision Control in chapter 1. The normal choices are PVCS or File-Based.

6 Choose a locking policy for the new repository from the Locking list.

Your options are optimistic locking and pessimistic locking.

7 Press the Create button to create the repository.

Backing up a Repository

All the information in a repository is contained in files and subdirectories in the file system directory that is the path of the repository. You can back up a repository by backing up all files and subdirectories in the repository directory.

Controlling Repository Access

Access control lets someone—perhaps the project manager or the team integrator—specify who can commit a new revision of a given package.

Many operating systems provide the capability to assign file and directory access rights. You can use these to provide limited access control to archives in a repository. Many networking
systems provide additional access control, for instance by enabling and disabling file sharing. You can use all of these capabilities with both PVCS and File System repositories.

Access control is supported by Visual Smalltalk Enterprise, beginning with release 3.1. Using this system, you can assign access rights to individuals and groups for each repository and repository entity.

By default, all users have access to all repositories and entities. To restrict access to a repository or entity, you must explicitly assign access permissions to individuals and/or groups for that repository.

The access database must be embedded in the PVCS files, using the procedure described in Configuring the Access Database in chapter 2.

Assigning Repository Access Rights

To view access permissions for a repository, select the repository in the Repository Browser, then select Repository / Edit Access. The list of users and groups with access permission is displayed in the following dialog:

*Figure 5-15: Repository Access Rights Editor*

Click on the name of a user or group to see their current permissions. Permissions are indicated in the check boxes to the right. The permissions are:

- **Read** - allows the user to view the contents of the repository. Read access is required for all other access permissions.
- **Create** - allows the user to create new repository entities, and to append information to the repository.
• *Destroy* - allows the user to destroy repository entities and the repository itself.

• *Administration* - allows the user to modify the set of users allowed to access the repository and their access permissions. You are not restricted to a single administrator.

To modify the access rights to a repository, you must already have administration rights. If no user is assigned administrator privilege, then all users are allowed administration access.

With a user or group selected, you can then add or remove permissions by clicking on the appropriate check box. Click *Apply* to save the changes, or *OK* to save the changes and exit the editor.

In order to remove a user or group from this list entirely, you must remove all accesses, removing *Read access last*. The next time you open the dialog, the user or group will not be listed. If you remove Read access before any of the others, the user or group will remain in the list, but with the permissions greyed out. They won't actually be able to access the repository.

To add access rights to a user or group not already listed, click on the *All* radio button. The list then displays all users and groups defined in the access database. Assign rights by clicking on the appropriate check boxes, then click *Apply*. When you click on the *With Permissions* radio button, the added user or group is included in the list.
Assigning Repository Entity Access Rights

To view access permissions for a repository entity, select its repository and then the entity itself in the Repository Browser, then select Entity / Edit Access. The list of users and groups with access permission is displayed in the following dialog:

Figure 5-16: Repository Entity Access Rights Editor

Click on the name of a user or group to see their current permissions. Permissions are indicated in the check boxes to the right. The permissions are:

- **Read** - allows the user to view the list of revisions, and to open or load a revision. Read access is required for all other access permissions.

- **Create** - allows the user to create a new revision of the entity. **Branch Only** restricts the user to creating new branches on the revision tree, while **Any** allows the user to create "trunk" level revisions as well.

- **Destroy** - allows the user to destroy revisions of the entity.

- **Administration** - allows the user to modify the set of users allowed to access the entity and their access permissions.

To modify the access rights to an entity, you must already have administration rights. The creator of an entity initially has administration rights, by default.

With a user or group selected, you can then add or remove permissions by clicking on the appropriate check box. Click **Apply** to save the changes, or **OK** to save the changes and exit the editor.
In order to remove a user or group from this list entirely, you must remove all accesses, removing Read access last. The next time you open the dialog, the user or group will not be listed. If you remove Read access before any of the others, the user or group will remain in the list, but with the permissions greyed out. They won’t actually be able to access the entity.

To add access rights to a user or group not already listed, click on the All radio button. The list then displays all users and groups defined in the access database. Assign rights by clicking on the appropriate check boxes, then click Apply. When you click on the With Permissions radio button, the added user or group is included in the list.

Locking

When a module is locked, no one else can commit a revision of that module.

To lock a module:

1. In the Package Browser, select the module.
2. Choose Module / Edit Lock.

   An entity lock dialog opens. Text indicates whether the module is locked or unlocked. If locked, the user who locked it is identified.

3. Click the Lock or Unlock button, or Cancel.

   Only the button to change the state is enabled.

You can manually lock a module at any time if it is not already locked. You can only unlock modules you have locked.

Each time you commit, the module is unlocked, so if you want to retain exclusive control over that module, you must manually re-lock after each commit.

Regardless of locking, any user that has access to a repository can open or load a module contained in it. Locking only controls who can commit a revision.

Optimistic Versioning

Optimistic versioning is controlled by the locking policy attribute of repositories.

When optimistic versioning is set for a repository, two people can load the same revision of a module. Both can modify the revision in their respective images without being notified that another
person is working on the same base revision. The first person to finish work commits the module. The second person to commit the module is warned that a module derived from the same revision has already been committed, so the second person should integrate the changes the first person made before committing to the main revision line.

**NOTE:** Even with optimistic versioning, you can still lock a module to prevent anyone else from committing a revision of it.

### Pessimistic Versioning

Pessimistic versioning is controlled by the locking policy attribute of repositories.

Under a pessimistic versioning policy, you are notified the first time you modify a module (that is, you make a change to a definition in that module and save it) that it is stored in a pessimistic repository. If the module is not locked, you have the option of editing without locking or locking the module and editing. If the package is already locked, you are notified and given the option of editing anyway.

Here is an example of the notification dialog that appears when you make the first edit to an unlocked module stored in a pessimistic repository.

**Figure 5-17: Notification Provided by Pessimistic Repository**

![Notification](image)

Pressing the *Edit Anyway* button causes the operation to proceed (the modified definition is saved in the image) but does not change the locked status of the module. If you press the *Lock Module* button, the system tries to lock the module, then proceeds with the save operation whether or not the locking was successful. To cancel the operation without modifying the definition, press the *Abort* button.
Once a particular package is modified, no special treatment regarding committing or revisions is assumed.

If the repository is not accessible over the network when you save your first modification to a module, you are notified and allowed to choose to edit anyway. A module cannot be locked if its repository is inaccessible.

As explained earlier, you can set up pessimistic versioning when the repository is created, but you cannot switch between optimistic and pessimistic versioning after the repository exists.

If the first person to make a change to a module from a pessimistic repository chooses not to lock it (perhaps she’s just experimenting), and another person makes a change, but also declines to lock, the situation is the same as with an optimistic repository: the first person to commit does so without any notification that anyone else is working with the module. The second person who attempts to commit a revision of the same base is notified that he has integration work to do.

### Teamwork

This section contains the following topics:

- Sharing Code Without Access to a Common Repository
- Detaching from a Repository to Work Off-Site
- Editing a package without loading it into the image.
- Resolving conflicts between packages.
- Reusing code from another package.
- Loading multiple packages to resolve circularities.
- Extending classes defined in other packages.
- Integrating revisions and packages.
- Rebuilding an image.

### Sharing Code Without Access to a Common Repository

Your normal day-to-day activities are centered around repositories. Repositories are typically located on a shared file system, accessed via a network. You do not have access to these repositories if

- you are using a portable computer that you disconnect from your network,
- your network is unavailable for an extended period of time, or
- you transfer your work to your home computer system that does not have access to your network.
If you want to share code with a colleague who does not have access to your repositories, you can

- give that person a complete copy of your repository directory and all its files,
- commit to a new repository, copy that new directory structure for your colleague,
- export the modules to a file,
- file modules out,
- build a Smalltalk Link Library from a module.

There are advantages and disadvantages for each. A copy of the repository directory contains all the historical revision information, but can be very large and contain much extraneous information. Committing designated modules to a new repository for transfer is much smaller, and an exported module file is even smaller. The file-out format does not contain the revision information, but is necessary for transferring to someone without Visual Smalltalk Enterprise. Building an SLL is typically a delivery solution, but, if you include source code, would work for code distribution purposes.

**Detaching from a Repository to Work Off-Site**

When you know that your repository is going to be unavailable, you need to detach your image from the repositories. To detach your image, select *Smalltalk / Team / Detach*.

The detach operation condenses your change log, copies all source from repositories to your local change log file, and saves the image. You now have full access to the source of modules in your image without accessing repositories. If you need to transfer your work to a different computer system, do so by copying the files

- V.EXE,
- CHANGE.LOG and
- RECOVER.LOG

to your target directory. (The target computer system must have Visual Smalltalk Enterprise installed.)

Use the off-site system to continue your development as planned, including saving your image as needed. As you work, modifications to modules are marked. If you are working away from your repositories for a limited time, you probably do not need to commit your modified modules. If you are working away for an extended time, create a local repository on your off-site system and commit modified modules to that repository.
When you are ready to transfer your work back to the on-site system, copy

- V.EXE,
- CHANGE.LOG,
- RECOVER.LOG,
- and any new repositories you have created

back to your on-site system. Make sure the files are placed into your Visual Smalltalk Enterprise directory. When you want to make your changes public, commit them to the shared repository. If you have committed modules to a local repository that you want to make public, connect to the new repositories (if any), and recommit those modules to the shared repositories. As with any large number of changes, save your image after committing to the shared repositories.

**Editing a Package Without Loading it into the Image**

You can edit a package without loading it into the image by opening it for examination. When you open a package to examine it, it is not part of your executable image. You can only perform a subset of development tasks:

- Add definitions
- Delete definitions
- Rename definitions
- View the source code in that revision of the package
- Edit the source code
- Reorganize the sequence of definitions
- Commit changes and make a new revision

When a package is open but not loaded, you cannot:

- Execute source code in the package
- Check variable references for validity
- Check for conflicts
- Perform any deeper debugging operations that depend upon execution or the semantics of the source code
- Use it to build a Smalltalk library

You can also open a file-in for examination. If you specify a file-in, using the *File* button in the load dialog, Visual Smalltalk Enterprise first converts the file-in to a package. Then, it opens the newly created package in the same manner.

This facility is useful to eliminate conflicts if you cannot load a package because a conflict has occurred.
CHAPTER 5 Team Programming Tasks

Resolving Conflicts

If one of the definitions in the package has the same name as one of the definitions in a package already loaded in your image, a conflict will arise. A conflict also arises if you try to load a package defining a *loose method*, which is a method defined for a class that is not in the package, and the class is not in the image. If a conflict occurs, you will not be able to load the package until you resolve it by performing one of the following actions:

- Delete one of the conflicting definitions
- Rename one of the conflicting definitions
- Define the class to which the loose method belongs
- Redistribute the contents of the affected packages. For example, you might make a new package with only some of the code from both packages in it.

If a package cannot be loaded in your image because of a conflict, you can open one or both of the packages, so you can edit the definitions to resolve the conflict.

**NOTE:** For a list of conflicts and instructions on how to resolve them, see Appendix A.

To change the name of a class, global variable, or pool in your image:

1. In a Package Browser, select the global you want to rename.
2. Pull down the *Global* menu and choose *Rename...*.
   
   You are prompted for the new name.
3. Type the new name and click *OK*.

To change the name of a class, instance, class instance, or pool variable in your image:

1. Browse references to the variable and leave the browser open for Step 3.
2. In the Package Browser, select *Variable / Rename...*.
3. In the browser with the references to the old variable, change all references to the old variable so that they use the new name.
To change the name of a class, global variable, or pool in a package you want to load:

1. In a Package Organizer, choose Package / Open....
   A dialog appears, letting you specify the package, revision, and repository to open.
2. Select the revision you want to work from.
   The new revision appears in the package list. The definitions it contains appear the definition list.
3. Select the class, global variable, or pool you want to rename.
4. Choose Definitions / Rename....
   You are prompted for the new name.
5. Type the new name and click OK.

To change the name of a method in the package you want to load:

1. In a Package Organizer, choose Package / Open....
   A list of the revisions of the package appears.
2. Select the revision you want to work from.
   The new revision appears in the package list. The definitions it contains appear the definition list.
3. Select the method you want to rename.
   Its source appears in the contents pane.
4. Select the old name of the method in the contents pane and type the new name.
5. Save your change.
6. Remove the old method.

To change the name of class, instance, class instance, or pool variable in the package you want to load:

1. Rename the item.
2. Change all references to the old variable so that they use the new name.

To redistribute the contents of the packages, first decide exactly how you want to refactor the package functionality. Then:

1. In a Package Organizer, choose Package / Open....
   A dialog appears, letting you specify the package, revision, and repository to open.
2 If you need to, open another Package Organizer and create a new, empty package.

3 Select the first definition you want to move. Drag it over to the package list until the package to which you want to move it is highlighted. Then, drop it into that package.

4 Continue dragging and dropping the definitions until you have reorganized the packages as you want.

5 Commit new revisions of the packages you have changed.

6 Load the new package(s) as required.

Reusing Code from Another Package

Load the package whose code you want to reuse. The code then becomes part of your application. If one of the definitions in the package has the same name as one of the definitions in a package already loaded in your image, a conflict is detected.

Loading Multiple Packages to Resolve Circularities

While you are developing your packages and defining classes in them, it is possible to end up with circular dependencies. In many cases, rather than redistribute class definitions, you can load and initialize these packages by being careful of their load order. Visual Smalltalk Enterprise handles most forward references, so circularities usually are not a problem. Problems occur only when packaging separates classes from their superclasses, because superclasses must be defined before subclasses can be created.

If such dependencies manifest themselves in initialization code, you can solve the problem by deferring initialization. Unmark the check box in the Load dialog to defer initialization for the package containing dependent classes. You must then initialize packages after they are all loaded.

Example 1

Suppose **Package One** contains class **ClassA** and **Package Two** contains classes **ClassB** and **ClassC**, and **ClassB** is a subclass of **ClassA**. Suppose also that **ClassA** requires **ClassC** in order to be initialized (some class variable must initialize to an instance of **ClassC**).

In this situation, **Package Two** must be loaded before **PackageOne** is initialized (**ClassA** needs **ClassC**), but **ClassA** must be loaded before **ClassB** is loaded (**ClassB** is a subset of **ClassA**).
Solution:

1. Load Package One, but don’t initialize.
2. Load Package Two and initialize.
3. Manually initialize Package One.

Other dependencies cannot be solved by controlling the initialization order. The most difficult are hierarchical dependencies, in which subclass-superclass relationships are involved.

Example 2:

Suppose Package One contains ClassA and ClassN, and Package Two contains ClassB and ClassM. Suppose also that ClassN is a subclass of ClassM, and ClassB is a subclass of ClassA.
Because ClassN is a subclass of ClassM, Package Two must load first. But, because ClassB is a subclass of ClassA, Package One must load first. This is impossible.

Solution:

Combine both packages into a single cluster, then load the cluster.

1. Create a new open cluster with Cluster / New... in the Cluster Organizer.
2. Add Package One to the cluster: Use Specification / New... to add a new specification for the package.
3. Add Package Two to the cluster: Use Specification / New... to add a new specification for the package.
4. Load the new cluster: In the Cluster Organizer, select the new cluster and choose Cluster / Load.
5. Commit the new cluster, if appropriate.

You can create clusters to load multiple packages just for the convenience of loading several modules at once.

Extending Classes Defined in Other Packages

You can extend another team member's class in one of two ways, depending upon the ultimate purpose of the extension:

- Create a new revision of the package that includes the extension. If you expect most or all users of the package to need or want your extension, use this technique.

- Organize the extension into another package. If your extension is specific to your application and some users of the package will not need or want it, use this technique. (This technique is also useful if you do not have permission to modify the package.)

- Alternatively, if the extension you need to make is a method, select the package which will hold the extension. Then use the Method / New... to add the method to a class from another package.
To make the extension in another package:

1. Select the package in the Package Browser—the one you want to hold the extension.

2. Extend the package as needed. The work you do becomes part of the selected package.

   or

   Move extensions from the original package into the second package using drag and drop.

### Integrating Revisions and Packages

Visual Smalltalk Enterprise lets you directly compare any two revisions of any two packages. Both packages can be loaded or opened for examination, although, in order to test your code meaningfully, each will ultimately have to be loaded in an image.

The comparison tools can make the integration process easier to manage, but the integrator still needs to be thoughtful and methodical, and must have the goal firmly in mind before beginning. Is the purpose to create a third package incorporating some portions of each of the other two? Is the purpose to make a new revision of one of the two packages, or new revisions of both? Plan the work of integration carefully before you begin.

**NOTE:** You can make the work of the integrator easier by minimizing changes to the base class library. Extensions to these classes, if necessary, should be included in a separate package.

Also, you can make your application easier to maintain if you create another package to hold an extension, instead of creating a new revision of the extended package. More fine-grained units with shallower histories will ordinarily be easier to integrate and maintain than fewer, larger units with complex histories.

Once you have the comparison browser(s) open, you can determine the definitions you want to keep, those you want to move or delete, and those that will need modification. In the Package Organizer or Comparison Browser, drag definitions from one package to another as needed. If two definition names conflict, use the Package Organizer to edit the text of one of the definitions as necessary.
Rebuilding an Image

You might need to rebuild an image for a variety of reasons:

- In order to test your application thoroughly, you must do what your users will do: load the application into a base image.
- You might need to upgrade to a newer version of the base image or development environment.
- A rebuilt image provides a known starting point, in case you fear that your application has changed something critical.

If you maintain an up-to-date configuration cluster, rebuilding an image should be as simple as loading the configuration cluster. If you do not want to use a configuration cluster, you can generate a build script to help with the rebuilding process.

In order to rebuild an image, you need a list of all the modules you need to load into the base image, and the order in which the packages must be initialized. You need to know the correct revision number of each package. You also need a fresh copy of the base image, V.EXE, from the VBACKUP directory.

When you have such a list, execute a fresh copy of the base image. Load the packages, and then initialize them in order. If this is a long or complex procedure, you may want to use an automatically generated build script to accomplish the task.

Visual Smalltalk Enterprise can generate a build script in a workspace, containing the name of each package loaded in your system, with the revision number currently loaded. This script may specify the revisions of packages you want, or you may have a package loaded that you do not want to include. In this case, you can edit the script as required. To create the build script:

1. In the Transcript, choose Smalltalk / Browse Packages to open a Package Browser.
2. If you have modified system packages, move all modified definitions they contain to another package if you want to load them into another image.
3. Commit each package in your image to a repository, if you have not done so since they were last modified.
4. Choose Smalltalk / Team -> Generate Build Script.

The script appears in a workspace.

If modifications have been made to system packages, or if a package has never been committed to a repository, or if changes have been made to a package since it was last
committed, performing this step produces the following notifier:

**Cannot generate a complete build script. Proceed anyway?**

Answering yes creates a build script containing all packages in your image that have been committed to a repository, using the revision number of the last commit operation. The script does not represent the complete state of your image.

5. Edit the script as necessary.
6. Save the build script to a file.
7. Save your image and exit.
8. Start up a fresh Visual Smalltalk Enterprise image.
   
   A clean V.EXE is installed by default in the VBACKUP directory.
9. Open the build script workspace.
10. Select the script and Do It.

If you are still developing, you will probably load the packages into a fresh copy of the development environment. However, when you have finished development, you must test the system as your users will receive it.

### Delivering An Application

This section discusses the tools and issues involved in building a deliverable application in Visual Smalltalk Enterprise. For a discussion of application delivery in general, refer to the *Visual Smalltalk Enterprise User’s Guide*.

Application Smalltalk code is delivered either as Smalltalk link libraries or as packages. Packages are useful only for distributing source code to other Visual Smalltalk Enterprise developers.

### Distributing Code in Packages

A package is the intended unit of distribution among developers. There are two forms in which you can distribute packages. You can either:

- export your modules to a file in package format, or
- distribute the code in a repository
To save a module in package format:

1. Load the desired revision of the module.
2. Select Module / Export ... in the Package Browser.
3. In the file dialog, select a directory and file name for the package, and click OK.

Notice that the default file name extension is .PKG.

The resulting file can be distributed to other developers. They can load the package using File / Install or Module / Load, and the installation process will migrate the package into their image.

In order to distribute a repository, create a new revision of the package in a distribution repository so the distribution repository contains only the final revision of the package.

To distribute a repository:

1. Load the desired revisions of your modules.
2. Copy the current revision to a new repository using Module / Copy... menu in the Package Browser.

   The Cluster Organizer and Package Organizer also provide Copy... items in their Cluster and Package menus, respectively.

   You could also simply commit your modules to the new repository.
3. Copy the repository to a shared drive or other distribution media, such as diskettes.

Your team members can access the repository from the shared drive or distribution medium as usual.

Distributing Code without Packages

To distribute a module to a Visual Smalltalk developer who does not have Visual Smalltalk Enterprise, either build Smalltalk libraries from your packages or file out the code.

Initializers for global variables, pool variables, classes, and ad hoc items cannot be defined outside of the development environment. If your package contains these items, global variable initializers, pool variable initializers, and ad hoc initializers are changed to Do Its when you save the package as a file-out. Class initializers become methods whose names take the form classNameInitializer.
Building Smalltalk Libraries

After your application is implemented, integrated, and tested, it is ready to distribute to users.

Visual Smalltalk applications are distributed as a Smalltalk image file, several run-time support files, possibly some Part files, and a collection of Smalltalk libraries. Instructions for putting these together to make a stand-alone application are provided in the Visual Smalltalk Enterprise User’s Guide.

Here we will focus on building Smalltalk libraries from modules.

Build from a Clean Image

When building a Smalltalk library for customer delivery, it is important to make the final build from a clean image. Extra objects tend to accumulate during development, which can inflate the image size or require libraries that you cannot deliver with a runtime application.

You don’t have to start from the clean V.EXE each time, though. It is often useful to have one or more baseline images, such as an application baseline image and a development baseline image, that have only the libraries you need at that time bound in. The development baseline image might be a Visual Smalltalk Enterprise image with the development installation of your favorite set of third party tools. The application baseline image would contain the necessary runtime versions, without any development tools.

Each baseline image requires three matched files: V.EXE, CHANGE.LOG, and RECOVER.LOG. These three files describe the current image, including source code for additions you make to the image. When you make an image a baseline for further work, make sure you include all three of these files.

Building a Smalltalk Library From a Module

Visual Smalltalk Enterprise makes it easy to build a Smalltalk library from a package or cluster, by using the Module / Build Library... menu item in the Package Browser.

This tool operates on a single module, so if your application is comprised of more than one package, you’ll want to compose these packages in a cluster to build a Smalltalk library from them.

To create a Smalltalk library from a module:

1. In your development image, create a configuration cluster containing your final application modules.
2 Commit all application modules in your development image.

3 Use Smalltalk / Browse / Unresolved to make sure there are no unresolved variables.

4 Make sure the **unpackaged** package is empty.

5 Update the application configuration module so that it contains a complete specification of each revision, and commit it.

If you get any conflicts, resolve them, commit the modules again, and start over with a fresh image until you can build the application without errors. Check Unresolved again to make sure that you have a clean build.

Only code referenced by the configuration module is actually written to the Smalltalk library, but following this process formally ensures that your application won’t have any dependencies on code outside the module.

Don’t forget to execute your application at appropriate times during the build process to make sure that it still works.

6 In a Package Browser, select Module / Build Library...

Figure 5-18: Build Library Dialog

The Build Library dialog opens. This is the Visual Smalltalk Enterprise equivalent of the Smalltalk Library Builder, designed to write a Smalltalk library from a module. The result is an SLL file that contains only your application.
7 Edit the description.

The description is a comment describing your application.

8 Set the Include Source checkbox to indicate whether you want source included in the SLL file.

When building a runtime application, you usually do not want source in the file, to make the application as small as possible.

9 Click the Build... button.

If you want to customize the build script, click Customize instead, and see Building Smalltalk Libraries below.

10 Select a directory and file name for your application library file, then click OK.

The utility then creates the library from the module and writes the file (files, if source is separate).

See Smalltalk Link Libraries in the Visual Smalltalk Enterprise Language Reference for further information about creating a Smalltalk link library and delivering applications.

Customizing a Build Script

If you click the Customize... button in the Build Library dialog, a workspace is opened containing a script for building a Smalltalk library from the selected module.

You may edit the script to customize how the library is built. Once the script is correct, you select the entire script and evaluate it with Do It. Since the script is in a workspace, you may also save it for future reuse.

The first few lines set up the environment for creating a library. It binds in the library builder, which is not normally in the image. It then assigns a reference to the module to the variable 'module' which is used as source for the library.

Next, it creates an instance of TeamLibraryInformation, which is used to control building the library. This is the object to which you send messages to customize how the library is built.

    info name: 'MYMODULE'.

The name: message sets the file name that will hold the library. You should set the name without the extension because the .SLL extension is automatically appended.

    info targetDirectory: 'D:\MYDIR\'.

This is the directory in which the library will be created.

```
info sourceInFile. "or #sourceSeparate."
```

You can send the message sourceInFile if you wish to embed the source in the .SLL file. To separate the source, send the message sourceSeparate. If you choose to include source in a separate file, two files will be created, one with an extension of .SML. In either case, you have control about which source is actually included in the library at the definition level. (See the next line.)

```
info includeSourceTest: [def | true].
```

The `includeSourceTest:` message stores a block that is evaluated with each definition as an argument to determine if source should be included in the library. The default shown here is to include all source. To hide all source, simply change true to false in the block. If you wished to use a more sophisticated policy, such as hiding source in certain categories, arbitrary Smalltalk code can be inserted in the block. Simply make sure that the last statement in the block evaluates to a Boolean expression.

```
info module: module.
```

This statement actually specifies the module to be used as the source for the library. You will not normally customize this line.

```
info description: 'Repository: My Repository
    Revision: 0.5
    Author: Brian
    Package: My Module
    Revision Comment: A Descriptive Comment.'.
```
A library has a description field that can store arbitrary text. The default string is based on the module information, but you are free to change it as you wish.

```smalltalk
info staticInitializationTest: [:def | false].
```

Definitions held in the module can have associated initialization expressions. One typical use is for a class definition have an initializer that reads `self initialize`. An **initialize** class method sets up state required for the class to function properly (for example, setting up class variables). This statement controls whether initialization is done when the library is built (statically) or when the library is bound by the application using it (dynamically). If static initialization is selected, the current values of the loaded definitions are used. As with source control, you can control this with broad strokes simply by editing the Boolean in the proposed block, or you can control initialization at the definition level by writing more complicated code.

```smalltalk
builder := info makeLibrary.
```

This statement actually builds the library as you have specified.

### Filing Out a Module Source Code

You can also deliver your application in source form by filing out all your packages. Visual Smalltalk Enterprise provides several ways to do this.

**To file out your application:**

- Select **Smalltalk / Team -> File Out All...** to produce a file-in format file for all code in all loaded packages except the system packages.

  or

- Create a configuration cluster then, in a Package Browser, select this configuration cluster that comprises your application and choose **Module / File Out**.

  **Module / File Out** only writes out source for the selected module in the Package Browser. Unless you are careful to check for unwanted packages, it is much safer than **Smalltalk / Team -> File Out All...**

Once your application is in file-in format, you can install it in a Visual Smalltalk image. If you save your image after installing it, you can distribute the binary .EXE file by shipping the V.EXE file.
CHAPTER 6

Extending the Tool Interface

This chapter describes the Application Programming Interface (API) that is used by the Visual Smalltalk Enterprise tools. You can use this interface to create your own custom tools and browsers, as well as automatically perform sequences of operations from Smalltalk scripts.

The semantic components of the Visual Smalltalk Enterprise environment—such as packages, clusters, definitions, specifications, and revisions—are described in chapter 2. You should be familiar with these components and have a firm grasp of the Visual Smalltalk Enterprise environment before attempting to use the application program interface.

The API is presented as a set of classes and public methods. The interface consists of two major components:

- the Semantic interface and
- the Repository interface.

The semantic interface is used by all of the extended Visual Smalltalk Enterprise tools and browsers. The semantic interface, in turn, uses the repository interface to create and fetch revisions of entities. The tools also directly access the repository interface for repository operations and browsing.

The repository interface is not specific to Visual Smalltalk Enterprise; you can use it to store components that have nothing to do with Visual Smalltalk Enterprise components.

One of the clusters that is present in your image is the Team/V Interface cluster. It contains packages with the implementations for the repository interface and the semantic interface.

The code for tools and browsers uses the public API. See chapter 2 for instructions for installing the tools source cluster.

The figure below shows the relationships between the various parts of Visual Smalltalk Enterprise.
You can write new tools and access the semantic functionality through the Visual Smalltalk Enterprise API. You can also write tools and scripts that use the repository interface to browse, query, and update repository contents.

You extend the set of entity types that are stored in a repository by defining new repository entity types, such as arbitrary text strings. You can then create revisions of your user-defined entities as well as those defined by Visual Smalltalk Enterprise.

The remainder of this chapter discusses concepts underlying the semantic and repository interfaces, and provides examples of the use of these interfaces to accomplish various tasks.

The Visual Smalltalk Enterprise Encyclopedia of Classes describes all of the public behavior of the semantic and repository interfaces. All of the information contained there may be found in the image.

You should only use the public methods in the interface. Any method marked private should not be used, as its implementation and/or functionality may change in future releases.

**Semantic Interface**

One way to use the Visual Smalltalk Enterprise interface is to create a tool box class. In that class you can add methods that perform the various operations that your team needs. The directory SAMPLE\TEAMVAPI contains several examples of the Visual Smalltalk Enterprise interface, which demonstrate additional uses of Visual Smalltalk Enterprise functionality. These can serve as starting points for you to create your own tools.
TeamVInterface Service

The class **TeamVInterface** is the foremost API class in the semantic subsystem. It provides access to the loaded modules and definitions in your image. The class **ToolInterface**, an auxiliary class, provides access to open modules and definitions. **ToolInterface** also provides access to tools, browsers and editors for viewing and editing modules and definitions. For both of these interface classes, the main entry point is the class method **current**.

The semantic API is supported and maintained from release to release. The class **ToolInterface** is not part of the semantic API, and its behavior is not guaranteed to be supported from release to release. However, we will discuss **ToolInterface** with respect to its role in providing access to open modules, and as a utility for displaying the results of the queries to the semantic subsystem.

The components of the semantic interface are analogs of the concepts that are explained in chapter 1, *Key Concepts*. Please review that chapter if you are unfamiliar with its content. The definitions that make up your program are accessed using various types of **Handle** objects.

The following table outlines the mapping between semantic components and the programmatic interface:

<table>
<thead>
<tr>
<th>Team Components</th>
<th>Programmatic Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>current set of loaded packages</td>
<td>SubsystemHandle</td>
</tr>
<tr>
<td>package</td>
<td>PackageHandle</td>
</tr>
<tr>
<td>cluster</td>
<td>ClusterHandle</td>
</tr>
<tr>
<td>Definitions</td>
<td></td>
</tr>
<tr>
<td>class</td>
<td>ClassDefinitionHandle</td>
</tr>
<tr>
<td>method</td>
<td>MethodDefinitionHandle</td>
</tr>
<tr>
<td>class initializer</td>
<td>ClassInitializationDefinitionHandle</td>
</tr>
<tr>
<td>instance variable</td>
<td>VariableDefinitionHandle</td>
</tr>
<tr>
<td>class variable</td>
<td>VariableDefinitionHandle</td>
</tr>
<tr>
<td>class instance variable</td>
<td>VariableDefinitionHandle</td>
</tr>
<tr>
<td>pool usage</td>
<td>VariableDefinitionHandle</td>
</tr>
</tbody>
</table>
Definitions

<table>
<thead>
<tr>
<th>Definition</th>
<th>Definition Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>global variable</td>
<td>GlobalVariableDefinitionHandle</td>
</tr>
<tr>
<td>pool</td>
<td>PoolDefinitionHandle</td>
</tr>
<tr>
<td>pool variable</td>
<td>PoolVariableDefinitionHandle</td>
</tr>
<tr>
<td>ad hoc initializer</td>
<td>InitializationExpressionDefinitionHandle</td>
</tr>
<tr>
<td>group definition</td>
<td>DefinitionGroup</td>
</tr>
<tr>
<td>local definition</td>
<td>DefinitionVersionHandle</td>
</tr>
<tr>
<td>specification</td>
<td>SpecificationHandle</td>
</tr>
</tbody>
</table>

Typically there is a one-to-one mapping between concept and interface components. The exceptions are some parts of a class definition. An instance of VariableDefinitionHandle represents each of the components for instance variables, class variables, class instance variables and pool usages. With the exception of SubsystemHandle, all the objects in the table can exist in either the loaded or open environment.

Subsystem

There is only one instance of SubsystemHandle in the system, representing the loaded environment. This object describes the name space for the current Smalltalk system, a combination of user-defined names and system-defined names. You send messages to this instance to access the loaded modules in your image. The loaded modules in your image are accessed by sending messages to the instance of SubsystemHandle returned from evaluating the expression

TeamVInterface current.

The loaded subsystem contains a top level cluster, displayed in the tools as "*top level*". This cluster describes the contents of the loaded system in terms of packages and clusters. To access the top level cluster evaluate

TeamVInterface current cluster

This cluster directly or indirectly contains all loaded modules.
Enumerating over Subsystem Components

Enumeration messages, such as `packagesDo:` and `clustersDo:`, can be sent to the subsystem to examine the loaded modules in the image. All the loaded definitions in the image, regardless of how they are distributed in packages or how deeply nested the clusters, can be accessed with the message `allDefinitionsDo:`. All global definitions (classes, pools and global variables) can be accessed with the message `globalDefinitionsDo:`.

The following expression can be used to collect all the class definitions in the system. We utilize the classification message `isClass` to distinguish class definitions from other global definitions.

```smalltalk
classDefinitions := OrderedCollection new.
TeamVInterface current globalDefinitionsDo:
    [:each | each isClass ifTrue: [classDefinitions add: each]]
```

The specifications from the top level clusters are accessed with the message `specificationsDo:`. The following expression collects the specifications of packages at the top level that are not organized into clusters.

```smalltalk
shouldBeClustered := OrderedCollection new.
TeamVInterface current specificationsDo:
    [:each |
        (each specifiedType = 'package') ifTrue: [shouldBeClustered add: each]].
```

All loaded specifications can be examined with the message `allSpecificationsDo:`. This expression finds all specifications that reference the repository named 'temporary', and should be recommitted to a permanent repository.

```smalltalk
questionable := OrderedCollection new.
TeamVInterface current allSpecificationsDo:
    [:each |
        each repositoryName = 'temporary' ifTrue: [questionable add: each]]
```

Subsystem Searching

There are a number of ways to search the entire subsystem. We can search for a global definition with a particular name with the following expression, which returns an instance of `ClassDefinitionHandle` for the class `Collection`, or `nil` if it is not found. Note that global names are symbols.
collectionDefinition := TeamVInterface current
definitionOfGlobalNamed: #Collection.

SubsystemHandle responds to three global-specific searching messages, one for each type of global in the system. The messages definitionOfClassNamed:, definitionOfGlobalVariableNamed:, and definitionOfPoolNamed:, can be used to narrow search results. If a matching definition does not exist, the searching messages return \textbf{nil}.

In addition to globals, you can also search for a particular method:

\begin{verbatim}
TeamVInterface current
definitionOfMethod: #select:
inClassNamed: #Collection
meta: false
\end{verbatim}

In this case we are searching the for instance method \texttt{select:} in the class \texttt{Collection}. The first argument to \texttt{definitionOfMethod:} \texttt{inClassNamed:meta:} is the a symbol for the method selector \texttt{select:}. The second argument, the name of the class, is also a symbol. The third argument is a boolean specifying whether to search the class or the metaclass. That is, whether to seach instance or class methods. If the \texttt{meta:} argument is \texttt{false}, an instance method is desired.

In addition to searching for a particular implementation of a method, we can search for all implementors of a message. SubsystemHandle responds to expressions of the form

\begin{verbatim}
TeamVInterface current implementorsOf: #select:
\end{verbatim}

where the argument to \texttt{implementorsOf:} is a symbol. The message \texttt{sendersOf:} is used in an analogous fashion to return all the senders of a message. Both messages return a group of definitions, which may be empty.

The message \texttt{referencesTo:} is also a searching method, returning a group of definitions that reference the argument, another definition. The referencing definitions are a heterogeneous collection: they can be method, class or initializer definitions. In this example, we search for references to the pool \texttt{ColorConstants}.

\begin{verbatim}
subsystem := TeamVInterface current.
subsystem referencesTo: (subsystem
definitionOfPoolNamed: #ColorConstants)
\end{verbatim}

SubsystemHandle also supports queries for modules by name. The relevant set of messages are \texttt{packageNamed:}, \texttt{clusterNamed:} and \texttt{moduleNamed:}. The message
moduleNamed: finds either a package or cluster with a matching name in the loaded system. The argument for these messages is a string. For example, the following expression returns a matching instance of PackageHandle, ClusterHandle or nil if no matching module could be found.

```
pkgHandle := TeamVInterface current
    moduleNamed: 'Business Model'
```

**Migration**

The primary means of editing a subsystem is by migration, a process that is used to add, delete and replace modules in a subsystem. Even the creation of new packages and clusters uses the migration process.

To create a new loaded package, send the message createPackageNamed: to an instance of SubsystemHandle. A similar message, createClusterNamed:, is used to create a new loaded cluster. The names of modules are strings, so the arguments to these messages are strings. Module creation will fail if the names of the loaded modules are not unique. The following expression creates a new package and a new cluster.

```
newCluster := TeamVInterface current
    createClusterNamed: 'Model Configuration'.
newPackage := TeamVInterface current
    createPackageNamed: 'List Extensions'
```

An existing package or cluster can be removed from the loaded system with the message unloadModule:. The following expression locates a loaded module, and then unloads it.

```
subsystem := TeamVInterface current.
module := subsystem moduleNamed:
    'Customized Text Control'.
subsystem unloadModule: module
```

There is a corresponding message, loadModule:, that is used to load an existing module. Because the typical way to reference an unloaded module is via the repository interface, this example has references to concepts discussed in the Repository Interface section. It also makes use of the class ToolInterface. In this example, we locate revision 0.2 of the package Customized Text Control, and load it into the image.
aPackage := ToolInterface current
revision: '0.2'
ofPackageNamed: 'Customized Text Control'
fromRepositoryNamed: 'Common UI Extensions'.
subsystem := TeamVInterface current.
subsystem loadModule: aPackage.

Often developers want to replace an old revision of a module with a new revision. The message migrateFrom:to: and its cousin migrateFromList:toList: are replacement operations. A migrate operation is different from an unload and load operation in sequence because the migrate operation maintains instances of classes in the package, while the unload operation destroys them. For example, suppose you have an instance of EnhancedTextControl, a class in your package. If you unload the package, the instance is mutated to become an instance of DeletedClass. Even if the class is recreated by loading another revision of the package, the instance is effectively destroyed. However, if you migrate to a new revision of the package, the instance maintains it identity and no special work is required by developers. Migration is also important when a developer needs to redefine a class and there are subclasses in other packages.

In the following example we use the message migrateFrom:to: to replace the current revision of the package Customized Text Control with the 0.3 revision.

    subsystem := TeamVInterface current.
    packageName := 'Customized Text Control'.
    loadedPackage := subsystem moduleNamed: packageName.
    newPackage := ToolInterface current
revision: '0.3'
ofPackageNamed: packageName
fromRepositoryNamed: 'Common UI Extensions'.
    subsystem migrateFrom: loadedPackage to: newPackage.

Loading, unloading and migrating messages have a return value that is an instance of MigrationResult. This result can be queried for the success of the migration operation, and for conflicts if it does not succeed. The dialog for displaying conflicts can be accessed via ToolInterface.

In the following variation, we account for the fact that the migration operation might fail. The result of migrating is assigned to the variable result, and queried with the message wasSuccessful. If the operation fails, ToolInterface is used to bring up a dialog on the conflicts.
subsystem := TeamVInterface current.
packageName := 'Customized Text Control'.
loadedPackage := subsystem moduleNamed: packageName.
newPackage := ToolInterface current revision: '0.3'
          ofPackageNamed: packageName
          fromRepositoryNamed: 'Common UI Extensions'.
result := subsystem migrateFrom: loadedPackage to: newPackage.
result wasSuccessful
  ifTrue: [Transcript cr; show: 'Migration of ', packageName, ' was successful'
  ifFalse: [ToolInterface current reportConflicts: result fromOperation: 'Migration of ', packageName]

Migration options offers finer control over the process of migration. The message migrateFromList:toList:withOptions: to a subsystem takes an instance of MigrationOption as the third argument. The first and second arguments are list of modules to be migrated. Working with lists of modules is more powerful than working with individual modules because you can deal with inter-module dependencies better. For example, if definitions have been redistributed among packages, migration of one package at a time will result in conflicts, but migration of all packages involved in the rearrangement will allow the migration to proceed without error.

The tools create default migration options based on the conflict resolution preference that controls whether to overwrite base modules, but you can create your own options. In this example, we customize the migration operation by creating our own list of packages that can be overwritten. The message modifyPackageList: to an instance of MigrationOptions sets the list of overwritten packages. The variable subsystem is set to the current loaded subsystem, and the variables newPackages and oldPackages contain collections of unloaded and loaded modules.

overwritten := OrderedCollection new.
overwritten add: subsystem unpackagedPackage.
(subsystem moduleNamed: 'Visual Smalltalk Enterprise')
  allModulesDo: [:each | each isPackage ifTrue: [overwritten add: each]].
options := MigrationOptions new.
options modifyPackageList: overwritten.
result := subsystem
  migrateFromList: oldPackages
toList: newPackages
withOptions: options.
result wasSuccessful
  ifTrue: [Transcript cr; show: 'Migration from script
  was successful']
  ifFalse:
    [ToolInterface current
     reportConflicts: result
     fromOperation: 'Migration from script']

1.4 Initialization

The subsystem maintains the initialization order of all loaded packages in the system. The entire subsystem is initialized with the menu item Smalltalk/Team/Initialize All. You can also initialize the entire system with the expression

```
TeamVInterface current performInitialization
```

All semantic objects respond to the message `performInitialization`. You can initialize clusters, packages and individual definitions. Because clusters do not maintain the initialization order of packages contained by the cluster, initialization of a cluster translates into initialization of contained packages in an arbitrary order.

You can examine the initialization order of the loaded packages with the messages `initializationOrder`, `packageAfter:`, and `packageBefore:`. You can rearrange the order with the messages `initializationOrder`, `move:after:` and `move:before:`. The moving messages allow users to rearrange the initialization order by specifying adjacent packages to a particular package, either before or after the particular package.

Classification and Testing

The semantic subsystem supports several classification distinctions. Any object can be queried to see if it is a definition with the message `isDefinition`. Modules are not definitions and return `false` from this message, but every kind of definition returns `true` from this message. Any object can be queried to see if it is a module with the `isModule` message.

Several messages are used for testing definitions. The message `hasSource` returns `true` if the receiver has source code associated with it. The message `hasInitialization` returns `true` if the receiver
has some definition that will be evaluated during initialization. The testing message `canBeEdited` is used to query whether it is possible to edit a definition. A side-effect of this message is to attempt to lock the receiver’s package if it is stored in a pessimistic repository.

**Modules**

The two kinds of modules are clusters and packages. Every module has a name, and responds to a request for its name:

```
packageName := pkgHandle name
```

Modules contain a reference to their repository object if the module has been committed. The message `repositoryName` returns the name of the repository or `nil`. The messages `repository`, `repositoryEntity` and `entityRevision` return repository objects that are associated with the receiver, or `nil` if the module has never been committed. For a more detailed discussion about these repository objects see the section titled `Repository Interface Concepts` later in this chapter.

For open modules, we send messages to `ToolInterface`. If we wanted to reference loaded module we would send messages to `TeamVInterface`. The following expression accesses first a loaded module via `TeamVInterface`, and then an open module via `ToolInterface`. The messages for accessing loaded packages are different from those used to access open packages.

```
loadedPackage := TeamVInterface current packageNamed: 'Design Artifacts'.
openPackage := ToolInterface current openedPackageNamed: 'Design Artifacts'.
```

**Module Components**

`PackageHandle` has specialized accessing messages related to definitions contained by packages. The message `definitions` returns a collection of the top level definitions stored in the receiver. The message `allDefinitions` returns directly and indirectly contained definitions. For example, `definitions` may return a class definition, but `allDefinitions` would return the class definition, the method definitions in the class, the instance variable definitions, etc.

Clusters contain specifications that can reference resolved modules. `ClusterHandle` supports two specialized accessing messages, `specifications` returns a collection of specifications contained by the receiver, and `modules` returns a collection of resolved modules that are referenced by the cluster’s
specifications. These collections match because some specifications might be unresolved due to optional or conditional inclusion. The following expression returns the resolved modules referenced by the cluster.

\[
\text{resolvedModules := aClusterHandle modules}
\]

### Searching through Modules

Instances of `PackageHandle` and `ClusterHandle` respond to the same searching messages as subsystems, but limit the scope of the search to the receiver. This way you can search a module and all its contained modules without searching the entire system, which is useful if you want to ignore information outside a specific area.

For example, you might want to check just your modules for debugging statements before integration. The Visual Smalltalk Enterprise user interface supports interactively searching the entire subsystem with the `Smalltalk/Browse/Senders of...` menu item. The following expression narrows the search by querying the Business Model cluster for senders of `halt`.

\[
\begin{aligned}
\text{cluster := TeamVInterface current} \\
\text{moduleNamed: 'Business Model'.} \\
\text{cluster sendersOf: #halt}
\end{aligned}
\]

After searching, you can display the results in a form suitable for editing. To do this, we use the utility class `ToolInterface`. The following revised expression searches for senders of `halt` and opens a browser:

\[
\begin{aligned}
\text{moduleName := 'Business Model'.} \\
\text{selector := #halt.} \\
\text{cluster := TeamVInterface current} \\
\text{moduleNamed: moduleName.} \\
\text{senders := cluster sendersOf: selector.} \\
\text{ToolInterface current} \\
\text{browseMethods: senders} \\
\text{highlight: selector asString} \\
\text{labeled: 'Senders of ', selector asString, 'in ', moduleName} \\
\text{ifNone: 'No senders of ', selector asString}
\end{aligned}
\]

The result of senders includes instance method definitions, class method definitions, class initializer definitions and ad hoc initializers.

`ToolInterface` provides access to tools for viewing and editing modules and definitions. It also provides access to open modules. Open modules can be examined and edited, but the code in these modules...
modules cannot be executed until they are loaded. Evaluating the following expression searches the open package Business Model Extensions for senders of `halt`.

```smalltalk
moduleName := 'Business Model Extensions'.
selector := #halt.
package := ToolInterface current
    openedPackageNamed: moduleName.
senders := package sendersOf: selector.
ToolInterface current
    browseMethods: senders
    highlight: selector asString
    labeled: 'Senders of ', selector asString, 'in ', moduleName
ifNone: 'No senders of ', selector asString
```

The message `referencesTo:` takes a definition as an argument, and searches the receiver for references. The message `referencesTo:` sent to a cluster searches the packages referenced by the cluster, whether they are directly or indirectly referenced by the cluster.

### Enumerating over Module Components

Modules also have messages for enumerating over their contents, but the contents of modules is different based on the type of modules. Clusters contain specifications and packages contain definitions. The semantic API provides shortcuts for enumerating over indirectly contained items. For example, clusters indirectly contain definitions and modules respond to enumeration methods to help you enumerate over indirectly contained definitions. The `all` prefix in messages to packages, clusters and subsystems implies enumeration over both direct and indirectly contained components.

Let’s look at a few examples. The first expression computes the total number of methods in the loaded system, including methods that are part of the base product. It enumerates through directly and indirectly contained methods by sending the `allDefinitionsDo:` message to an instance of `SubsystemHandle`.

```smalltalk
"total number of methods in the loaded system"
allMethodCount := 0.
TeamVInterface current allDefinitionsDo:
    [:each | each isMethod ifTrue:
        [allMethodCount := allMethodCount + 1]].
```
The second enumeration example counts the number of methods in the Business Model package, regardless of whether they are directly or indirectly contained by the package. The message `allDefinitionsDo:` is sent to an instance of `PackageHandle`.

"total number of methods in the model package"

```smalltalk
modelMethodCount := 0.
(TeamVInterface current
 packageNamed: 'Business Model')
allDefinitionsDo:
[:each | each isMethod
 ifTrue: [modelMethodCount :=
 modelMethodCount + 1]].
```

The third enumeration example counts only directly contained methods in the Business Model package. (Any method whose class is also defined by the package is an indirectly contained method because methods are always organized with their class whenever possible.) The message `definitionsDo:` is sent to an instance of `PackageHandle`.

"number of methods that are extensions to other classes"

```smalltalk
extensionMethodCount := 0.
(TeamVInterface current
 packageNamed: 'Business Model')
definitionsDo:
[:each | each isMethod
 ifTrue: [extensionMethodCount :=
 extensionMethodCount + 1]].
```

Class definitions contain a variety of other definitions besides methods: class initializer definitions, instance variable definitions, class variable definitions, etc. Of these definitions, only methods can be organized either with their class or independently in another package. All other definitions contained by a class are always organized with the class. Pool variables are similar to methods - they can be organized with their containing pool or they can be organized independently. The semantic interface has enumerating protocol to distinguish between definitions that can be organized independently, *discrete definitions*, and those that cannot be independent.

Modules respond to the message `allDiscreteDefinitionDo:`, which enumerates over all definitions that could be organized independently (they do not have to be currently organized independently). This message examines directly and indirectly contained definitions.
Clusters have specialized enumeration messages for their components, specifications. The message `specificationsDo:` covers all directly contained specifications, while `allSpecificationsDo:` covers directly and indirectly contained specifications.

The implementation of the module enumerating message `modulesDo:` in `ClusterHandle` enumerates over resolved modules. Let’s look at an example that uses `modulesDo:` and compare it to a variation that uses the enumerating message `specificationsDo:`.

This expression collects all modules in the specified cluster that need to be committed. It does this by enumerating through the modules and collecting the ones that have been modified, sending the message `isModified` to each module.

```smalltalk
clusterHandle := TeamVInterface current
            clusterNamed: 'Model Configuration'.
toCommit := Set new.
clusterHandle modulesDo:
    [:module |
        module isModified ifTrue: [toCommit add: module]].
```

Here is a variation of the expression that collects modified modules. This variation uses the message `specificationsDo:`. Note that each specification may or may not be resolved to a module, depending on its optional and conditional inclusion flags. This variation of the expression is more awkward than the `modulesDo:` variation because of the need to query the resolved module, which may not exist.

```smalltalk
clusterHandle := TeamVInterface current
            clusterNamed: 'Model Configuration'.
toCommit := Set new.
clusterHandle specificationsDo:
    [:specification |
        module := specification resolvedModule.
        module == nil
        ifFalse: [module isModified
            ifTrue: [toCommit add: module]].
```

Queries that involve attributes of specifications are better suited to enumeration with `specificationsDo:`. For example, the following expression identifies references to modules stored as branch revisions. It uses the revision number data stored in specifications, converts it to a revision number and determines whether it is on the main branch. The resolved modules are unnecessary to determine whether branch revisions are involved.
clusterHandle := TeamVInterface current
clusterNamed: 'Model Configuration'.
branches := Set new.
clusterHandle specificationsDo:
    [:specification |
        (RepositoryRevisionNumber fromString:
          specification revisionNumber) isMainBranch ifFalse: 
          [branches add: specification]].

Modules and definitions can be queried for their container object. Definitions respond to the message package, specifications respond to the message cluster, and all components respond to the message subsystem.

There is also an enumeration method, applicable to all modules, for traversing the containers for a module. The message containerSpecificationsDo: is used in the following example to determine if a module is referenced by an optional specification.

moduleHandle := TeamVInterface current
moduleNamed: 'Target'.
moduleHandle containerSpecificationsDo:
    [:spec | spec isOptional ifTrue: [^true]].

This enumeration message can also be used to collect the indirect containers for a module. The following expression collects the clusters containing specifications that resolve to a particular module.

moduleHandle := TeamVInterface current
moduleNamed: 'Target'.
clusters := Set new.
moduleHandle containerSpecificationsDo:
    [:spec | clusters add: spec cluster].

Editing Cluster Contents

New specifications can be added to clusters, and existing specifications can be removed. The message addSpecificationNamed:type:repositoryName:
revisionNumber:string conditionalFlags:optional: is sent to an instance of ClusterHandle to create a new specification. The attributes of a specification are either strings or booleans, as reflected in the arguments to the creation message.
cluster := TeamVInterface current clusterNamed: 'UI Components'
class addSpecificationNamed: 'Enhanced Text Control'
type: 'package'
repositoryName: 'Common UI'
revisionNumber: '1.34'
conditionalFlags: (Array with: 'Windows')
onoptional: false

Existing specifications that are optional are removed in the following example. The message removeSpecification: is sent to an instance of ClusterHandle.

cluster := ToolInterface current
openedClusterNamed: 'UI Components'.
candidates := OrderedCollection new.
class specificationsDo: [:each |
  each isOptional ifTrue: [candidates add: each]].
candidates do: [:each |
  cluster removeSpecification: each].

Definitions

Definitions are the components of packages. All definitions respond to a common set of messages, as specified by the abstract superclass DefinitionHandle, and are inherited by the concrete definition classes we discuss later in this chapter.

All definitions respond to requests for their package using the message package. This expression evaluates to the package containing the class definition for Collection. If the class is not found, then a MessageNotUnderstood error will occur because nil is sent the message package.

(TeamVInterface current
definitionOfClassNamed: #Collection) package

A more robust way to write this code is to check to see if the search is successful before sending the message package.

classDef := TeamVInterface current
definitionOfClassNamed: #Collection.
classDef == nil
ifTrue: [^nil]
ifFalse: [^classDef package]

Definitions respond to a number of classification messages. The messages isDefinition and isModule are useful for distinguishing definitions and modules, and various kinds of definitions can be distinguished with messages like isMethod and isPool.
Modules and definitions maintain a modified state, indicating that they have been edited in some way. The testing message **isModified** can be used to query a definition to see if it has been modified.

```smalltalk
dirty := aDefinition isModified
```

The searching method **findCorrespondingIn:** is used to search the argument, a module, for a definition that defines the same program element as the receiver.

Definitions can be reorganized into different packages with the messages **copyTo:** and **moveTo:**, and deleted with the message **removeFromSystem.** The arguments to the copy and move messages are instances of **PackageHandle.** The **removeFromSystem** message removes a definition from a package. If the package is part of the loaded subsystem, it and its corresponding executable will also be removed from the image.

Definitions have a common editing operation, renaming. The message **rename:** is sent to a definition with a string or symbol specifying its new name as the argument to the message.

### Executables and Values

Definitions that are part of a loaded package have a corresponding executable object or value. Some definitions have specialized executables. Class definitions have instances of **Class** as corresponding executables. Method definitions have instances of **CompiledMethod.** Pool definitions have instances of **PoolDictionary** (or **Dictionary** for backwards compatibility).

Variables have a value rather than a specialized executable object. Pool variable definitions and global variable definitions have a value that is any object. The value of variables is application specific.

All definitions respond to the message **value** by returning the corresponding executable or value. If the definition is part of an unloaded package, the message will return **nil** because there is no corresponding executable. The following statements set the value of **MyGlobal** and compare the value of a global variable with the initial value. The last statement returns **true** because the value of the definition is the same as the initial value.

```smalltalk
initialValue := 3 + 4.
MyGlobal := initialValue.
globalVariable := TeamVInterface current definitionOfGlobalNamed: #MyGlobal.
^globalVariable value = initialValue
```
The testing message `hasValue` can be used to determine if a definition has a corresponding value or executable. The message `isLoaded` returns the boolean true if the package containing the definition is loaded. These two testing message generally return the same value.

**Source-Based Definitions**

Some definitions are source-based, and inherit from the abstract class `SourceBasedDefinitionHandle`. There are two boolean messages that might be confused for source-based definitions. The message `isSourceBased` is a classification message for querying the kind of definition. The message `hasSource` is used to query, on an individual basis, whether a source-based definition has source. (Definitions that originate in an SLL may not have source, even if they are source-based definitions.)

The subclass `MethodDefinitionHandle`, is a source-based definition representing methods. The source for method definitions is processed to construct the compiled form of methods.

Instances of `PoolVariableDefinitionHandle` and `GlobalVariableDefinitionHandle` have source that is used to set the initial value of variables. The compilation context for pool variable and global variable initializers does not include self. When evaluated, these initializers set the value of the variable to the result of the initializer expression. The message `performInitialization` evaluates the initializer and sets the value of the variable.

Instances of `InitializationExpressionDefinitionHandle`, representing ad hoc initializers, are another type of source-based definition. This source is evaluated during `performInitialization`, but the result of evaluating the initializer is not assigned or used. Ad hoc initializers are compiled in the context of `UndefinedObject`, thus `self` is bound to `nil`.

All the source-based definitions we have discussed so far are discrete definitions, and can be independently organized. The last source-based definition, `ClassInitializationHandle`, is not a discrete definition, and must always be organized with its associated class. The result of evaluating this definition is not assigned or used. The source for class initializers is compiled in the context of the class, so class variables, class instance variables and pool variables are accessible. Frequently, class variables are directly or indirectly initialized with the class initializer. Because class variables are shared variables with subclasses, it can be confusing to initialize them using conventional class initialization.
methods due to inheritance and the potential of reinitializing the variables. Either the class definition or the class initialization definition can respond to the request to set a class to its initial state with the message `performInitialization`.

Source-based definitions can be queried for their source with the message `sourceString`. The source for these definitions is set with the message `sourceString:notifying`. Setting the source may involve recompilation, but will not cause a definition to be reset to its initial state.

Source-Based Searching: The Grep Example

Let’s try out these concepts by creating a tool that searches for occurrences of a string in any definition, similar to the UNIX GREP utility. This example uses a data structure that is specialized for definitions, `DefinitionGroup`.

```
"Find any loaded definitions whose source contains the string <aPatternString>, and place them in a definition group."

aPatternString := 'patternString'.
pattern := Pattern new: aPatternString.
matches := DefinitionGroup new.
TeamVInterface current allDefinitionsDo: [:definition |
    (definition isSourceBased and: [definition hasSource])
    ifTrue: [(pattern match: definition sourceString
               index: 1) ~= nil
             ifTrue: [matches addDefinition: definition]]].

"To browse the definition group, evaluate:"
ToolInterface current
    browseDefinitions: matches
    highlight: aPatternString
    labeled: 'Definitions matching: ', aPatternString.
```

This example is meant to be used as a template. To use it, customize the value of the variable `aPatternString`. The variable `pattern` contains an instance of `Pattern`, created from the pattern string. An instance of `DefinitionGroup` is assigned to the variable `matches`, which will be used to record the definitions that match the pattern string.

The message `allDefinitionsDo:` is sent to the loaded subsystem, which enumerates through all definitions in all packages.

Only definitions that have source code associated with them should be searched for an occurrence of the pattern. Thus, for each definition, we must first determine whether it is source-based and whether it currently contains source code. The messages `isSourceBased` and `hasSource`, return `true` or `false`. When both
conditions are true, then the source code for the definition is
retrieved by sending the message sourceString. The resulting
source string is then used as the argument to the message
match:index. If a match is found, the definition is added to the
definition group matches.

The last statement opens a browser on the definitions stored in
matches, using ToolInterface.

Editing Source

The source of source-based definitions can be edited with the
message sourceString:notifying: The first argument is the new
source string, and the second argument is an object that handles
errors and warnings in tools. If the second argument is nil, it
indicates that the editing operation is not interactive.

The following statements edit the initialization source for a global
variable. The return value of the source setting method is checked
for success.

```smalltalk
global := TeamVInterface current
definitionOfGlobalVariableNamed: #CurrentModel.
result := global
  sourceString: 'ModelingMode == #Business
      ifTrue: [BusinessModel new]
      ifFalse: [EmptyModel new]
  notifying: nil.
result == nil ifTrue: [self error: 'global editing failed']
```

Version History

Definitions that contain source can also be queried for their
history. This history is the local history that is specific to a
development environment. These definitions respond to the
message version which provides access to the definition history.
The class DefinitionVersionHandle maintains references to
previous versions of these definitions. Instances of
DefinitionVersionHandle operate like a linked list, each one
maintaining a link to a previous version. The message versions to
a definition returns a collection of instances of
DefinitionVersionHandle.

The following statements access the previous version for a
method.

```smalltalk
current := methodDefinition version.
preceding := current previousVersion.
preceding == nil
  ifTrue: [self error: 'No previous versions']
```
If the request for a previous version returns \texttt{nil}, then there are no preceding versions. The messages \texttt{version} and \texttt{previousVersion} return an instance of \texttt{DefinitionVersionHandle} or \texttt{nil}.

\texttt{InitializationExpressionHandle} represents ad hoc initializers. In addition to the standard testing and classification methods, ad hoc initializers are source based and versioned. The editing operation supported by this class is the standard editing message, \texttt{sourceString: notifying:}.

### Variable Definitions

Global variables and pool variables are very similar in functionality, though different in scope and organization.

#### Pool Searching and Enumeration

Pool definitions also respond to \texttt{definitionsDo:}, \texttt{allDefinitionsDo:}, \texttt{looseDefinitionsDo:} and \texttt{discreteDefinitionsDo:}. Pool variable definitions, which are contained by pool definitions, are discrete definitions. Global variables have no component definitions, but respond to searching messages for completeness.

#### Constant and Variable Designation

Pool variables and global variables can be designated as constant or variable. The messages \texttt{beConstant} and \texttt{beVariable} are used to change the treatment of the variable in the system. If the variable is a constant, then it cannot be the target of an assignment statement. Its value can only be set via initialization, by sending the message \texttt{performInitialization}. If the variable is designated a variable, its use is not restricted. The testing message \texttt{isConstant} is used to determine whether a global variable definition should be regarded as a constant or a variable.

#### Creating Variables

\texttt{PoolVariableDefinitionHandle} has similar protocol to \texttt{GlobalVariableDefinitionHandle}. However, instance creation for global variables and pool variables is different. To create a new global variable definition, send the message \texttt{addGlobalVariableNamed:} to an instance of \texttt{PackageHandle}. To create a new pool variable send the message \texttt{addPoolVariableNamed:} to an instance of \texttt{PoolDefinitionHandle}, or use the message \texttt{addPoolVariableNamed:inPoolNamed:} to a package. New pools are created with the message \texttt{addPoolNamed:} to a package.
Creating a Pool

In this series of statements, we create a pool and add variables to it. These pool variables can be directly referenced in methods of classes that include the pool as a pool usage.

```smalltalk
editingPackage := TeamVInterface current packageNamed: 'Status'.
poolDefinition := editingPackage addPoolNamed: #CommonStatusIndicators.
poolDefinition == nil ifTrue:[^self error: 'Pool creation error'].
poolDefinition addPoolVariableNamed: 'Stable'.
poolDefinition addPoolVariableNamed: 'Experimental'.
poolDefinition addPoolVariableNamed: 'Unknown'.
```

There is a problem with the example above; it creates a pool and populates it with pool variables, but does not assign any values to the pool variables. To complete the job of creating a pool that is ready for use, initialization for each pool variable needs to be defined and the variables need to be initialized. Here is a rewritten form of the example:

```smalltalk
editingPackage := TeamVInterface current packageNamed: 'Status'.
poolDefinition := editingPackage addPoolNamed: #CommonStatusIndicators.
poolDefinition == nil ifTrue:[^self error: 'Pool creation error'].
#(
  #('Stable' '2')
  #('Experimental' '1')
  #('Unknown' '0'))
do: [:array |
  variableName := array at: 1.
  source := array at: 2.
  variableDefinition := poolDefinition addPoolVariableNamed: variableName.
  variableDefinition == nil ifTrue:[^self error: 'variable definition failed'].
  result := variableDefinition sourceString: source notifying: nil.
  result == nil ifTrue:[^self error: 'illegal source'].
  variableDefinition performInitialization]
```

In this variation, we use a literal array to contain the name and source of each pool variable. The initial value of each variable is set with the message `performInitialization`. After each step,
creating the variable definition and setting the source, we check for errors by checking for a nil return value. An alternative to checking for return values is to handle exceptions. Here is an exception handling variation which wraps the entire set of statements in an exception handler for the general exception TeamVError:

```smalltalk
[editingPackage := TeamVInterface current packageNamed: 'Status'.
 poolDefinition := editingPackage addPoolNamed: #CommonStatusIndicators.
 #('Stable''2')
 #'(Experimental' '1')
 #:((Unknown''0'))
 do: [:array |
   variableName := array at: 1.
   source := array at: 2.
   variableDefinition := poolDefinition addPoolVariableNamed: variableName.
   variableDefinition sourceString: source notifying: nil.
   variableDefinition performInitialization]
on: TeamVError do: [:exception |
   MessageBox message: 'Pool creation error: ', exception description]
```

### Classes and Methods

Classes and methods are related. A loaded method definition cannot exist unless a corresponding loaded class definition exists. This relationship is reflected in the container relationship of definitions. If a method is organized into the same package as its class definition, then the method will be directly contained by the class definition. However, methods can be organized as loose method definitions. In this case the class definition is not in the same package, and the method definition is directly contained by the package.

#### Classes

A class definition is a top level class definition plus auxiliary definitions: instance variable definitions, class variable definitions, class instance variable definitions, pool usage definitions, method definitions, and a single class initialization definition. All of the auxiliary definitions for a class, except method definitions, are always contained by the class definition and are accessed through the class definition. Instances of ClassDefinitionHandle respond
to requests for these definitions. For example, the message
\texttt{instanceVariables} returns a collection of instance variable
definitions.

\begin{verbatim}
(TeamVInterface current definitionOfClassNamed: #Rectangle) instanceVariables
\end{verbatim}

Loose method definitions are contained by the package instead of by the class definition, which is in a different package. Class definitions provide some extra protocol for accessing loose definitions related to the class.

All loose definitions related to the class but organized into other packages are accessed with the message \texttt{looseDefinitions}. It returns a collection of definitions from other packages. This expression returns a collection of loose definitions for the class \texttt{Number}.

\begin{verbatim}
(TeamVInterface current definitionOfClassNamed: #Number) looseDefinitions
\end{verbatim}

Class definitions also respond to requests for related definitions, regardless of their organization into packages. You can query instances of \texttt{ClassDefinitionHandle} for its superclass and subclass definitions. This example requests both and records them in \texttt{relatedDefinitions}:

\begin{verbatim}
relatedDefinitions := Set new.
classDefinition := TeamVInterface current
definitionOfClassNamed: #Number.
relatedDefinitions add: classDefinition
superclassDefinition.
relatedDefinitions addAll: classDefinition
subclassDefinitions.
\end{verbatim}

\textbf{Searching through Classes}

Standard searching method are also supported by classes, making classes the smallest scope of the four levels that respond to these searching methods. You can search for all kinds of definitions related to classes: methods, instance variables, class variables, etc.

\begin{verbatim}
classDefinition := TeamVInterface current
definitionOfClassNamed: #Point.
xinstVar := classDefinition
definitionOfInstanceVariableNamed: 'x'.
\end{verbatim}

You can send these searching messages to subsystems, clusters, packages and classes.
sendersOf:
  implementorsOf:
  referencesTo:

Classes also support two specialized searching methods for instance variables and class instance variables. These methods search the receiver, a class definition, for assignments and accesses to the specified variables. The standard message referencesTo: returns any kind of reference. The specialized message assignmentsTo: returns methods or initializers that assign to that variable. The specialized message accessesTo: return non-assignment references.

In this example, we search the class Point for assignments to the instance variable x. Then, using the class ToolInterface, we open a browser on all the methods that perform the assignment.

```smalltalk
classDefinition := TeamVInterface current
definitionOfClassNamed: #Point.
xDefinition := classDefinition
definitionOfInstanceVariableNamed: 'x'.
assignments := classDefinition assignmentsTo: xDefinition.
ToolInterface current
  browseMethods: assignments
  highlight: 'x'
  labeled: 'Assignments to x in Point'
ifNone: 'No assignments to x'
```

Enumerating over Class Components

Instance of ClassDefinitionHandle contain definitions, and therefore respond to definition enumeration requests just as modules do. The messages definitionsDo:, discreteDefinitionsDo: and allDefinitionsDo: are the standard enumeration methods we have discussed so far.

The following table summarize the scoping of the definition enumerating messages. The messages to PackageHandle differentiate between directly and indirectly contained definitions. They do not provide shortcuts for accessing related definitions in other package. This table tells whether an enumerating message would include in its iteration each of the following examples:

- **method in class**, a discrete definition that is directly contained by a class definition.
- **loose method in package**, a discrete definition that is directly contained by the package that is the receiver of the enumerating message
- *loose method in other package*, a discrete definition that is directly contained by a package other than the receiver.

<table>
<thead>
<tr>
<th>Receiver</th>
<th>Selector</th>
<th>Scope:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>method in class</td>
</tr>
<tr>
<td>PackageHandle</td>
<td>definitionsDo:</td>
<td>no</td>
</tr>
<tr>
<td>PackageHandle</td>
<td>allDefinitionsDo:</td>
<td>yes</td>
</tr>
<tr>
<td>PackageHandle</td>
<td>allDiscreteDefinitionsDo:</td>
<td>yes</td>
</tr>
</tbody>
</table>

The enumerating messages for **ClassDefinitionHandle** differentiate between related definitions that are organized into the same package as the class and those that are organized into other packages. Classes also have specialized enumerating methods such as `instanceMethodsDo`, `classMethodsDo`, `allInstanceMethodsDo`, and `allClassMethodsDo`: Like the module and subsystem level methods, the enumerating methods prefixed by all imply enumeration over an extended set of definitions. However, this set is different for classes. For instances of **ClassDefinitionHandle**, the all prefix ignores the organization of definitions into packages.

Methods are the only part of a class definition that are discrete definitions, so the method enumerating messages `instanceMethodsDo` and `classMethodsDo` together iterate over the same set of definitions that `discreteDefinitionsDo` iterates over. The following table summarizes the enumerating messages for `ClassDefinitionHandle`. It tells whether an enumerating message would include in its iteration each of the following examples:

- *method in same package as class*, a discrete definition that is directly contained by a class definition
- *method in other package*, a loose method
- *instance variable definition (same package as class)*, a definition contained by a class definition, and not a discrete definition.
Suppose your system contains a loaded package containing the coordinate system independent methods for `Number`, such as `isAbove`, `isBelow`, `isLeftOf`, `isRightOf`, `above`, `below`, `left`, `right`. Your system also contains a second loaded package containing the class `Number` and the arithmetic methods `+`, `-`, `/`, `//`, `\`, and `*`.

Let’s examine the effect of the various enumeration messages. First we locate the instance of `ClassDefinitionHandle` for the class `Number`.

```smalltalk
toolkitDelegate defineClass: #Number to: #Number on: #Number.
```

The enumeration message `definitionsDo:` covers the definitions that are in the same package as the class definition. In this case, the coordinate system methods for number are not printed in the `Transcript` because they are in a different package. `Number` has a class variable called `SinValues`. In addition to the arithmetic methods, this variable is also printed in the `Transcript` because the enumeration covers all parts of the class definition.

```smalltalk
numberClass := TeamVInterface current definitionOfClassNamed: #Number.

numberClass definitionsDo: [:each |
    Transcript cr; show: each printString.
]
```
The message `allDefinitionsDo:` covers definitions regardless of their organization into packages. In this case, coordinate system and arithmetic method selectors are in the `allSelectors` collection. The enumeration includes the `Number` class variable `SinValues`, but this expression tests each definition to see if it is a method, using the selector `isMethod`.

```smalltalk
allSelectors := Set new.
numberClass allDefinitionsDo: [:each | each isMethod ifTrue: [allSelectors add: each selector]].
```

Discrete definitions are those that can stand alone. Method definitions are discrete definitions because they can be organized into packages separate from their class definition. This expression enumerates over discrete definitions, using `discreteDefinitionsDo:`, reorganizing methods. Methods from the package that contains the class are covered, but not other parts of the class definition, such as instance and class variable definitions.

```smalltalk
arithmeticPackage := TeamVInterface current packageNamed: 'Arithmetic Methods'.
numberClass discreteDefinitionsDo: [:each | each moveTo: arithmeticPackage].
```

The message `looseDefinitionsDo:` enumerates over definitions contained in other packages. In this case, the coordinate system methods but not the arithmetic methods are covered. The loose definitions examined in this enumeration must be discrete definitions.

```smalltalk
outsiders := OrderedCollection new.
numberClass looseDefinitionsDo: [:each | each isMethod ifTrue: [outsiders add: each selector]].
```

### Editing

`ClassDefinitionHandle` supports a variety of editing messages. Redefinition may include recompilation of methods in this class and in subclasses.

There are a series of messages to add new variables to a class. In this example we add a new instance variable to the class `Sphere`.

```smalltalk
classDefinition := TeamVInterface current definitionOfClassNamed: #Sphere.
classDefinition addInstanceVariableNamed: 'volumeCache'.
```
You can also change the class hierarchy by sending the message
`changeSuperclassNameTo` with the name of an existing class or `nil` if the receiver should be the root of a class hierarchy. In these statements we change the superclass of the class named `AirplaneSeat`.

```
classDefinition := TeamVInterface current
definitionOfClassNamed: #AirplaneSeat.
classDefinition changeSuperclassNameTo: #TransportationSeat.
```

All attributes of class definition can be modified with this multiple-keyword message.

- `superclassName: aSymbol` - Where `aSymbol` is one of the values: `#none`, `#byte`, `#object`.
- `indexableType: aSymbol` - Where `aSymbol` is one of the values: `#none`, `#byte`, `#object`.
- `instanceVariableNames: aCollection` - Where `aCollection` is a collection of strings containing variable names.
- `classVariableNames: aCollection>` - Where `aCollection` is a collection of strings containing variable names.
- `poolNames: aCollection` - Where `aCollection` is a collection of strings containing pool names.
- `classInstanceVariableNames: aCollection` - Where `aCollection` is a collection of strings containing variable names.

Here is an example of this message in which the class `AirplaneSeat` is redefined. We do not change the indexable type or the instance variables, but reset the rest of the parts of the class definition.

```
classDefinition := TeamVInterface current
definitionOfClassNamed: #AirplaneSeat.
classDefinition superclassName: #TransportationSeat
indexableType: classDefinition indexableType
instanceVariableNames: classDefinition instanceVariableNames
classVariableNames: (Array with: 'DefaultColorScheme')
poolNames: (Array with: 'ColorConstants')
classInstanceVariableNames: Array new
```
Creating Classes and Methods

Most program elements are created by sending messages to a package. Class definitions are created by sending the following message to an instance of PackageHandle. This message is very similar to the one used to change existing class definitions.

```smalltalk
addClassNamed: aSymbol
superclassName: aSymbol
indexableType: aSymbol
instanceVariableNames: aCollection
classVariableNames: aCollection
poolNames: aCollection
classInstanceVariableNames: aCollection
```

Where `aSymbol` is one of the values: `#none`, `#byte`, `#object`.

```smalltalk
instanceVariableNames: aCollection
```

Where `aCollection` is a collection of strings containing variable names.

```smalltalk
classVariableNames: aCollection
```

Where `aCollection` is a collection of strings containing variable names.

```smalltalk
poolNames: aCollection
```

Where `aCollection` is a collection of strings containing pool names.

```smalltalk
classInstanceVariableNames: aCollection
```

Where `aCollection` is a collection of strings containing variable names.

The indexable type is specified with a symbol. The symbol `#byte` is used to create a class with byte-indexable fields, in which each indexable field contains binary data. The symbol `#object` is used to create a class with object-indexable fields, in which each indexable element is a reference to another object. The symbol `#none` is used to create a class that adds no indexable fields to the ones defined by the superclass. The class can inherit either byte- or object-indexable fields with the `#none` argument.

Methods

Methods are discrete definitions represented by instances of MethodDefinitionHandle. They are source-based and versioned. As with other source-based definitions, the source is accessed with the message `sourceString`, and set with the message `sourceString: notifying:`. Methods are unusual in that the source string implicitly defines the name of the method, its selector. The return value from editing the source of a method is a method definition or `nil`. `nil` indicates that the redefinition of the source failed. The return of a method definition indicates that the redefinition was successful, but the returned method definition
may not be the receiver. If the source specified a selector different from the receiver’s selector, then a new method definition is returned.

This example sets the source of a method definition, and checks to see if the method was correctly redefined.

```smalltalk
result := aMethodDefinition sourceString: newSource
           notifying: nil.
result == nil
  ifTrue: [Transcript cr; show: 'Redefinition failed'].
result == aMethodDefinition
  ifTrue: [Transcript cr; show: 'Redefinition succeeded']
  ifFalse: [Transcript cr; show: 'New method defined']
```

There are messages to support accessing the attributes of a method definition. The message `selector` returns a symbol representing the message selector for the definition. The message `className` also returns a symbol, this one representing the name of the class the method is associated with. The message `messages` return a collection of messages sent by the method. The message `isMeta` returns `true` if the definition is a class method definition and `false` if it is an instance method definition.

Methods are created with the message `addMethodInClassNamed:meta:sourceString:notifying:`. The last two keywords, `sourceString:` and `notifying:`, are also found in the message for editing the source of definitions. Here is an example of creating a new method definition.

```smalltalk
methodDefinition := aPackageHandle
                  addMethodInClassNamed: #String
                  meta: false
                  sourceString: 'asRevisionNumber
                   "Return an instance of RepositoryRevisionNumber, based on the receiver."

                   ^RepositoryRevisionNumber fromString: self
                   notifying: nil
```

Annotations

All components of the semantic subsystem have annotations, which are used for associated auxiliary information. Definitions, modules, specifications and subsystems all have annotations. Annotations are used by the system for comments, categories and author information.
Annotations consist of a label and a value that must be a string. Users can add their own annotations or modify existing annotations, though annotations that begin with a period character are read-only from the tool set.

The class **AnnotationManager** contains predefined labels in the form of class messages. For example, the message **categoriesLabel** returns a string that is the label of the annotation for method categories. The message **commentLabel** returns a string that is the label of the annotation for comments.

All semantic objects respond to a request for a the value of a particular annotation. This example uses the message **annotationNamed:ifAbsent:** where the second argument is a block that is evaluated if the annotation is not present. If there is no comment, the offending definition is added to the **needsWork** collection.

```smalltalk
aDefinition
  annotationNamed: AnnotationManager
categoriesLabel
  ifAbsent: [needsWork add: aDefinition]
```

Values are assigned to annotations with the message **annotationNamed:put:** which creates an annotation if it does not exist. The message **hasAnnotationNamed:** returns **true** if the receiver has an annotation with the specified label. Annotations can be removed with the message **removeAnnotationNamed:**.

The following lengthy example collects data about categories and performs category editing operations. The section of code commented “Setup” does two important things. First, it determines the scope of the operations. This scope could be the entire image, but most likely there is a subset. Secondly, it caches the annotation label.

The next section of code collects the categories for all the implementors of the specified selector, and if there is more than one category, opens an inspector on the **categories** collection. It uses the message **annotationNamed:ifAbsent:**.

The third section of code resets the value of the category annotation if necessary. It doesn't unnecessarily set the category annotation because resetting annotations marks the definitions as modified. This section uses the messages **annotationNamed:ifAbsent:** and **annotationNamed:put:**.
"Set up"

label := AnnotationManager categoriesLabel.
"Customize the scope by specifying the name of a cluster"
scope := TeamVInterface current
  clusterNamed: 'Modeling Cluster'.
"Customize the selector by specifying the selector for a message."
selector := #printString.

"Current category labels for implementations of a message."
defGroup := scope implementorsOf: selector.
categories := Set new.
defGroup definitionsDo:
  [:each |
    categories add: (each annotationNamed: label
      ifAbsent: ['no category']).
  ]
categories size > 1 ifTrue: [categories inspect]

"Set new category for all implementations within the scope."
"Customize the new category"
newCategory := 'printing'.
defGroup := scope implementorsOf: selector.
defGroup definitionsDo: [:each |
  (each annotationNamed: label ifAbsent: ['']) =
    newCategory
    ifFalse: [each annotationNamed: label
      put: newCategory]].

Repository Interface

The repository interface provides operations to create and access new repositories, entities, and revisions.

A repository is a shared disk directory that provides permanent storage for entities and their revisions. A repository contains a collection of entities, and an entity contains a collection of revisions. An entity is usually a module (a package or cluster), although you can create new entity types. A revision contains a single, specific version of user data for an entity. Repositories and entities are sometimes referred to as containers since they contain revisions of entities.

Repositories, packages, clusters, and revisions are collectively known as repository components.
The figure below shows the relationship of repository components:

*Figure 6-2: Relationships of Repositories, Entities, and Revisions.*

You create new repositories, entities, and revisions using the repository interface.

You need to distinguish between the physical data on the disk and the Smalltalk objects used to represent and access the physical data in the Repository interface. The repository interface uses instances of the classes `Repository`, `RepositoryEntity`, and `EntityRevision` to represent repositories, entities, and revisions on the disk. These instances describe data which may or may not actually exist on the disk.

Before you can operate on one of these components, you need to establish a connection in order to access the physical data represented by one of these instances. To establish a connection to a component, send it the message `open`. If the physical data does not exist, an exception occurs. You can test for whether the physical data exists before trying to open it with the message `exists`. When you are through accessing a component, send it the `close` message to terminate the connection and release the system resources being used.

You access the physical data in a revision with instances of `ReadStream` and `WriteStream`. When you initially create a revision, it is in a *proposed* state and can be read and written. When you are through modifying a revision, you *commit* it. This
makes the revision read-only so it cannot be modified further. Should you decide not to commit a proposed revision, you can \textit{abort} the revision.

The repository interface consists of four principle classes, \textit{RepositoryServices, Repository, RepositoryEntity,} and \textit{EntityRevision}.

\textbf{RepositoryServices} is the class that provides access to the basic repository functions, such as creating and accessing repositories.

Instances of class \textbf{RepositoryRevisionNumber} represent revision numbers.

\section*{Identifying Repositories, Entities, and Revisions}

Repositories, entities, and revisions are identified using a \textit{repository system path}. This construct identifies the repository components that contain a specific entity or revision. The parts of a repository system path are

- Repository name
- Entity name
- Entity type
- Entity revision number string

A repository is identified by its name—a string that can be any length and can contain spaces and punctuation in addition to alphanumeric characters.

An entity is identified within a repository by entity name and type. The entity name is a string of any length that can contain spaces and punctuation in addition to alphanumeric characters. The entity type is also a string, either ‘package’, ‘cluster’ or a string identifying an entity type defined outside of Visual Smalltalk Enterprise. The name and type must uniquely identify an entity within a repository. Entities having different types can share the same name.

Within an entity, an entity revision is identified with a revision number string. For more information on the format of revision number strings, see \textit{Revision Control} in chapter 1.

When using the repository interface, you identify a specific entity or revision in the repository with an instance of class \textbf{RepositorySystemPath}. 
**NOTE:** The repository system path lets you symbolically describe the location of a repository component, but there may not be any physical data at that location. A repository system path is similar to a file system path in this respect—you can create paths that don’t correspond with physical data.

Once you have a repository system path, you can *resolve* it to create an instance of **Repository**, **RepositoryEntity**, or **Revision**.

As with the repository system path, having an instance of one of these classes does not mean that there is physical data for it. In fact, in order to create a new component, you create a repository system path naming the non-existent component, resolve the path to create an instance of either **Repository**, **RepositoryEntity**, or **Revision** representing the new component, and then send the instance the *create* message to actually create the physical data.

**Creating Instances of RepositorySystemPath**

There are several instance creation messages implemented by class **RepositorySystemPath**. The instance creation message you use depends on the kind of component being identified. The following table shows which instance creation message to use to identify specific components

<table>
<thead>
<tr>
<th>Message</th>
<th>Creates a path that identifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>repositoryName:</td>
<td>a repository</td>
</tr>
<tr>
<td>repositoryName: repositoryEntityName: repositoryEntityType:</td>
<td>an entity (usually a package or cluster)</td>
</tr>
<tr>
<td>repositoryName: repositoryEntityName: repositoryEntityType: revisionString:</td>
<td>an entity revision</td>
</tr>
</tbody>
</table>

For example, the following expression creates an instance of **RepositorySystemPath** representing revision 1.0 of the Graphics Subsystem package in the Alpha Project repository:
path := RepositorySystemPath
repositoryName: 'Alpha Project'
repositoryEntityName: 'Graphics Subsystem'
repositoryEntityType: 'package'
revisionString: '1.0'.

Obtaining a Reference to a Repository, Entity, or Revision

Once you have an instance of RepositorySystemPath that identifies a repository component (a specific repository, entity, or revision), you can create an instance of Repository, RepositoryEntity, or EntityRevision that identifies the same component by sending the message resolve to the instance of RepositorySystemPath.

The class of the object returned by resolve is determined by which fields have been specified in the repository system path.

For example, if only the repository name is specified in the repository system path, resolve returns an instance of Repository. If the repository name, entity name, and entity type are specified, resolve returns an instance of RepositoryEntity. If all fields are filled in, an instance of EntityRevision is returned. If the path can’t be resolved to a single entity, an exception is raised.

The table below summarizes the behavior of resolve when various fields are supplied with the repository system path is created.

Table 6-3: Resolving a RepositorySystemPath.

<table>
<thead>
<tr>
<th>Repository</th>
<th>Name</th>
<th>Type</th>
<th>Revision</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>Returns the specified Repository.</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>Returns the specified RepositoryEntity</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Returns the specified EntityRevision.</td>
</tr>
</tbody>
</table>

When you resolve a repository system path, the following steps are performed:

1. The path is checked to make sure it contains the proper combination of components as indicated in the table above.
2. The kind of component (repository, entity, or revision) the path identifies is determined.
3. All components in the path preceding the last component are checked for existence. The final component is not required to exist. An exception is raised if any components except the last can’t be found. This permits the creation of non-existent
entities since you need a repository system path to identify the new entity before you can create it.

4. An instance of the corresponding component type (Repository, RepositoryEntity, or EntityRevision) is returned.

**NOTE:** Resolving the path to a component does not check whether the physical data for the component actually exists. In order to confirm the existence of the physical, send `exists` to the result of `resolve`.

### Matching Multiple Repository Components

You can also perform a “wild-card” match on a repository system path that does not have all fields specified. In this case, the path resolves to a collection of all components (entities and revisions) that match the specified part of the pattern. If no components are found that match the repository system path, an empty collection is returned.

You can “wild card” fields in a repository system path by setting them either to nil or an empty string. The repository name must always be specified. The entity name, type, and revision number fields can be wild-carded.

You perform a wild-card resolution by sending the `resolveMatching` message to a path. The example below prints the names of all packages in the “Alpha Project” repository in the system transcript:

```smalltalk
repository := repositoryServices repositoryFromPath: 'F:\ALPHA'.
path := RepositorySystemPath
    repositoryName: 'Alpha Project'
    repositoryEntityName: ''
    repositoryEntityType: 'package'
    revisionString: ''.
allPackages := path resolveMatching.
Transcript cr; show: 'Alpha Project Packages:'.
allPackages do: [:package |
    Transcript cr; tab; show: package name].
repository close.
```
The following table shows the result of wild-carding various fields in the repository system path. The first four columns indicate which fields of the repository system path are wild-carded. A nil in a column indicates that the field is wild-carded. A check mark in a column indicates that the field has been specified.

Table 6-4: Repository, RepositoryEntity, and EntityRevision Operations.

<table>
<thead>
<tr>
<th>Repository</th>
<th>Name</th>
<th>Type</th>
<th>Revision</th>
<th>#resolveMatching result</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>Error, a repository must always be specified.</td>
</tr>
<tr>
<td>✓</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>All entities in the specified repository.</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>nil</td>
<td>nil</td>
<td>All entities having the specified name in the specified repository.</td>
</tr>
<tr>
<td>✓</td>
<td>nil</td>
<td>✓</td>
<td>nil</td>
<td>All entities of the specified type in the specified repository.</td>
</tr>
<tr>
<td>✓</td>
<td>nil</td>
<td>nil</td>
<td>✓</td>
<td>All revisions of all entities that match the revision number.</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>nil</td>
<td>✓</td>
<td>All revisions that match the given name and revision.</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>nil</td>
<td>The entity that matches the name and type specified.</td>
</tr>
<tr>
<td>✓</td>
<td>nil</td>
<td>✓</td>
<td>✓</td>
<td>All revisions of all entities having the specified type and revision number.</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>The specified revision.</td>
</tr>
</tbody>
</table>

Unlike resolve, resolveMatching only returns entities and revisions that match the path and whose physical data exists. Therefore, it is unnecessary to send exists to objects in the collection returned by resolveMatching.

Common Operations on Repositories, Entities, and Revisions

The components that represent repositories, entities, and entity revisions share some common concepts and operations. Each repository component has a path similar to a repository system path that contains the information needed to locate the physical data and a connection that provides a means of accessing the physical data.

Each component has three properties—a comment, a creator, and a creation time stamp. You can specify the comment string for repositories and entities at any time. The comment string for a
revision must be specified before the revision is committed. The creator and creation time stamp are set automatically by the 
Repository Interface when the component is created.

The following operations are common to repositories, entities, and 
entity revisions (instances of Repository, RepositoryEntity, and 
EntityRevision):

<table>
<thead>
<tr>
<th>Selector</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>exists</td>
<td>Answers true if a connection can be established to the physical data.</td>
</tr>
<tr>
<td>creation-TimeStamp</td>
<td>Answers a two-element array containing the date and time the component was created.</td>
</tr>
<tr>
<td>creator</td>
<td>Answers a string naming the person who created the component</td>
</tr>
<tr>
<td>comment</td>
<td>Answers the comment for the component.</td>
</tr>
<tr>
<td>comment:</td>
<td>Sets the comment for the component. Comments for revisions must be set before the revision is committed.</td>
</tr>
<tr>
<td>create</td>
<td>Creates the physical data represented by the object's path.</td>
</tr>
<tr>
<td>destroy</td>
<td>Removes the component from the object containing it and destroys the physical data for the component.</td>
</tr>
<tr>
<td>open</td>
<td>Establishes a connection to the physical data.</td>
</tr>
<tr>
<td>close</td>
<td>If open, closes the connection to physical data and releases the system resources used by the connection. Also closes any contained components that are open. Normally sent just to repositories.</td>
</tr>
<tr>
<td>name</td>
<td>Answers a string containing the name of the component.</td>
</tr>
</tbody>
</table>

Common Operations on Repositories and Entities

The repository components that contain other components share some common operations too. These container components are 
repositories, since they contain entities, and entities, since they contain revisions. The operations common to container 
components are:

<table>
<thead>
<tr>
<th>Selector</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>components</td>
<td>Answers a collection of the components contained by the receiver.</td>
</tr>
<tr>
<td>componentsDo:</td>
<td>Iterates through the components contained by the receiver.</td>
</tr>
</tbody>
</table>
Protocol for Accessing Repositories, Entities, and Revisions

You use the **exists**, **open**, and **close** messages for to test for the existence of physical data for repository components and to establish and break connections to the physical data as follows:

- Send **exists** to a component before you send **open** to make sure that the physical data exists. If **exists** returns true, the **open** should succeed.
- Send **open** to the component before you perform other operations on it. **Open** ensures that the connection to the repository is valid and that all cached data is up to date. Although many operations automatically open the component if it is closed, it is good style to explicitly test for the existence of the component and then open it. This lets your application detect and report errors more robustly.
- Send **close** to the component when you are finished with it. **Close** closes any open streams to the component, closes all open components contained by the component, terminates the connection, and releases the system resources used by the connection. Unlike **open**, which only operates on the component specified, the **close** operation is recursive on all the components contained by the component being closed.

While it is good practice to explicitly **open** every component before you access it, it is not necessary to explicitly **close** every component, since closing the repository will close all open components in the repository. So, conventional usage is to **close** the repository when you are through operating on its components. If you are accessing a large number of components in a repository, you can reduce the total amount of system resources used by closing the individual components (entities or revisions) when you are through with them.

Accessing Repository Services

The sole instance of **RepositoryServices** is accessed from the Smalltalk/V Service Registry using the following expression:

```smalltalk
repositoryServices := ServiceRegistry globalRegistry serviceNamed: #RepositoryServices ifNone: [self error: 'Repository services not available']
```

For more information on using the Smalltalk/V service registry, see the *Smalltalk/V Programming Reference*. 
The code examples in the remainder of this section assume that the variable `repositoryServices` was initialized using the expression above.

**NOTE:** Repository services are only available in the Visual Smalltalk Enterprise development environment. If your application can be used in a Visual Smalltalk runtime application, you should use the `ifNone:` block in the expression above to set up your application to operate without a repository.

### Creating a Repository

You use repository services to create a new repository. When creating a repository, you provide

- the repository name,
- the path to the file system directory that contains the repository files,
- and the type of the repository

You use the `createRepositoryNamed:path:type:` message to create a repository as shown in the following example:

```smalltalk
repositoryServices := ServiceRegistry globalRegistry serviceNamed: #RepositoryServices ifNone: [self error: 'Repository services not available'].
newRepository := repositoryServices createRepositoryNamed: 'Alpha Project' path: 'F:\ALPHA' type: 'PVCS'.
```

The repository name is a string that can be any length and can contain spaces and punctuation in addition to alphanumeric characters. If a repository that is referenced by the image already exists with the same name, an exception is raised.

The path indicates either an existing file system directory or a new subdirectory in an existing directory. In the second case, the directory is automatically created. This directory can contain a PVCS or File-Based repository, but not both.

The type indicates the kind of repository—either 'PVCS' for a PVCS-based repository or 'File-Based' for a repository that doesn’t use PVCS. The repository type is set when the repository is created and can not be changed.
Opening and Closing a Repository

A repository must be open in order to perform any operations on its components such as creating new entities or revisions or reading old revisions.

To initially connect to a repository and open it, use the `repositoryFromPath:` message as shown in the following example:

```smalltalk
repository := repositoryServices repositoryFromPath: 'c:\beta'.
```

This expression determines whether a repository exists in the given directory (raising an exception if it doesn’t), opens it, and registers it with repository services. It is not an error to open a repository that is already open. Repository services adds the repository to its collection of open repositories. The expression returns an instance of `Repository` that you can use to perform subsequent operations on the repository.

When you are through working with a repository, you should send it the `close` message. Closing a repository performs the following operations:

- Closes all open components within the repository (including file streams that are open on entity revisions).
- Releases the system resources the repository connection is using.
- Removes the repository from the repository services.

There are two ways of accessing a repository again after it has been closed.

- You can keep a reference to the closed repository and send it the `open` message to reopen it, or
- You can create a new reference with `repositoryFromPath:`. This automatically opens the repository each time.

The first technique is preferred, since it is faster and doesn’t require you to remember the directory path of the repository.

**NOTE:** Before you send `open` to a repository to reopen it, you should always check for the existence of the repository (or establish an exception handler in case the `open` fails), since the physical data might have been moved or deleted by others after you closed it.
The following code illustrates the recommended way to access a repository.

```smalltalk
repositoryServices := ServiceRegistry globalRegistry
                   serviceNamed: #RepositoryServices
                   ifNone: [self error: 'Repository services not available'].

repository := repositoryServices
              repositoryFromPath: 'c:\beta'.

" use repository"

repository close.

" when you need the repository again. "

repository exists
  ifTrue: [repository open]
  ifFalse: ['failure logic'].

" use repository again"

repository close.
```

### Accessing Open Repositories

The repository service provides access to all open repositories. You can iterate through instances of `Repository` representing all the open repositories as shown in the following example:

```smalltalk
repositoryServices repositoriesDo:
  [:repository | ...
```

You can obtain a collection of strings containing the names of all open repositories with:

```smalltalk
names := repositoryServices repositoryNames.
```

and access a specific open repository named “Alpha Project” with

```smalltalk
repository := repositoryServices
              repositoryNamed: 'Alpha Project' ifAbsent: [...].
```

**NOTE:** There is no way to access a repository that is not open. You must use `repositoryFromPath:` to open it before you can access it.

### Accessing Entities in a Repository

There are several ways of accessing the entities (packages and clusters) in an open repository.
Accessing an Entity with a Path

If you know the repository, name, and type of the repository entity you want to access, simply create a repository system path naming that entity and send it `resolve` to create a reference to that entity:

```
| path package |
path := RepositorySystemPath
repositoryName: 'Alpha Project'
repositoryEntityName: 'Graphics Subsystem'
repositoryEntityType: 'package'.
package := path resolve.
```

The successful evaluation of the statements above assigns an instance of `RepositoryEntity` to the `package` temporary variable. Once you have an instance of `RepositoryEntity`, you can send it `exists` to find out if there is physical data representing the entity:

```
package exists
ifTrue: [package open]
ifFalse: [self error: 'Cannot locate package'].
```

Accessing all Entities in a Repository

To access all entities in a repository, you can either use `resolveMatching` as shown earlier, or you can obtain a collection of all components in a repository with the `components` message.

```
allEntities := repository components.
```

If you want to process each of the entities in a repository, use the `componentsDo:` message:

```
repository componentsDo: [
  :repositoryEntity | ...
]
```

Creating a New Entity

You create a new repository entity by creating a path naming the new entity, resolving that path to get an instance of `RepositoryEntity`, and then sending `create` to the instance. The repository must have been created and be open in order to create an entity in it.

The following example shows how to create a new package to contain the Graphics Subsystem package in the Alpha Project repository.

```
repositoryServices := ServiceRegistry globalRegistry
serviceNamed: #RepositoryServices
ifNone: [self error: 'Repository services not available'].
```
repository := repositoryServices
    repositoryFromPath: 'F:\ALPHA'.
path := RepositorySystemPath
    repositoryName: 'Alpha Project'
    repositoryEntityName: 'Graphics Subsystem'
    repositoryEntityTypeName: 'package'.
entity := path resolve.
entity exists
    ifTrue: [self error: 'The entity already exists']
    ifFalse: [entity create].
repository close.

Accessing Revisions of an Entity

If you just want to know what revisions of an entity exist, you can send the message revisionNumbers to an entity, as illustrated in the follow code that prints all the revisions of a specific entity to the system transcript:

repositoryServices := ServiceRegistry globalRegistry
    serviceNamed: #RepositoryServices
    ifNone: [self error: 'Repository services not available'].
repository := repositoryServices
    repositoryFromPath: 'F:\ALPHA'.
path := RepositorySystemPath
    repositoryName: 'Alpha Project'
    repositoryEntityName: 'Graphics Subsystem'
    repositoryEntityTypeName: 'package'.
revisionNumbers := path resolve revisionNumbers.
Transcript cr; show: 'EntityRevisions:'.
revisionNumbers do: [:aString |
    Transcript cr; tab; show: aString].
repository close.

The revisionNumbers message returns an ordered collection of strings containing the revision numbers. The revision numbers are sorted so that the branches of a revision immediately follow the revision. So, if the revision numbers on an entity are 1.0, 1.1, 1.1.0.1, 1.1.0.2, and 1.2, the strings in the collection returned by revisionNumbers are ‘1.0’, ‘1.1’, ‘1.1.0.1’, ‘1.1.0.2’, and ‘1.2’.

Similarly, the revisionNumberList message returns an ordered collection of instances of RepositoryRevisionNumber for the revisions in an entity.

As with all container components, you can access the revisions in an entity with components and componentsDo.
To access a single revision, construct a complete repository system path and resolve it.

**Reading a Revision**

You read the data in an existing revision by opening the revision and requesting aReadStream on its contents. The following example demonstrates the steps you use to read the data in an existing revision.

```smalltalk
repositoryServices := ServiceRegistry globalRegistry serviceNamed: #RepositoryServices ifNone: [self error: 'Repository services not available'].
repository := repositoryServices repositoryFromPath: 'F:\ALPHA'.
path := RepositorySystemPath
    repositoryName: 'Alpha'
    repositoryEntityName: 'Graphics Subsystem'
    repositoryEntityType: 'package'
    revisionString: '1.0'.
revision := path resolve.
revision exists
    ifTrue: [revision open]
    ifFalse: [self error: 'Cannot find the specified entity revision.'].
readStream := revision readStream.
[Transcript cr; show: readStream contents. Transcript cr; show: entityRevision comment]
ensure: [readStream close].
repository close.
```

This example carries out the following steps:

1. Opens the repository so operations can be performed on its components.
2. Creates a repository system path naming the revision to be read.
3. Resolves the path, which creates an instance of EntityRevision that is assigned to the revision variable.
4. Uses exists to check that the revision exists.
5. Establishes a connection to the physical data with open.
6. Obtains an instance of ReadStream that contains the revision data.
7. Displays the contents of the stream in the transcript.
8. Displays the revision comment in the transcript.
Uses ensure: to make sure that the ReadStream is closed, even if an error occurs accessing the revision.

Closes the repository, closing all open components.

Creating a New Revision

You create the physical data for a new revision by sending create to an instance of EntityRevision. You then send the writeStream message to get an instance of WriteStream to use to set the contents of the new revision.

The newly created revision is initially in the proposed state and its contents can be modified. When the contents are finalized, you commit the revision to make it read-only. If you change your mind and want to discard the proposed revision, you abort it.

As long as a revision is in the proposed state, its revision number is reserved and cannot be used by anyone else. If the revision is aborted, the reserved revision number is released for reuse.

A proposed revision is stored as physical data in the repository, but only the person who created it can access it.

A proposed revision should always be either committed or aborted. Otherwise, if you create a new revision and do not commit or abort it, the revision number will be unavailable in the future, even though no revision exists with that number, and the physical data for the proposed revision will remain inaccessible in the repository.

It is good practice to structure the code that creates a revision to guarantee that it is either committed or aborted. The following example uses the ifCurtailed: message to ensure that the revision is aborted if any problem occurs during a commit. For more information on ensured execution, see the Exception Handling chapter in your Smalltalk/V Programming Reference.

The following example demonstrates the creation of a new revision:

```smalltalk
repositoryServices := ServiceRegistry globalRegistry serviceNamed: #RepositoryServices
ifNone: [self error: 'Repository services not available']
```
repository := RepositoryServices
    repositoryFromPath: 'F:\ALPHA'.
path := RepositorySystemPath
    repositoryName: 'Alpha Project'
    repositoryEntityName: 'Graphics Subsystem'
    repositoryEntityType: 'package'
    revisionString: '1.1'.

revision := path resolve.
[revision exists
    ifTrue: [self error:
        'The proposed revision already exists.'
    ]
    ifFalse: [revision create].
writeStream := revision writeStream.
writeStream nextPutAll:
    'This is the data for revision 1.1.'.
writeStream close.
revision comment:
    'This is the comment for revision 1.1.'.
revision commit]
    ifCurtailed: [revision abort].
repository close.

The steps carried out by this example are

1. Opens the repository so operations can be performed on its components.
2. Creates a repository system path naming the revision to be created.
3. Resolves the path, which creates an instance of EntityRevision that is assigned to the revision variable.
4. Uses exists to verify that the revision does not exist.
5. Uses create to create the new revision.
6. Obtains an instance of WriteStream to set the contents of the revision by sending writeStream to the new revision. Closes the stream after the contents are written.
7. Uses comment: to set the comment for the new revision.
8. Sends commit to make the revision read-only.
9. Uses an exception handler (the ifCurtailed: message) to guarantee that the proposed revision is aborted if anything goes wrong while the new revision is being created or committed.
10. Closes the repository, closing all open components and streams.
Locking and Unlocking Entities and Revisions

You lock entities to prevent other users from creating new revisions. You would normally do this if it will take a significant amount of time to prepare a new revision and you don’t want any intervening revisions committed.

You lock an entity by sending the lock message to an instance of EntityRevision. The lock is automatically removed when you commit a new revision, or when you send the unlock message.

**NOTE:** Even though you lock and unlock EntityRevisions, the entire entity is effectively locked.

Relocating a Repository

If the directory path of a repository changes you can specify a new directory. The path of the repository directory might change because the directory was moved or the drive letter of the directory changed because of network drive mapping changes.

For example, if the variable alphaRepository contains a reference to repository ‘Alpha’ at location ‘F:\ALPHA’ and the directory is moved to ‘E:\ALPHA’, you would record this change with the following expression:

```
alphaRepository resetRepositoryLocationTo: 'E:\ALPHA'
```

**NOTE:** You move a repository by copying the directory containing the repository files, and then using the expression above to notify the repository interface of the change. The expression does not physically move the directory.
Visual Smalltalk Enterprise for OS/2 includes optional tools and browsers to support the development of Smalltalk/V applications that interact with objects defined by the OS/2 System Object Model (SOM).

The SOM tools consist primarily of two browsers: the SOM Class Browser and the SOM Typedef Browser. These browsers let you peruse the SOM Interface Repository for classes and types (typedefs) defined externally by SOM.

You can select externally defined SOM classes and typedefs, and then create new packages containing automatically generated Smalltalk classes that serve as proxies for existing SOM classes and typedefs.

Do not confuse the SOM Interface Repository that the SOM tools browse with a Visual Smalltalk Enterprise repository. The SOM Interface Repository is a standard part of SOM, and is not part of Visual Smalltalk Enterprise. The SOM Interface Repository contains class and type definitions that have been registered with SOM.

Installing the SOM Tools

The SOM Tools are supplied in the LIB subdirectory of your Visual Smalltalk Enterprise installation. You must install these tools into Visual Smalltalk Enterprise before you can use them.

You can either install the Smalltalk libraries containing the SOM base and tool support, or you can load the clusters from the repository containing these applications. Installing the Smalltalk libraries is faster and uses less memory than installing the packages. Loading the clusters keeps the organizational and explanatory information with the code if you want to browse it.

Installing the SOM Tools from Smalltalk Libraries

The SOM tools install on top of the base support, providing tools that facilitate SOM interactions. All of these files are contained in the SAMPLE\SOM subdirectory of your Visual Smalltalk Enterprise installation directory.
To install SOM support from Smalltalk libraries:

1. Start Visual Smalltalk Enterprise for OS/2 image.
2. Choose Smalltalk / Browse Services to open the Service Manager.
3. Select “SOM Support” and click Install>>.
4. Select “SOM Tools” and click Install>>.

Loading the SOM Tools from Clusters

Instead of installing the SOM base and tool support from Smalltalk libraries as described in the previous section, you can load a cluster from the repository located in SAMPLE\SOM\SRCREPOS subdirectory of your Visual Smalltalk Enterprise directory.

To load the SOM product cluster:

1. Open the Repository Browser.
   Choose Smalltalk / Open -> Repository Browser.
2. Select Repository / Connect... and find the SRCREPOS directory in the directory dialog.
3. In a package browser, choose Module / Load... to open the Load dialog.
4. In the Load dialog, select SOM Repository from the repositories list.
5. Select SOM Product Cluster from the entities list, and select the most recent (greatest) revision number in the revisions list.
6. Press Load to load the cluster.

You can now browse the SOM product cluster in the Package Browser.

The SOM Browsers

Two browsers are provided with the SOM Tools:

- The SOM Class Browser lets you browse the SOM classes defined in the SOM Interface Repository and automatically create new Smalltalk/V proxy classes to represent the SOM classes you select.
- The SOM Typedef Browser lets you examine the SOM types defined in the SOM Interface repository and automatically
create new Smalltalk/V proxy classes to represent the SOM types you select.

Remember that the SOMIR environment variable defined in your CONFIG.SYS file defines which SOM Interface Repositories you can browse with the SOM tools. If you do not see the definitions you expect to in the SOM browsers, check the setting of the SOMIR variable.

**SOM Class Browser**

You open a new SOM Class Browser by evaluating the expression

```
SOMClassBrowser open
```

Figure 7-1 shows the SOM Class Browser. The class list at the upper left shows all the SOM classes defined in your SOM repositories.

An asterisk next to a SOM class name indicates that a Smalltalk proxy class has been generated for that SOM class and is currently loaded in your image.

The top middle pane contains the method list, showing the methods implemented by the selected SOM class.

The radio buttons beneath the method list, labeled Smalltalk managed and SOM managed, display the memory ownership policy for the return value of the selected method. These buttons are not displayed if memory ownership does not apply to the return value. Memory ownership conventions are discussed in more detail in the Visual Smalltalk Enterprise Language Reference.

The top right pane displays the parameter list for the method, showing each of the parameters of the selected method. The radio buttons beneath the parameter list display the memory ownership policy for the selected parameter. These buttons are not displayed if no memory ownership policy applies to the selected parameter.

The contents pane at the bottom displays details about the items selected in the lists above. If no method is selected, it shows details about the selected class, such as its corresponding Smalltalk class name, its superclass(es), and modifiers from the interface repository. If a method is selected, it displays information about the method such as the corresponding Smalltalk method name, whether the method is called using IDL or OIDL calling conventions (see the SOM User’s Guide for details), and the types associated with the parameters and return value.
Figure 7-2 below shows the SOM Type Definition Browser. The type list at the left shows the SOM type definitions. The details pane on the right shows detailed information about the selected type definition, such as its size and alignment. Structures and unions display their fields here, while an enum displays its enumeration values.
Building Smalltalk Proxy Classes

The SOM Class Browser and SOM Type Definition Browser are used to build Smalltalk classes that provide access to the operations of the SOM classes and SOM typedefs they represent. These automatically-generated Smalltalk classes are called proxies, since they represent the actual SOM objects inside of Smalltalk.

Proxies for SOM Typedefs

Smalltalk proxies are defined for SOM types that are structs, unions, or enums. SOM types that already have a corresponding Smalltalk class, such as Integer or Float, require no additional proxy classes. Proxies for SOM structs, unions, and enums are subclasses of SOMStructure, SOMUnion, and SOMEnumeration. SOM pointer, array, and sequence types are represented by the Smalltalk classes SOMPointer, SOMArray, and SOMSequence.
Mirroring the SOM Class Hierarchy

All proxy classes inherit directly or indirectly from **SOMProxyClass**. The Smalltalk proxy class hierarchy can parallel the SOM class hierarchy, or it can ignore and flatten the SOM class hierarchy so that each proxy class defines proxy methods for its corresponding SOM class as well as each of its superclasses.

Creating a parallel hierarchy preserves the organization in the SOM hierarchy and reduces the number of proxy methods defined through the use of inheritance. It introduces a number of proxy classes that may not be interesting to the SOM programmer, however.

The *Flatten hierarchy* option on the *Build Options* menu determines whether a separate Smalltalk proxy class is created for each SOM class in the SOM hierarchy, or whether all methods inherited and implemented by the selected SOM class are defined when the Smalltalk proxy class is built. If the SOM class hierarchy is flattened, each Smalltalk proxy class is a direct subclass of **SOMProxyClass**.

It is not possible to recreate the SOM class hierarchy exactly if it uses multiple inheritance. Multiple inheritance is resolved in Smalltalk/V by inheriting from the proxy for the primary parent of the SOM class, and defining the methods inherited from all secondary parents of the Smalltalk proxy class.

For example, if the SOM class **Child** inherits the method *isPrimaryParent* from class **PrimaryParent** (its primary parent) and method *isSecondaryParent* from class **SecondaryParent** (a secondary parent), the Smalltalk proxy class **Child** would have the Smalltalk class **PrimaryParent** for its single superclass, hence inheriting *isPrimaryParent*, and directly implement *isSecondaryParent*.

Mapping SOM Names to Smalltalk Class Names

When building Smalltalk/V proxy classes, SOM names must be mapped to Smalltalk names. Normally the name used for the Smalltalk class is the same as that of the SOM class or type definition.

A name conflict occurs when a SOM class or type definition name is the same as that of an existing Smalltalk class. If a name conflict arises, the text “SOM” is added to the beginning of the SOM name when creating the Smalltalk name. For example, if a SOM class is named **OrderedCollection**, the name proposed for the Smalltalk proxy class will be **SOMOrderedCollection**. If that still does not
resolve the conflict, the text ".I" is appended to the SOM name. The Build Options / Prompt for class mappings menu item lets you enable and disable the confirmation of the proposed names.

**Building Smalltalk Classes Into Unloaded Packages**

The SOM tools always generate Smalltalk classes and methods into an open package. Because the generated code is not loaded, you can generate (or regenerate) code for existing classes without risk of breaking currently executing code.

After you have examined the open package, you can load it into your image.

The Build Options / Set package... menu item lets you set the name of the package the Smalltalk classes are generated into.

**Automatically Building Referenced Classes**

A SOM class or type definition usually refers to other SOM classes and type definitions, for example:

- SOM classes referenced as superclasses,
- SOM classes and type definitions referenced by structure elements, and
- SOM classes and type definitions referenced as parameter types.

The SOM tools can automatically build Smalltalk proxies for the referenced SOM classes and type definitions. Whether or not referenced classes are automatically built is controlled by the Build Options / Build Referenced Classes menu item.

When the class building tools are configured to automatically build Smalltalk proxy classes for referenced SOM classes, they perform a transitive closure of the referenced classes. This can result in classes referenced by more than one SOM class, such as SOMObject (the superclass of all SOM classes) being built multiple times if you generate different SOM classes into different packages.

You can specify a clamp list—a list of packages to check for referenced proxy and type definition classes. Before a class is built to satisfy a class or typedef reference, each package in this list is searched for such a class definition. If the proxy or type definition class is found in one of these packages, it is not generated again.

Building referenced classes automatically, especially when used in conjunction with a clamp package list, simplifies proxy building significantly. One package that is almost always added to the
clamp package list is the SOM Proxy Core Skeleton, which defines a number of SOM proxy classes that are supplied with the basic system.

Proxy Methods for SOM Classes

For each method in the SOM class, two methods are defined in the Smalltalk proxy class. The first method performs the actual SOM message send. The second method packages the parameters into a form suitable for passing to the SOM method, calls the first method, and then creates a Smalltalk result from the SOM method result.

These methods are divided into the SOM api and SOM api wrappers method categories for convenient browsing.

See your Visual Smalltalk Enterprise Language Reference for more information about the conventions used to interface between SOM and Smalltalk.

The Build Options Menu

Both SOM browsers have a Build Options menu. You use the items on this menu to set the options to be used when Smalltalk proxies for SOM classes and type definitions are automatically constructed.

Many of these build options have checkmarks next to them that appear when the option is enabled for the next build. Each item is checked when you select it and unchecked if you select it a second time.

The Build Options menu contains the following items:

- **Build Options / Flatten hierarchy**
  When the Flatten hierarchy option is not checked, Smalltalk classes are built to mirror the SOM class hierarchy as closely as possible, differing only where necessary because of multiple inheritance differences. The class builder uses the Smalltalk/V proxy for the primary superclass in the SOM hierarchy as the superclass of the class being built. The class builder only implements proxy methods directly implemented by the SOM class or methods inherited by the SOM class from a secondary superclass.

  When the Flatten hierarchy build option is checked, the Smalltalk proxy class is made a direct subclass of SOMProxyClass. All methods of the SOM class, both
inherited and directly implemented, are implemented in the Smalltalk proxy class. While this option appears on the Build Options menu of the SOM Type Definition Browser, it has no effect when creating typedef proxies.

- **Build Options / Prompt for class mappings**
  If checked, the build tools open a dialog every time a new Smalltalk class is generated that lets you confirm or change the proposed Smalltalk class name. When not checked, the proposed name is used without confirmation.

- **Build Options / Build referenced classes**
  When checked, the SOM tools automatically build Smalltalk proxy classes for referenced SOM classes and type definitions. If this build option is not checked, the referenced classes are left undefined.

  Enabling this build option simplifies proxy building significantly, especially when used in conjunction with the *clamp package list...* option described below.

- **Build Options / Set package...**
  This menu item opens a dialog that lets you name the open package that the generated classes will be added to. This package will be created if it is not currently open.

- **Build Options / Inspect name registry**
  This menu item opens an inspector on the proxy map, described below. You can use this inspector to see or modify how SOM classes are mapped to Smalltalk classes. In general, this menu item is not used. The information it makes available is available and changeable through other browsers. This is, however, the only avenue for changing a name mapping once it has been entered.

- **Build Options / Load mapping file...**
  This menu item opens a dialog that lets you load the current proxy map from a file.

- **Build Options / Save mapping file...**
  This menu item opens a dialog that lets you save the current proxy map to a file.

  The SOM class mappings are built using tools and dialogs and can represent a significant amount of work. Memory mapping conventions, in particular, are set on a parameter-by-parameter basis and this menu pick allows you to save that work so it can be easily retrieved if the proxy classes are to be regenerated at a future time.
• **Build Options / Clear mappings...**
  This menu item clears the current proxy map. You may want to do this when beginning work on a new set of SOM proxy classes.

• **Build Options / Clamp package list...**
  This menu item opens a dialog that lets you select the packages that contain Smalltalk proxy and type definition classes that terminate the reference chain when the *Build referenced classes* build option is enabled. The list you select packages from includes all the open and loaded packages in the image.

### SOM Class Browser *Class* Menu

The SOM Class Browser has a *Class* menu. This menu contains the following menu choices.

• **Build class** builds a Smalltalk proxy class for the selected SOM class and places it in the package previously defined with **Build Options / Set package...**.

• **Sort alphabetical** sorts the SOM class names in the class list alphabetically by name.

• **Sort hierarchical** sorts the SOM classes in the class list by superclass. Subclasses are shown indented, as in the Class Hierarchy Browser.

• **Show only defined methods** shows only methods implemented by the selected class in the method list.

• **Include inherited methods** shows methods both implemented in the selected class and inherited by the selected class.

• **Update** causes the tool to discard its current class list and rebuild the class list from the interface repository.

### SOM Type Definition Browser *TypeDef* Menu

Once additional menu is defined in the TypeDef browser, the TypeDef menu. It has only one menu item:

• **Build class for typedef** generates a Smalltalk class for the selected typedef using the build options previously set. The Smalltalk class is added to the package you specified with the **Build Options / Set package list...** menu item.
SOM Implementation Packages

The SAMPLE\SOM\SRCREPOS subdirectory of your Visual Smalltalk Enterprise installation directory is a repository that contains two clusters that hold the SOM system implementation. If you installed the Smalltalk libraries provided in the SOM directory, you got source code but it was not organized into packages. Installing the packages provides additional organization and comments to help you understand the system.

The SOM product cluster defines the system and contains two clusters: the SOM Base Cluster and the SOM Tool Cluster.

SOM Base Cluster

The SOM Base Cluster holds packages that define the classes and methods that provide basic SOM support. These packages are:

- **SOM Control**
  This package contains the SOMControlManager class and global object definitions used by the SOM system.

- **SOM DLLs**
  This package defines subclasses of DynamicLinkLibrary for the three DLLs used in SOM—SOM.DLL for basic SOM functions, SOMD.DLL for DSOM functions, and SOMTC.DLL for TypeCode functions.

- **SOM Proxy Base Classes**
  This package defines class definitions for the superclasses for all generated proxy classes. All Smalltalk proxies for SOM classes inherit from SOMProxyClass. Smalltalk typedef classes (except for pointers) inherit indirectly from SOMProxyTypeDef. The immediate subclasses of SOMProxyTypeDef define the types that proxies can be defined for.

- **SOM Proxy Core Skeleton**
  This package defines the basic set of SOM classes needed to support the SOM tools. It can be viewed as example of accessing SOM functionality. Since it includes proxy classes to access basic SOM functionality, you will probably want to include this package in any SOM application you build.

- **SOM Proxy Enhancements**
  This package contains enhancements to the SOM Proxy Core classes, including printing methods for the core classes, and enhancements for some proxy methods to make them more robust. This package also contains special
SOM proxy classes, such as **SOMAny**, **SOMArray**, **SOMSequence**, and **SOMTypeCode**. These classes implement behavior for SOM data types that have no specific SOM class or typedef.

- **SOM System Additions**
  This package contains methods that have been added to core system classes to support SOM.

**SOM Tool Cluster**

The **SOM Tool Cluster** holds packages that define the classes and methods that support the SOM browsers and the functions that they invoke. These packages are:

- **SOM Bootstrap Code**
  This package contains a loose method that can be executed to generate a package that defines the classes found in the **SOM Proxy Core Skeleton** package. This generated package contains a superset of the classes and methods found in the shipped package. These definitions were not added to the shipped package since each method must be checked for correctness, especially where memory management conventions are involved.

- **SOM Browsers**
  This package contains the classes that implement the browsers.

- **SOM Build Manager**
  This package contains the classes that manage the build process, including maintaining selected options and mappings.

- **SOM Class Builder**
  This package contains the classes that build proxy Smalltalk classes for SOM classes. **SOMProxyClassBuilder** is the main class. Supporting classes handle building methods and managing parameter and return values.

- **SOM Typedef Builders**
  This package contains classes that build Smalltalk classes for SOM type definitions. Currently Smalltalk classes can be built for SOM **enums**, **foreigns**, **structs**, and **unions**.

**SOM Proxy Map**

The SOM Proxy Map is an implementation detail that is made visible here for advanced users who may want to customize the SOM tools. Most users can ignore this section.
The SOM Proxy Map maps SOM classes and type definitions to corresponding Smalltalk proxy classes. It is a dictionary keyed on SOM name whose values are instances of SOMClassBuildSpecification. A SOMClassBuildSpecification has instance variables for Smalltalk name, SOM name (which is redundant information, included as an optimization), and a dictionary of method specifications.

This second level dictionary is keyed on method name and has values that are instances of SOMMethodBuildSpecification. These instances define the Smalltalk method name that corresponds to the SOM method name as well as memory ownership conventions. There is an instance variable for ownership of the method return values and a dictionary for parameter convention, whose keys are parameter names. The memory ownership is specified by the symbols SOM or SMALLTALK.
Appendix A

Problem Solving

This appendix describes system administration and problem solving procedures that are not part of your daily development activities. These topics include

- conflicts, the Conflict Notification dialog, why conflicts occur and how to resolve them.
- the recovery log file, the Recovery Log Workspace, and how to recover from crashes.
- working with inaccessible repositories.

Conflicts and How to Resolve Them

This section describes the ways in which definitions can conflict, the Conflict Notification dialog, what causes the conflicts, and suggestions for resolving them.

Conflicts can occur when you load a module, when you unload a module or when you migrate from one revision of a module to another. By default, conflicts occur between the definitions in one module and those in another. However, if you have so specified in your preferences, conflicts can also occur between the definitions in a module and items in the base class library or because class definitions are being removed for instances that are present. To receive information about such conflicts, set the conflict notification preferences in the Conflict Preference Editor.

**NOTE:** See the section entitled *Conflict Preference Editor* in chapter 2 for further information about these preferences.

If preferences have not been set to report these conflicts, definitions that would conflict with items in the base class library overwrite them instead, and class definitions can be removed even if instances still exist.

If a conflict is detected, a Conflict Notification dialog is opened that reports the errors. The requested load, migrate, or unload operation does not occur.
To resolve a conflict, you can either modify definitions in modules already loaded in your image, modify the base class library itself, or open the module you could not load, and modify the definitions it contains before trying to load it again.

**Conflict Notification Dialog**

The Conflict Notification dialog is a non-modal informational dialog that displays the error messages resulting from a failed load, unload, or migrate operation.

*Figure A-1: Conflict Notification Dialog*

The top pane contains a brief description of the operation that produced the conflict. The text pane below it contains a description of the nature of the conflict. The radio buttons control whether to display a relatively short summary or a complete description of the error. The complete description provides a key phrase summarizing the conflict (such as [Undeclared Pool Conflict] in the figure above) that can be matched against the headings in the following section for more information.

Buttons at the bottom of the dialog are

- **OK** dismisses the dialog.
- **Compare** opens a Definition Comparison Browser on the definitions that caused the conflict. The dialog remains open.

**Resolving Conflicts**

If the information presented by the Conflict Notification dialog is not sufficient, refer to the appropriate header in the following sections to find out why the error occurred and how to resolve it.
Existing Definition Conflict

Multiple Definition Conflict
A conflict between a loaded module and an open one produces an Existing Definition Conflict. If the conflict is between multiple open modules, the message is Multiple Definition Conflict.

A definition in a module you are loading (or in a revision to which you are migrating) is already defined in another module loaded into your image (or possibly in the base class library, if the preference for reporting base image conflicts is turned off). The definition has the same name and is of the same kind.

For example, you have already defined an instance method named isNorth for the class Latitude, but now you are trying to load a module containing the same or different source code for an instance method of the same name.

NOTE: A class method and an instance method of the same name, defined in the same class, do not conflict.

Resolutions
This conflict can be resolved in any of the following ways:

- Rename one of the conflicting definitions in the standard class library or an already loaded module.
- Load one of the modules, rename the conflicting definition, then load the other module.
- Open one of the modules and delete the conflicting definition.

Type Mismatch Conflict
A global definition—a definition of a global variable, pool, or class—in a module you are loading (or a revision to which you are migrating) is already defined in another module loaded into your image or in the base class library. The definition has the same name but is a different kind of global.

For example, you are loading a module defining the class Map, but another module already loaded in your image defines a global variable named Map.

NOTE: This type of conflict can occur with the base class library regardless of the setting of the conflict preference items.
Resolutions
This conflict can be resolved in any of the following ways:

- Rename one of the conflicting global definitions in the standard class library or an already loaded module.
- Load one of the modules, rename the conflicting definition, then load the other module.
- Open one of the modules and delete the conflicting definition.
- Open one of the modules and change the kind of the conflicting definition to be the same kind as the definition in the other module, or the base class library.

Existing Subclasses Conflict

Undeclared Superclass Conflict
If you unload the a module or migrate to another revision that deletes a class that has subclasses, the Existing Subclasses Conflict is the result. If a class in a module you are loading has an undefined superclass, you see an Undeclared Superclass Conflict.

For example, you are deleting a module containing the class `CartographicUnit`, while its subclasses `Latitude` and `Longitude` are defined in another module still loaded in your image.

**NOTE:** This type of conflict can occur with the base class library regardless of the setting of the conflict preferences items.

Resolution
This conflict can be resolved in any of the following ways:

- Define the missing superclass in a module already in your image.
- Find the module that defines the missing superclass and load it.
- Delete the class with the missing superclass.

Undeclared Global Conflict
A class in a module you are loading uses an undefined pool, or a method in a module you are loading belongs to an undefined class, or a pool variable in a module you are loading belongs to an undefined pool. Or, if you delete the selected module, or migrate to the selected revision, one of these conditions will occur.
For example, you are loading a module containing the method isNorth, which belongs to the class Latitude, but the class Latitude is defined neither in that module, nor in any other module loaded in your image, nor in the base class library.

NOTE: This type of conflict can occur with the base class library regardless of the setting of the conflict preference items.

Resolution
This conflict can be resolved in any of the following ways:

- Define the missing class or pool in a module already in your image.
- Find the module that defines the missing class or pool and load it.
- Delete the method with the missing class, or delete the pool usage definition from the class, or delete the pool variable definition.

Variable Name Conflict
A class in a module you are loading includes a class variable of the same name as a class variable belonging to another class in the same hierarchy, or an instance variable of the same name as an instance variable belonging to another class in the same hierarchy, or a class instance variable of the same name as a class instance variable belonging to another class in the same hierarchy.

If the variables are not of the same kind, they do not conflict.

For example, you are loading a module containing the class Van, which defines a class variable Capacity. However, the superclass of Van—the class Bus—already defines this class variable and is already a part of your image.

NOTE: Pool usage is not inherited. Therefore, the same pool usage definition in two classes belonging to the same hierarchy does not cause a conflict either. Instead, it is required if the classes must both use the pool.

Resolution
This conflict can be resolved in any of the following ways:

- Redefine one class to change the name of one of the variables.
• Change the kind of one variable so that they are not both the same kind.

• Redefine one class to delete one of the variables.

**Indexable Type Conflict**

Classes in the same inheritance hierarchy cannot mix indexable types. That is, a byte-indexable superclass cannot have an object-indexable subclass, or vice versa.

• If a class is object-indexable, any classes below it in the hierarchy must also be object-indexable.

• If a class is byte-indexable, any classes below it in the hierarchy must also be byte-indexable.

• Byte-indexable classes can neither define nor inherit instance variables.

This conflict occurs when you try to load a module containing a class that violates one of these rules.

For example, suppose you wanted to create a subclass of String called SubString, to be used for an efficient implementation of text editing. SubString defines three instance variables: baseString, startIndex, and stopIndex. Because String is a byte-indexable class belonging to the base class library, loading a module containing SubString will produce a conflict.

**Resolution**

Reimplement one of the classes so that the rules above are not violated. How you choose to do so depends upon your application, but one technique to consider is:

• Redefine the class so it is neither byte-indexable nor object-indexable.

• Add an instance variable to the class to encapsulate the indexable structure it requires.

This variable can be an instance of an indexable class such as Array or ByteArray.

• Add accessing methods to your class such as **at:** and **at:** **put:** to access the new variable.
Conflicts and How to Resolve Them

Pool Reference Conflict
A pool is being removed but is still referenced from a class. A class definition remains in your image that refers to a pool definition in a module you are unloading. Or, a class definition remains in your image that refers to a pool definition that does not exist in the revision to which you are migrating.

Resolution
Browse references to the pool with Smalltalk / Browse -> References to...). Then:

- Delete the references to the pool.
- Move the pool definition to a module you are not going to remove.
- Move the pool definition to the revision to which you wish to migrate.

Undeclared Pool Conflict
A class definition refers to a pool that is not defined.

Resolution
Browse references to the pool with Smalltalk / Browse -> References to...). Then:

- Delete the references to the pool.
- Add the pool definition.

Existing Instances Conflict
A class in a module you are removing has instances present in your image. Or, the revision to which you are migrating does not define the class whose instances are present.

For example, you delete the Tutorial module while the Navigator window you had running remains open.

Resolution
This conflict can be resolved in any of the following ways:

- Browse references to the class and delete all its instances.
  (Choose Smalltalk / Browse -> References to...)
- Move the class definition to another module.
- Move the class definition to the new revision.
Multiple Specification Conflict

Multiple contradictory specifications exist for a module. More than one specification with the same module name exists (in the image and in modules to be loaded), and one or more other fields in the specifications have contradictory information.

For example, you have a cluster loaded into your image that specifies version 1.2 of the package Cartography. If you attempt to load a cluster that specifies version 1.1 of the package Cartography, the load operation will fail with a Multiple Specification Conflict.

Resolution

This conflict can be resolved either of the following ways:

- Using the Cluster Organizer, open the clusters you are attempting to load. Edit the portion of the specifications for the problem module so that they match the data from the loaded package, if any. If there is no loaded package, then edit the specifications to they match among themselves. Commit the edited clusters and retry the operation with the new revision of the cluster.

- Using the Package Browser, migrate the problem package to a revision that matches the specifications from the clusters you are attempting to load.

Unresolved Specification Conflict

There is not enough information to resolve the specification to a module. The set of specifications with the module name (in the image and in modules to be loaded), taken together, does not have all required fields filled in.

For example, you attempt to load a cluster in your image that contains a minimal specification for the package Cartography. The image and other specifications in the cluster do not contain any other specifications for the package Cartography. The attempt to load will fail with an Unresolved Specification Conflict.

Resolution

This conflict can be resolved in any of the following ways:

- Load a revision of the problem module that creates a full specification. Then retry the operation.

- Using the Cluster Organizer, open the cluster containing one of the matching specifications. Edit the specification so that it completely specifies the problem module. Commit the edited cluster and retry the operation with the new revision.
Recovering From an Image Crash

Instructions for recovering your work after an image or system crash are given in the Visual Smalltalk Enterprise User’s Guide. This section covers a few more details specifically of interest to Visual Smalltalk Enterprise user.

The Change Log

Although Visual Smalltalk Enterprise images still use CHANGE.LOG, it is no longer used to recover changes from an unsaved image. Visual Smalltalk Enterprise uses the change log to record source code and versioning information for all methods and other permanent source pointers.

The Recovery Log

Visual Smalltalk Enterprise adds a new log file named RECOVER.LOG. It can be found in the current directory with CHANGE.LOG. The recovery log contains a history of every change you make to your image, including all expressions you have evaluated with Do It or Show It. The recovery log contains the permanent information maintained by the change log, plus these evaluated expressions.

The format of the recovery log differs from that of the change log. All changes should be recovered from RECOVER.LOG; you should not use CHANGE.LOG.

Since RECOVER.LOG records every change you make, it can grow quite large over time. Unlike the change log, the information in this file is only read when you open the Recovery Log Workspace to recover from a crash. You can then select those chunks that you need to restore and file them in.

You should not try to delete the recovery log file manually. The image remembers the place in the file that marked the last image save, so you should ask the Recovery Log Workspace to discard the contents with the Recover / Reset Log menu item.

The recovery log is also modified when you condense changes.

• Create a new open cluster. Edit the cluster to include a complete specification for the problem module. Designate this specification as optional. Commit and load the new cluster. Retry the load operation.
The Recovery Log Workspace

You can inspect the recovery log for any changes you made since you last saved the image by opening a Recovery Log Workspace with the Smalltalk / Open -> Recovery Log menu item. The Recovery Log Workspace opens displaying the text representing the changes you have made since your image was last saved.

*Figure A-2: Recovery Log Workspace*

### NOTE: Although the Recovery Log Workspace has File It In and Do It menu items, the syntax of the expressions is different than that used for either Smalltalk code or in the change log. Do not use the recovery log workspace for any purpose other than recovering changes when you were not able to save your image.

### Recovery Log Workspace Recover menu

The Recovery Log Workspace has one new menu, Recover, that contains the following items:

- **Skip Do Its** controls whether Do It expressions are executed during recovery. When checked, these expressions are ignored during the file in process. By default, this item is checked and Do Its are not executed.

- **View Unsaved Changes** causes the Recovery Log Workspace to show only those changes that have been made since the last image save. By default, this item is checked.

- **View Entire Log** causes the Recovery Log Workspace to show the entire contents of RECOVER.LOG.
• **Reset Log** discards all recovery information after confirmation from the user. If confirmed, this operation deletes the contents of the RECOVER.LOG and updates the contents of the workspace to reflect the new state of the log which is empty.

**Recovery Log Format**

The recovery log records each time you save your image with a line like:

```plaintext
*** saved image on: 10/31/94 04:33:35 PM ***
```

A similar line marks every time you exited your image from the *File / Exit* command (whether you saved the image or quit without saving). If the last line in the Recovery Log Workspace is not an exit, it means that your image crashed.

Your changes are recorded as sequences of strings; each sequence separated from the other by one or more blank lines. Each sequence is terminated with an exclamation mark.

**NOTE:** If you want to select and replay a single change, be certain to select from the blank line preceding the change through the exclamation mark terminating the change.

The exact format of each change is internal to Visual Smalltalk Enterprise and subject to change in the future, so it is not documented here.

You will normally not be concerned with the contents of each change since you will want to select and replay all changes, except perhaps for the last change that caused the crash or *Do Its*. (If the *Recover / Skip Do Its* menu item toggle is on, *Do Its* are ignored when recovering changes.)

However, there will be times you need to examine the changes—to determine what has been changed since the image was last saved or to locate a change that caused the image to crash, for example. Inspecting the text of a change should be sufficient to determine its purpose.

**Using the Recovery Log Workspace to Recover Changes**

You can replay the changes recorded in the recovery log by selecting the text to be replayed and choosing *Smalltalk / File It In*. If you are viewing unsaved changes only, you can select the
entire file. If you are viewing the entire log, you will probably select text between the last “saved image on:” line and the end of the file.

If an error occurs while trying to recover one of the changes, you are notified and given a choice whether to try to continue or cancel the operation.

NOTE: The “saved image on:” and “exited image on:” lines are comments only so they are ignored by the file in process. Even if you have indicated that you want to replay Do Its, filing in these lines will have no effect—the image will not be saved or exited as a consequence.

To recover changes from the Recovery Log Workspace:

2. If the Recovery Log Workspace only shows the “saved image on:” line, your image is current and there are no unrecovered changes.
3. Otherwise, select all the text between the “saved image on:” line that matches the date and time of your image file and the end of the recovery log.
4. Choose the Smalltalk / File It In menu item in the Recovery Log Workspace to replay all the recorded changes.
5. When processing completes, all changes have been reinstalled in your image. It is a good idea to save your image at this point, recording the changes in the image instead of RECOVER.LOG.

NOTE: If you know that a change you made caused the image to crash, you should avoid replaying that change when you recover changes in the Recovery Log Workspace.
Working with Inaccessible Repositories

When repositories on a network become inaccessible for any reason, the module information pane in the Package Browser contains a status line indicating that the repository is not connected.

The --WORK-- Directory

When you access an open or loaded module stored in a PVCS repository, information about that module is copied to your local workstation in a subdirectory of your current working directory named --WORK--. If the repositories become temporarily inaccessible, you may be able to continue working for a short period of time with modules that you had accessed before the time the repository became inaccessible. You cannot perform any operation that requires access to the original repository such as committing to the repository, loading, or migrating. You can, however, unload or close packages when their repositories are inaccessible.

When the repositories are available again, the ‘not connected’ status line disappears and you can perform all normal operations.

When you exit the image, the --WORK-- directory contents are removed.

Detaching from Repositories

If you have advance notice that a repository will become inaccessible (such as a server shutdown during your working time), you should detach from the repository using Smalltalk / Team -> Detach.

Detaching copies the source for all currently loaded and opened modules in your image to your changes file, equivalent to disconnecting Visual Smalltalk Enterprise from all repositories. It automatically compresses your changes file and saves the current image to assure that the changes file is as small as possible and to prevent you from accidently exiting without saving.

See Detaching from a Repository to Work Off-Site in chapter 5 for more information about how to work safely when repositories are unavailable.
Working Without a Repository

You can only open, lock, load, migrate, or commit modules and clusters to a repository when you are connected to it and it is physically accessible.

If a repository is unavailable and pessimistic locking is on, you are given the opportunity to proceed even if you can’t lock the module you are trying to edit.

If you need to commit a new revision of a module, create a local repository and commit new revisions to the local repository. When the shared repository becomes available, you can commit the new version of the module back to the shared repository.

If several developers create local revisions of a module while a shared repository is inaccessible, use the comparison tools to integrate changes into a new revision.
This appendix describes the rules used to format Smalltalk methods when you choose the following menu items:

- *Method / Format* in the Package Browser
- *Debugger / Format* in the Debugger
- *Definition / Format* in the Package Organizer
- *Definition / Format* in the Definition Group Browser

After describing the programmatic interface for the formatter, this appendix begins with the formatting rules for the top level of a method and proceeds to smaller and smaller syntactic units.

Each section below begins with a description of the syntactic unit and a template for formatting that unit. Specific formatting information and examples follow.

### Programmatic Interface to the Formatter

To access the formatter programmatically, evaluate an expression of the form:

```
ToolInterface current
  formatSource: <source string to be formatted>
  withMethodPattern: <true if method, false if expression>
  notifying: <text pane or nil>
```

For example, the workspace below shows the result (highlighted) of executing an expression to format the source for a method.
Methods

Methods are composed of five parts: a method header, optional comments, an optional temporary declaration list, an optional primitive specification, and the code block for the method.

\[
\begin{align*}
<\text{method header}> \\
<\text{comments}> \\
<\text{temporaries}> \\
<\text{primitive specification}> \\
<\text{code block}>
\end{align*}
\]

Temporaries and primitives can appear in either order.

The Method Header

The method header is left-justified. Unary message selectors appear alone on a single line.

\text{printString}

Binary message patterns also appear on a single line, with the binary selector and parameter name separated by a single space.

\text{+ aPoint}

Keyword messages are broken across multiple lines, one keyword and one parameter name per line.

\text{displayOn: aGraphicsContext at: offset}
Method Comments

A method can have any number of comments. Each comment begins on a separate line, with the first line of the comment indented one tab stop under the method header. Formatting within the comment is unchanged: everything between the quotes is copied verbatim.

```smalltalk
printString

"Answer a String containing a human-readable representation of the receiver."
"This method should not be overridden by subclasses. If a subclass requires a different printed representation, it should override the method #printOn: instead."
```

The comments are separated from the following material by a single blank line. If the method is uncommented, a single blank line still separates the method header from the following material.

Temporary Declarations

The temporary declarations appear on a single line, indented at the base indentation. Exactly one space is placed between each pair of temporary names, and one space separates each of the vertical bars from the temporary names.

```smalltalk
| xOffset yOffset |
```

Primitive Specifications

If the method includes a primitive specification, it appears on a line by itself, indented one tab stop.

```smalltalk
new

"Answer a newly created instance of the receiver."

<primitive: 70>
```

The Code Block

The code block appears on the line after the primitive specification, indented one tab stop.
Code Blocks

Blocks of code are composed of a sequence of statements.

\[
\text{\texttt{<statement>}}
\]

\[
\ldots
\]

\[
\text{\texttt{<statement>}}
\]

Code is formatted relative to a base indentation which starts at one tab stop for methods.

Statements

Each code body can have any number of statements. Each statement begins on a separate line, indented at the current level of indentation. Many statements require more than one line, in which case each line is indented as specified below plus the current level of indentation. Adjacent statements are separated by a period.

\[
x\text{Offset} := \text{self left} + \text{aPoint x}.
\]

\[
y\text{Offset} := \text{self top} + \text{aPoint y}
\]

Blocks

A block is composed of optional block arguments and a code body enclosed within square brackets.

\[
[\text{\texttt{<code body>}}]
\]

\[
[\text{\texttt{<block arguments> |}}
\]

\[
\text{\texttt{<code body>}}]
\]

If the block includes no block arguments and the code body can be formatted on a single line, then the block is formatted on a single line. There are no spaces between the enclosing brackets and the code body.

\[
[\text{\texttt{self abs <= 100}}]
\]

If the block has arguments, they appear on a separate line. Each argument is followed by a single space, resulting in a space between the last argument and the vertical bar. There is no space between the opening bracket and the first argument, nor any space between the colon and the argument name.
If the code body requires multiple lines, or if there are block arguments, each line of the code body is formatted on a separate line, indented by the base indentation. No space is placed between the last statement in the code body and the closing bracket.

```
[:firstArg :secondArg |
  firstArg <= secondArg]
```

### Statements

There are three types of statements: expressions, assignment statements, and return statements.

### Expressions

There are four types of expressions: simple, unary, binary, and keyword.

#### Simple Expressions

A simple expression is either a variable name or a literal.

```
<variable name>

<literal>
```

Variable names simply print as themselves.

```
aString
```

Literals preserve the formatting specified by the programmer.

```
#(
  ('again' 'undo')
  ('cut' 'copy' 'paste'))
```

#### Unary Expressions

A unary expression is composed of a receiver and a unary message selector.

```
<receiver> <unary message selector>
```

The unary message selector always follows the receiver, separated by a single space. The receiver gets no special formatting.

```
anArray size
```
Binary Expressions

A binary expression is composed of a receiver, a binary operator, and an argument.

\[ \text{<receiver> <binary operator> <argument>} \]

The receiver, operator, and argument are formatted on the same line. The operator is separated from both the receiver and the argument by a single space. The receiver and argument are formatted as normal.

\[ 50 \ @ \ 75 \]

Keyword Expressions

A keyword expression is composed of a receiver and any number of keyword/argument pairs.

\[ \text{<receiver> <keyword} \ <argument> \ ... \ <keyword} \ <argument> \]

\[ \text{<receiver> <keyword} \ <argument> \]

\[ ... \]

\[ \text{<keyword} \ <argument> \]

\[ \text{<receiver> <keyword} \]

\[ ... \]

\[ \text{<keyword} \]

\[ \text{<argument>} \]

If the receiver and all of the arguments can be printed on a single line, and if the number of pairs is relatively small (2 or 3), the keyword/argument pairs are printed on the same line as the receiver.

\[ \text{self data copyFrom: 1 to self data size – 1} \]

\[ \text{self map at: key ifAbsent: [self invalidMapLocation]} \]

Otherwise, each keyword/argument pair is printed on a separate line, indented one tab stop more than the receiver. The arguments are printed on the same line as the corresponding keyword, separated by a single space, if the argument can be printed on a single line.

\[ \text{self data} \]

\[ \text{replaceFrom: startIndex} \]

\[ \text{to: self size} \]
Cascaded Messages

```
with: replacementData
startingAt: 1
```

If the argument requires more than one line, then the argument is on a separate line, indented one tab stop more than the keyword.

```
self data
do:
    [:each |
      each printOn: aStream]
```

**Assignment Statements**

Assignment statements consist of a variable name, an assignment operator, and an expression.

```
<variable name> := <expression>
```

The expression is formatted as described above, with the variable name and assignment operator immediately preceding the message.

```
d := 4 * a * c
```

If a value is assigned to more than one target, all of the targets and assignment operators appear on the same line as the expression.

```
a := b := c := 0
```

**Return Statements**

Return statements consist of a return arrow and an expression.

```
^<expression>
```

The expression is formatted as described above, and the return arrow immediately precedes it.

```
^result
```

**Cascaded Messages**

Cascaded messages consist of a receiver and two or more message-sends.

```
<receiver> <message-send>; ...; <message-send>
```

```
<receiver>
  <message-send>;
  ...
  <message-send>
```
The message-sends are separated by semicolons. No semicolon appears after the last message-send. No space appears between a message-send and the semicolon that follows it, and either a single space or an end-of-line appears between the semicolon and the following message-send.

If the receiver and all of the message-sends can be formatted on a single line, and if the number of message-sends is relatively small (2 to 4), a cascaded message is formatted on a single line.

Transcript show: value printString; cr

If it cannot be formatted on a single line, a cascaded message is formatted with the receiver on the first line and each message-send on a separate line indented one additional tab stop.

(Dictionary new
   at: $a put: #apple;
   at: $b put: #box;
   at: $c put: #car;
   at: $d put: #dog;
   yourself)

Exceptions

Several exceptions to the preceding formatting rules are detailed below.

Conditionals

Conditional execution using the messages ifTrue:, ifFalse:, ifTrue:ifFalse:, and ifFalse:ifTrue: are formatted with the keyword(s) on a separate line, indented one tab stop more than the conditional expression to make the conditional nature of the block arguments more obvious.

(index > 0 and: [index < self size])
   ifFalse: [self invalidIndex]

Iteration

Iteration using the message to:do: and to:by:do: should be formatted with the starting, ending, and step values on a single line because they control the iteration. It is formatted as if the to: or to:by: message were parenthesized, creating an interval as the receiver of an enumeration message.
Rather than specify in the preceding discussions all possible placements of comments and how to deal with them, we have developed the following general guidelines:

- The contents of a comment are never changed. The formatting may be important to the content.
- Comments before a method body—method comments—are formatted differently than comments within the method body.
- Comments preceding the last blank line before the code body, including comments before or within the method header, are method comments; their formatting is unchanged.

For example, the following method has two method comments (separated by a blank line) and one comment in the method body. Formatting removes the blank line between the first and second comments.

```smalltalk
default
  "Answer the default instance of the receiver."
  "ExampleClass default."
  "Check to see if the default already exists. If not, create it."
  default == nil
  ifTrue: [default := self new].
^default
```

Visual Smalltalk syntax specifies that comments can appear between any two tokens (the before token and the after token). Comments within the method body are formatted according to the remaining rules.

- Comments associate with a single token. If a <Return> is between the before token and the comment, the comment associates with the after token. Otherwise, the comment associates with the before token.
- If the before token and the after token are formatted on the same line, the comment appears between them on the line separated from either by a single space.
- If the before and after tokens appear on different lines, the result depends on which token the comment is associated with.
Comments associated with the before token appear on the same line as the before token, separated by a single space.

Comments associated with the after token appear on a line by themselves, indented to the same tab stop as the line containing the after token.
Symbols

* (asterisk)
  SOM class names 299

. (period)
  in annotations 23
  cluster annotations 35
  in revision numbers 49
  statements, separating 328

; (semicolons) 332

[ ] (square brackets)
  in blocks 328

A
  aborting revisions 280, 293
  abstract classes 183
  access control 221
    configuring 61
  operating system 222
  to repositories 52
  access database
    adding groups 61
    adding users 61
    configuring 61
    creating 61
  access lists, creating 61
  AccessDB directive 61
  accessing
    See also opening
closed repositories 288
  entities
    in repositories 289–290
    revisions of 291–292
  method formatter 325–326
  open repositories 289
  repository components 286
  repository services 286
  revision physical data 279

ad hoc initializers
  See also class initializers
  creating 148, 193
  defined 26
  defining 32–33
  Definition/New -> Initializer command 148
  described 192

adding
categories
  to classes 127
  to packages 81–84
code to packages 187
conditional inclusion flags to modules 151, 156, 202
Mapping package to Map Browser
  Configuration Cluster 98
  specifications to clusters 200–201
  unresolved variables 171
  variables to definitions 130
after tokens 333

** all ** category 128
annotation labels, creating
  in Cluster Organizer 153
  in Package Organizer 149
Annotation menu
  Cluster Organizer 154
  Package Organizer 149
annotations
  See also comments
  cluster 35
  definition 27
  described 22
  package 149
  specification 37
viewing
  all in definitions or packages 142
  cluster 150, 153
  definition 144
  specification 154
ANSI standard support 66
API (Application Programming Interface) 245–246
application frameworks
  application-specific use of 184–185
  packages
    as basis for 184
    defining in 21
applications
  building
    from clean images 239
    clusters recording packages within 22
    as collections of packages 21
configuring with conditional inclusion flags 37
consisting of multiple packages 19
contained in single packages 18
distributing changes to 100
dividing into packages 182–185
extended packages and 235
loading modules for different platforms 45
recording structural and organizational information 41
repositories, running without 287
restructuring file-ins as packages 182
arguments
in binary expressions 330
block 328, 332
in keyword expressions 330
array types, proxies for SOM 301
assignment operators 331
assignment statements 331
asterisk (*)
SOM class names 299
at: method 316
at: put: method 316
author revision attribute 50

B
backing up repositories 51, 221
base class library
Existing Subclasses Conflicts 314
extending 194
minimizing changes to 235
Undeclared Global Conflicts 315
base classes
modifying safely 195
problems with modified 195
redefinition of, by packages 194
before tokens 333
binary expressions 330
binary message patterns 326
binary operators 330
block arguments
in conditional expressions 332
syntax 328
boldface
in manual xv, xvi
boldfaced text
tools, displayed in 104
Bottom (Definition/Move submenu) command 145
branch revision numbers
adding to revision numbers 121
dependency on parent numbers 49–50
Browse (Definition menu) command 146
Browse Packages (Smalltalk menu) command 105
Browse References command
Definition Group Browser Variable menu 171
Global menu 123
Package Browser Variable menu 131
browsers
See also SOM Class Browser; SOM Typedef Browser
locating modules with 102
overview 101
browsing
cluster code 79
definitions
Definition Group Browser 158–160
Message Browsers 160–161
messages for selected 146
overview 158
implementors 127
message senders and implementors 127
SOM classes 298, 299
SOM type definitions 300
SOM types 298
unresolved references 170–172
Build class (Class menu) command 306
Build class for typedef (Typedef menu) command 306
Build Library dialog 169–170
Build Library... (Module menu) command 119, 169
Build Options menu
Build Referenced Classes command 303
described 304–306
Flatten Hierarchy command 302
Prompt for class mappings command 303
Build referenced classes (Build Options menu) command 303, 305
build scripts
rebuilding images and 236–237
Build... (Build Library dialog) button 169
building
applications
from clean images 239
runtime images 243
Smalltalk Libraries from modules 239–241
Smalltalk proxy classes 301–304, 306
SOM classes managing process of 308
byte-indexable classes 316
C

CartographicUnit, class 18
Cartography package
  adding categories and methods to 81–84
  comparing revisions of 92–95
  creating revision to 88–89
  described 18–19
  viewing specifications for 80
cascaded messages 331–332
categories
  See also methods 81–84, 127
  creating 189
  creating methods in 189
  deleting from image 127
  hierarchical 125
  moving methods between 128, 189
  ordering 64
  purpose of 124
  searching for 127
  SOM api and SOM api wrappers 304
  sorting preferences for 64
  viewing organization of methods into 106
category list, Package Browser 106
Category Order Preference Editor 64
change logs
  compressing 210
  definition versions 26
  purpose of 319
CHANGELOG file
  building baseline images 239
  caution against using for crash recovery 319
changing
  See also editing; modifying
  cluster contents 37
  conditional inclusion flags list in image 156
  definitions. See revisions
  initialization order of definitions 187–188
  methods 84
  names to resolve conflicts 230
  superclass structure 123
CharacterConstants pool 130
circularities, resolving 232–234
clamp lists, specifying 303
Clamp package list... (Build Options menu) command 306
Class (Global/New submenu) command 122
Class Definition Editor, Package Browser
  defining classes in 116–117
  described 114
class definitions
  creating 112, 146
  editing 112, 114
  problems with modified base 195
  referring to undefined pools 317
  resolution of references to pool variables and superclasses 134
  SOM 307
  superclasses for generated proxy classes 307
  viewing 79, 96
Class Editors, Package Browser 112–114
class hierarchies
  circularities and 232–234
  defined in packages 20
  forming packages 183
  Indexable Type Conflicts 316
  mirroring SOM 302
  restructuring with packages 190
  subclassing Smalltalk/V proxy class under SOMProxyClass 304
  viewing 113, 123
Class Initializer (Definition/New submenu) command 147
class initializers
  See also ad hoc initializers; initializers
  creating 191–192
  defined 26
  described 191
  vs. initialize methods 30
Class Instance Variable... (Definition/New submenu) command 147
class instance variables
  creating 147
  defining 28–30, 117
class instances
  See also class hierarchies; class instance variables; classes; instance variables
  converting to instances of other classes 21
  representing repositories, entities, and revisions on disk 279
  undefined classes in revisions for 317
Class menu (SOM Class Browser) 306
class methods, instance variables and 28
Class Variable... (Definition/New submenu) command 147
class variables
  vs. class instance variables 28–29
  conflicts between 315–316
  creating 147
  defining 28, 117
Class... (Definition/New submenu) command 146
classes

See also proxies

abstract 183

adding categories to 127

in application frameworks 184

base library 194

building Smalltalk proxy 301–304

Class Editors and 112–114

conflicts between indexable types 316

custom tool box 246
defined in other packages, extending 234
defining 116–117
displaying all cluster package 79

extending for applications 185

initializing 190, 192

moving to new packages 97–98

packages holding

cooperating 20

inheritance-related 20

single vs. multiple 183

parts of, defining 27–33

pool variable access 31

programmatic interface 247–248

repository interface 280

SOM 299

Clear mappings... (Build Options menu)

close message
described 286

operation performed by 285

repository interface component

connections 279

Close... (Package menu) command 144

closing

clusters 39

packages

Close (Package menu) command 144

effects of 25

inaccessible repositories 323

repositories 288–289

repository components 286

Cluster menu
described 151

Load command 132

Cluster Organizers

annotating clusters 153

Cluster menu 151

complete and partial specifications 36

opening 80, 90, 150–151

parenthesized text in 104

purpose of 150

Specification menu 152

specification operations available in 43–44

cluster revisions

creating 91–92

described 49

clusters

See also Cluster Organizers; cluster revisions; specifications

characteristics of 35

configurations and groups 43

creating 200

defining 115

definitions in 106

described 33–34

installing SOM Tools from 298

loading

and resolving specifications 40

steps performed during 135

locating open vs. loaded 102

operations on 38

organization of packages in 22

role in team development process 196–197

SOM Base Cluster 307–308

**top level** 39

tracking modified 46

code blocks

format of 328

line location of 327

statements in 328

code body, comments in 333

code. See source code

collaborating classes 183

collapsing

cluster specifications 204

definitions 144

Color Preference Editor 65

Comment (Global/New submenu) command 123

comment message 285

comment: message

operation performed by 285

revision 294

comments

See also annotations

ad hoc initializer expressions 33, 148

class 27

cluster 35

committing module revisions and 212

described 23

displaying, for globals 123

editing

class 112

global variables and initialization expressions 110

module 109
Index

formatting 333–334
method 84, 327
module 115
package 185
recover log 322
repository component 284
revision 50, 119, 120
variable 117
viewing package 79
Commit All (Module menu) command 119
Commit Dialog
Cartography package 88
described 120–121
commit message 294
Commit... (Module menu) command 119
committed revisions 48
committing
clusters
described 39
tutorial 91–92
configuration clusters 99
modules 119
packages
creating new revisions and 88–89
described 25
to different repositories 121
Map Browser 98
to new repositories 121
unresolved references and 53
version histories and 209
revisions 211, 279, 293
Compare (Module menu) command 120
Compare (Package Definition Browser) button 167
Compare All command
(Definition Group Browser Definition menu 160
Message Browser Definition menu 163
Version menu 158
Compare menus
Definition Comparison Browser 165
Version Comparison Browser 163
Compare Versions command
Definition Group Browser Definition menu 160
Method menu 127
Package Organizer Definition menu 146
comparing
See also integrating
definitions 164
method versions 160
package revisions 49, 216–218
packages 92–95, 120, 166
versions
Compare All (Version menu) command 158
of methods 86–87
of source-based definitions 215–216
Comparison browsers 101
comparison tools
Definition Comparison Browser 163–165
Package Comparison Browser 166
Version Comparison Browser 161–163
compilation preferences editor 66
compiler options 66
Complete (Specification menu) command 153
Complete All (Specification menu) command 153
complete specifications
See also partial specifications;
specifications
contents of 36
converting to partial 206–207
completing specifications 153
components message
accessing all entities in repository 290
title revisions, accessing 291
operations performed by 285
component flavours message
title revisions, accessing 291
operations performed by 285
processing all entities in repository 290
correction errors 193
compressing change logs 210
conditional execution 332
conditional expressions, formatting 332
Conditional Inclusion Context editor 156–157
conditional inclusion flags
adding to modules 151, 202
class load process and 135
deleting 151
described 198
inclusion phase of cluster loading 40
purpose of 37
setting, in environment 202
in Specifications Editors 155
specification name field indication of 203
specifying modules for inclusion 156–157
substituting modules in clusters 45
conditional specifications 45
collection clusters
rebuilding images and 236
updating 90–92
collection files. See master collection files; PVCS \VCONFIG.EXE utility
configurations
  combining with groups 42–43
  optional specifications and 44
  recording revisions using 41
configuring
  development environment 55–60
  PVCS access control 61
Conflict Notification dialog. See also Conflict Preference Editor; conflicts; resolving conflicts 312
Conflict Preference Editor 67
conflicts
  See also errors; resolving conflicts
  base class 194
  copying definitions 146
  Existing vs. Multiple Definition 312
  item definition 22, 24, 26
  loading optional specifications 45
  overview 311–312
  package load process and 134
  SOM and Smalltalk class name 302
  specifying preferences for checking 67
  unloading modules 118
Connect Repository dialog 75
connecting to repositories 51
  repository interface components 279
  repositoryFromPath: message 288
  tutorial 74–76
connections
  closing repository component 286
  as common to all repository components 284
  determining if possible to establish 279, 285
containers. See entities; revisions
conventions
  tools, text display in 103
converting
  clusters 204
  complete specifications to partial 207
  configurations 204
  groups 204
  minimizing a specification 206
Copy To... command
  Package Organizer Definition menu 146
  Specification menu 153
Copy... (Module menu) command 119
copying
  See also detaching from repositories
  definitions between packages 146
  drag and drop, using 104–105
  module revisions to repositories 119
repository information 51
  specifications between clusters 153
  crash, image. See image, recovering from crash of
create message 281
  entities 290
  operation performed by 285
  physical data for new revisions, creating 293
Create Repository dialog
described 140
  specifying repository directories 221
createRepositoryNamed:path:type: message 287
creating
  See also adding; building; defining
  ad hoc initializers 193
  annotation labels 149
  build scripts 236
  categories 82, 189
  class definitions 112
  class initializers 191–192
  cluster revisions 91–92
  clusters 200
  definitions 146–148
  entities 290
  initializers 148
  methods 189
  package revisions 88–89
  packages 96–98, 144, 185
repositories
  Create Repository dialog 140, 220–221
  createRepositoryNamed:path:type: message 287
  operations for 51
  repository components 279–280
  Repository, RepositoryEntity, Revision
class instances 281
  RepositorySystemPath class instances 281
  revisions 25, 293–294
  specifications 36, 37, 152
creation time stamp property 284
creationTimeStamp message 285
creator message 285
creators property, repository component 284
Customize... (Build Library dialog) button 169
D
data, Smalltalk objects representing hard disk. See also physical data 279
Debugger menu
  Messages command 161
declarations, temporary 327
declaring global and pool variables 33
Defaults (Preferences menu) command 69
defining
See also adding; building; creating
access to pool dictionaries 147
class access to pools 30
class initialization expressions 147
class initializers 30
class instance variables 28–30, 117
class variables 28, 117
clusters 115
instance variables 27–28, 116
methods 27, 128
modules 116–117
packages 115
pools and pool variables 31
variables 130
Definition (Global/New submenu) command 123
Definition Comparison Browsers
Compare menu 165
conflicting definitions opening 312
described 163–165
Left and Right menus 165
opening 163
Definition Group Browser
Definition menu 159–160
panes 158–159
unresolved variables, definitions referring to 171
Definition Group Browsers
opening 123
definition list
ad hoc initializer comments in 33
described 23
Definition menu
Definition Group Browser 159–160
Message Browser 161
Package Organizer 145
definitions
See also global definitions; initializers; list of definitions
ad hoc initializer 32–33
adding variables to 130
browsing
Definition Group Browser 158–160
Message Browser 160–161
changing. See revisions
of classes in packages 20–21
class-related 27–33
comparing 164
comparing method 95
comparing versions of source-based 215–216
conflicts between modules 311
copying between packages 146
deleting
from image 118, 159
from packages 145
determining which modules own 108
discarding changes to 25
dragging and dropping into other packages 187
filtering modified 180
formatting 146, 160
global variable 31
load failures due to conflicts among 24
loading 24
moving
between packages 144
in definition list 145
organizing 144–148
overview 25
packages and 18, 183
pool and pool variable 31
recomputing 120, 163
referencing unresolved variables 171
removing from Definition Group Browser list 160
reordering within packages 187–188
reverting to older versions of 216
SOM global object 307
storing file-out representations of 145
TeamVInterface, class 247–248
types of 26
versions of, types having 209
viewing
all referring to specific variable 131
annotations for 149
original versions and histories of 209
package revision 166
in selected packages 142
source code for 93
within packages 143, 144
Delete Category (Method menu) command 127
Delete command
Annotation menu 149
Definition Group Browser Definition menu 159
Global/New submenu 122
Method menu 126
Package Browser Variable menu 130
Package Organizer Definition menu 145
Specification menu 152
Variable menu 171
Index

Delete Unreferenced (Variable menu) command 171
DeletedClass, class 67
deleting
categories
- from image 127
- conditional inclusion flags 151, 156
definitions
- from image 118, 159
- from packages 145
globals from images 122
methods from image 126
modules from image 118
recovery log contents 319, 321
specifications from clusters 152
unresolved variables 171
variables from image 130
delivering applications
building
- from clean image 239
- runtime images for 243
  Smalltalk Link Libraries from modules 239–241
distributing code 237–238
Map Browser as Smalltalk Link Library 100
Description (Global/New submenu) command 123
destroy message 285
Detach (Team submenu) command 323
detaching from repositories
- before becoming inaccessible 323
- procedure for 228–229
development environment
- configuring 55–60
- setting conditional inclusion flags in 202
  system packages in 18
directives
- conflicts 60
- master configuration file settings 58–60
directories
EXTRAS/TOOSRC 245
repository
  shared file systems 51
  specifying new 295
  --WORK-- 324
disconnecting from repositories 51
distributing
- changes to applications
  exporting modules 119
  tutorial 100
code
- in delivering applications 237–238
DLLs (dynamic link libraries), SOM 307
Do It expressions, ignoring 320, 321
documentation conventions xv
Down (Definition/Move submenu) command 145
drag and drop
- adding
definitions to other packages 187
- specifications to clusters using 200
described 104–105
global definitions 124
- operations supporting 104–105
dynamic link libraries (DLLs), SOM 307
DynamicLinkLibrary, class 307
E
Edit Access 222
Edit... (Specification menu) command 153, 154
editing
See also changing; modifying;
  Specification Editors; entries for specific
  Specification Editor
class definitions 112, 114
  hierarchy superclass structure 123
packages without loading 229
specifications 153
loaded 155
ensure: message 293
entities
- accessing
  all in repository 290
  paths, using 290
  revisions of 291–292
creating 290
defined 278
  identifying 280–284
  instances of RepositoryEntity representing 279
  locking and unlocking 295
  operations performed on 284–286
  references to, obtaining 282–283
  repository system paths for non-existing 282
Entity menu 138
EntityRevision, class
  assignment to revision variable 292, 294
  creating physical data for revisions 295
  as repository interface class 280
  revisions and instances of 279
enums, proxies for SOM 301
environment variable, SOMIR 299
environment, development
  configuring 55–60
  setting conditional inclusion flags in 202
  system packages in 18
errors
  See also conflicts; image, recovering from crash of
  Conflict Notification dialog descriptions of 312
  recovering recover log changes 322
executing
  class initializers 192
  initialization expressions after migrating 134
  source code in revisions 48
execution, conditional 332
Existing Definition Conflicts 313
Existing Instances Conflicts 317
Existing Subclasses Conflicts 314
exists message
  operation performed by 285
  sending before open 286
  testing for existence of physical data 279
expanding
  cluster specifications 204
  definitions 144
Export... (Module menu) command 119
expressions
  in assignment statements 331
  binary 330
  keyword 330
  in return statements 331
  simple 329
  unary 329
EXTRASOOLSRC directory 245
EXTRASSOMREPOS subdirectory 307

F
fields
  module description 36
  module specification 203
File It In (Smalltalk menu) command 180, 181
File menu
  Install command 180, 181
  Save command
    method comments 84
    methods 83
File Out All... command
  Definition menu 160
File Out... command
  Definition Group Browser Definition menu 160
  Module menu 119
  Package Organizer Definition menu 145
File-Based repositories
  described 52
  vs. PVCS repositories 219
file-in process
  described 180
  ignoring Do It expressions during 320
  loading vs. 24
  of previously filed-out modules 119
file-ins
  loading
    as packages 133
    restructuring as packages 182
file-out process
  building runtime images 243
  definitions 145, 160
  described 119
files
  to copy for working off-site 229
  loading from repositories 133
  proxy map 305
  required for building baseline images 239
  tools source cluster, instructions for installing 245
filtering
  definitions
    in Definition Group Browser 160
    in Package Organizer 146, 153, 160, 168
    specification list contents 153, 168
Find Category... (Method menu) command 127
Find Class (or Global)... (Global menu) command 123
Find Cluster... (Cluster menu) command 151
Find Module... (Module menu) command 120
Find Package... (Package menu) command 144
Flatten hierarchy (Build Options menu) command
  described 304
  purpose 302
Format command
  Definition Group Browser Definition menu 160
  Method menu 127
  Package Organizer Definition menu 146
formatter, programmatic interface to method
  325—326
formatting
  cascaded messages 331—332
  comments 333—334
  conditionals and iterations 332
  definitions 146
  method source before comparisons 163, 165
  methods 127, 326—329
  source-based definitions 160
  statements 329—331
frameworks. See application frameworks
Global (Package Browser) menu 109

global definitions
  drag and drop, using 124
  lists of 106
  processing order 134
  SOM 307
  type mismatch conflicts 313–314
  viewing descriptions of 123

global list 106

Global menu
  Definition command 96
  described 121
  New -> Class command 116
  viewing class definitions 79

Global Variable (Definition/New submenu) command 147

Global Variable Editors, Package Browser 110–111

global variables
  See also variables
  ad hoc initializers and 193
  created by ad hoc initializers 33
  creating definitions for 147
  defining 31
  naming 122
  unbound names declared as 53
  unresolved 171–172

globals
  conflicts caused by undeclared 314
  deleting from images 122

groups
  described 38
  loading with configurations 42–43
  recording module relationships 41

H

hiding packages in clusters 80

hierarchical categories
  creating 189
  described 125

hierarchies. See class hierarchies

Hierarchy (Global/New submenu) command 123

History Browsers
  opening 109
  panes 157
  reverting to previous versions
    procedure 210–211
  Version menu 158
  viewing previous versions of
    definitions 27
    methods 85–87

I

identifying repositories, entities, and revisions 280–284

Idiom Tracking application
  as cluster of packages 20
  groups and configurations 42
  ifCurtailed: message 293, 294
  ifFalse: message 332
  ifFalse:ifTrue: message 332
  ifNone: block 287
  ifTrue: message 332
  ifTrue:ifFalse: message 332

image
  adding new packages to 144
  building
    applications from clean 239
    runtime 243
  changing global names in 230
  deleting
    categories from 127
    definitions from 159
    methods from 126
    modules from 118
    variables from 130
  rebuilding 236–237
  recovering changes to 321
  recovering from crash of 319–322
  viewing changes to 321

implementor box
  described 107
  modifying methods 129

Implementors command
  Definition menu 160
  Method menu 127

implementors, browsing
  for definitions 146
  opening Definition Group browsers 127

Include inherited methods (Class menu) command 306

Inclusion Context Editor 156–157

inclusion phase, cluster loading 40

Indexable Type Conflicts 316

inheritance
  class instance vs. class variable 28–29
  instance variable names 27
  proxy classes from SOMProxyClass 302
  purpose of 190
  subclass functionality from abstract classes 183

inheritance box 106–108

initialization
  changing order of definition 187–188
circularities and 232
instance variable 27
package 192
Initialization (Global/New submenu) command 123
initialization expressions defining class 147
inglobal, viewing 123
pool variables 124
Initialize command
Global menu 123
Module menu 120
Variable menu 131
initialize method, class initializers vs. 30, 191
Initializer (Definition/New submenu) command 148
initializers
See also ad hoc initializers; definitions defining class 147
overview 26
package loading and 24
processing order of 25, 120, 134
initializing
class instance variables 29–30
class variables 28
classes 190–192
globals, selected 123
independently of loading 135
packages 24, 120
pool variables 131
repositoryServices variable 287
Inspect command
Global menu 123
Package Browser Variable menu 131
Variable menu 171
Inspect name registry (Build Options menu) command 305
Install (File menu) command 180, 181
installing vs. loading file-ins 180–181
Install (Version Comparison Browser Right and Left menus) command 162
Install (Version menu) command 158
installing
vs. loading file-ins 180–181
methods as currently executable version 162
SOM Tools 297–298
tools source cluster 245
instance method references to instance variables 28
Instance Variable... (Definition/New submenu) command 147
instance variables
See also class instance variables; class variables creating 147
defining 27–28, 116
class variables
creating 147
defining 27–28, 116
encapsulating indexable structures using 316
integrating revisions and packages 235
Integrator, Team 177–178
interface cluster 245
italicized text
displayed in Package Browser 83
indicating modifications 208
tools, displayed in 103
italics
in manual xvi
iterations, formatting 332
K
keyword expressions 330
keyword messages 326
keywords 330
L
labels, annotation 149
Latitude, class 19
Left menu
Definition Comparison Browser 165
Version Comparison Browser 162
libraries. See base class library; Smalltalk Link Libraries
Library Build Dialog, opening 119
Linguistics package 19
literals, simple expressions as 329
Load command
Cluster menu 132
Module menu 132
Package Organizer Package menu 132, 144
Load dialog 132–133
Load mapping file... (Build Options menu) command 305
load phase, cluster loading 40
Load... command
Module menu 118
Preferences menu 69
loading
See also conflicts; file-in process clusters phases of 40
specification processing 38
Index

steps performed during 135
tutorial 76–80
installing file-ins vs. 180–181
module revisions 132–133
modules
   conflicts during 311
new revisions 100
packages 24
   in Package Organizers 144
   forward references and 193
   initialization during 192
   item definition conflicts 26
to resolve circularities 232–234
steps performed when 134
proxy maps from file 305
SOM Tools from clusters 298
Local Implementors command
   Definition Group Browser Definition menu 160
   Method menu 127
Local Senders command
   Definition Group Browser Definition menu 160
   Method menu 127
Lock (Module menu) command 120
lock message 295
locking
   See also optimistic versioning; pessimistic versioning
packages 225
   Lock (Module menu) command 120
repositories, selecting policy for 140
revisions 295
login names 56
Longitude, class 19
loose methods, conflicts created by 230

M
Map Browser
   Configuration Cluster
      building .SLL from 169
      creating revision of 91–92
      loading 76–80
      viewing specifications in 90
   overview 73
   removing class from 96–98
   updating Configuration Cluster 98–99
   viewing specifications for 80
mapping
   SOM names to Smalltalk class names 302
   SOM to Smalltalk proxy classes 308–309
   specifications to modules 40
master configuration files
   setting up 58–60
matches, repository system 283–284
memory ownership policy 299
menus, additions to Smalltalk 103
Message Browsers
   described 160–161
   opening 127, 160
   overview 101
messages
   See also methods
      browsing
         for definitions 146
         methods, all sent by selected 160–161
         senders and implementors of 127, 160
cascaded 331–332
class collaboration and 184
   conditional execution, formatting 332
   creating RepositorySystemPath instances 281
   repository component operations 285
Messages command
   Debugger menu 161
   Definition Group Browser Definition menu 160
   Method menu 127
message-sends 331–332
method categories
   using 188
   method comments, formatting 327, 333
   Method Definition dialog 126
   method definitions
      browsing all senders of 160
      creating 146
   method headers 326
   method list 106
   Method menu 124
   method templates
      in Class Editors 112
      creating methods in 189
   Method... (Definition Organizer Definition/New submenu) command 146
methods
   See also categories; class methods;
      instance methods; messages; method definitions
      added to classes outside of packages 185
      adding 81–84
      browsing all messages sent by 160–161
      changing
         names of 231
         receivers of 84
      comparing versions of 86–87, 162, 215–216
Index

creating 189

defining 27, 128

formatting 127

formatting source before computing comparisons 163, 165

installing as currently executable version 162

modifying 128–129

moving

between categories 127, 128 into new categories 189

parts of 326–327

restoring previous versions of 86

reverting to previous versions of 210

SOM classes

displaying 306

implemented by 299

proxies 304

SOM core system class 308

SOM Proxy Core class enhancements 307–308

SOM Proxy Core Skeleton package classes, defining 308

viewing

in categories 106

as inherited from superclasses 106–108

previous versions of 85

source code 112

Migrate dialog 133–134

Migrate... (Module menu) command 119

migrating

clusters 39

to different revisions of packages 25, 100, 133–134

minimal specifications

contents of 35

Minimize (Specification menu) command 153

Minimize All (Specification menu) command 153

Modified command

Specification/Show submenu 153, 168

modifying

base classes 195

clusters and specifications 43–44

methods 128–129

modules 211

packages 81

SOM class to Smalltalk class mapping 305

Module / Load... command 76

module definition editor 115

module description fields

described 36

Module Editor, Package Browser 109

module list 77, 105

Module menu

Build Library command 100, 169

Commit... command 120

Committ command 88

Committ... command 97, 98

Compare command 92

described 117–120

Load command 100, 132

Load... command 181

Migrate command 100

New -> Package command 97

New -> Package... command 185

New submenu 115

modules

building Smalltalk Libraries from 239–241

committing 119

conditional loading. See conditional inclusion flags; conditional specifications configurations vs. groups and 38

creating Smalltalk Link Libraries from 119, 169–170

defining 115

exporting 119

filing out 119

identifying revisions of 213

loading

revisions of 132–133

locating with browsers 102

locking 120, 225

modifying 211

naming 117

opening 93

recommending without requiring 37

recording relationships between 41

renaming 117

replacing currently selected 119

revisions

committing 211

opening 132

reverting to other 218

specifying preferred 44

viewing 213

in "top level" cluster 39

unloading 118

Move (Package Organizer) submenu commands 145

Move To... command

Package Organizer Definition/Move submenu 146

Specification menu 153

moving

classes to new packages 97–98
definitions
between packages 144
in definition list 145
drag and drop, using 104–105
methods between categories
existing methods 189
Recategorize... command 128
Rename Category... command 127
module specifications to **top level**
category 39
repositories 295
specifications to top level cluster 153
Multiple Definition Conflicts 313
Multiple Specification Conflicts 318

N
name field, module specification 203
name message 285
names
See also conflicts
category 35
definition 280
mapping SOM names to Smalltalk class
302, 305
module 37
pool and pool variable 32
repository 56, 51, 140
resolving conflicts by changing 230
sorting SOM class 306
undefined 53
naming
global variables 122
modules 117
pools 122
SLL files 169

networks, access control to repositories on 52
New Category... (Method menu) command 127

New command
Global menu 122
Module menu 117
Package Organizer Definition menu 145, 146
new method template 82–83

New... command
Annotations menu 149
Method menu 126
Package Browser Variable menu 130
Package Organizer Package menu 144
Specification menu 152, 154
Variable menu 171

Next (Package Comparison Browser) button 167

O
object-indexable classes 316
Open command
Definition Comparison Browser Smalltalk menu 163
Open dialog 131
open message
described 286
operation performed by 285
repository interface components 279
open revisions 48
opening
Build Library dialog 169
Cluster Organizers 80, 90, 150–151
clusters 38
Definition Comparison Browsers 163
Definition Group Browsers 123
inspectors on values for globals 123
Library Build Dialog 119
Library Build Script workspace 169
Message Browsers 127, 160
Method Definition dialog 126
module revisions 93, 131
Package Browsers 74, 92, 105
Package Commit Dialog 120
Package Organizers 141
packages 23, 215
Recovery Log Workspaces 320
repositories 288–289
Repositories dialogs 75
SOM Class Browsers 299
SOM Proxy Map inspector 305
SOM Type Definition Browsers 300
SOM Type Definition Browsers 300
Specification Editors 154
unresolved variables, inspectors on 171
Version Comparison Browsers 127

operating systems, access control and 221
optimistic versioning 225
Optional (Specification menu) command 153
optional inclusion flags
category load process and 135
described 199
determining if turned on 203
purpose of 37

optional specifications
configurations and 44
conflicts when loading 45
designating specifications as 201
overview 44

Options menu Preferences command 63
OS/2
  See also platforms; SOM (System Object Model) tools
overlapping
definitions, detection of 22
of specifications in clusters 34

P
Package (Module/New submenu) command 185
Package Browser
category order 64
Package Browsers
  adding code defined in other packages 187
  bold-faced text in 104
categories 124
Contents pane 109–114
defining packages or clusters in 115
Load dialog in, opening 132
opening 74, 92, 105
ordering categories in 64
overview 105
panes 105–108
as programming aid 186
regular text in 104
underlined text in 103–104
viewing class definitions in 96
Package Comparison Browser
  bold-faced text in 104
buttons 167
  green text in 103
panes 166–167
underlined text in 104
package list, Package Organizer 142
Package menu
described 143–144
Load command 132
Load... command 181
package names 22
Package Organizer
  adding code defined in other packages 187
  annotating packages 149
  creating initializers 148
  opening Load dialog in 132
  organizing definitions 144
  Package menu 143
  panes and lists in 141
  purpose of 141
Package Organizer (Smalltalk/Open submenu)
  command 141
package revisions. See also revisions 48
package temporary variable 290
packages
  See also revisions
    adding
categories and methods to 81–84
code to 187
annotating 149
building Smalltalk classes into unloaded 303
characteristics of 22–23
closing 25, 144
commenting methods in 84
committing to repositories 25, 121
comparing
  Compare (Module menu) command 120
  Package Comparison Browser 166
  procedure 92–95
  revisions of 216
creating
  Module/New command 144
  new revisions of 88–89
  procedure 185
tutorial 96–98
defining 115
definitions in 18
deleting definitions from 145
designated owners of 177
determining which specify cluster
  specifications 40
dividing applications into 182–185
dividing work among team members 21–22
editing without loading 229
for generated proxy classes, specifying 305
holding cooperative and related classes
  20–21
inaccessible repositories and unloading 323
inclusion in multiple clusters 34
initializing 120
integrating with revisions 235
loading 24
  forward references and 193
  multiple to resolve circularities 232–234
  specifying repositories and revisions
    of 118
  steps performed during 134
locating open vs. loaded 102
method versions, working with previous
  85–87
migrating to different revisions of 25, 100
Index

moving
categories between 128
definitions between 144, 146
opening 23
operations on 23–25
Package Organizer list of 142
referenced proxy and typedef classes,
checking for 303
reorganizing 185–186
restructuring
class hierarchies with 190
file-ins as 182
reusing code from other 232
simple 20
SOM implementation 307
supporting SOM browsers and functions
308
system 194
units of functionality 18–20
unloading 25, 135
viewing
annotations for 149
classes defined in 79
definitions within 142, 143, 144
older revisions of 214–215
parameters, SOM method 299
parent revision numbers 49
parenthesized text 104
partial specifications
See also complete specifications;
specifications
contents of 35
converting complete specifications to
206–207
pathnames, repository 140
paths
accessing repositories using 290
repository 51, 287
repository system
contents 284
creating 281–282
parts of 280
resolving 282–283
period (.)
in annotations 23
cluster annotations 35
in revision numbers 49
statements, separating 328
pessimistic versioning 226–227
physical data
checking for existence of
before opening repository components 279
exists message 285, 286
reopening closed repositories 288
connecting to Repository interface 279
creating
repository 281
for revisions 293–294
determining existence of 279, 285
proposed revisions stored as 293
testing for existence of 286
platforms
repository independence of 219
pointer types, proxies for SOM 301
Pool command
Definition/New submenu 147
Global/New submenu 122
pool dictionaries
creating definitions for 147
defining access to 147
Pool Editors, Package Browser 111
Pool Reference Conflicts 317
Pool Usage (Definition/New submenu)
command 147
Pool Variable (Definition/New submenu)
command 147
pool variables
adding to pools 130
defining 31, 147
defining initialization expressions for 111
initializing 131
initialization expression execution 124
resolution of references to 24, 134
pools
ad hoc initializers and 193
class definitions referring to undefined
317
classes using same 315
defining
before referencing 117
class access to 30
described 31
naming 122
preference files
saving preferences to 69
preferences
conflict notification 311
Preferences (Options menu) command 63
Preformat command
Definition Comparison Browser Compare
menu 165
Version Comparison Browser Compare
menu 163
Previous (Package Definition Browser) button
167
primitive specifications 327
problem solving
  conflict resolution 311–319
  image crash recovery 319–322
  repositories, working with inaccessible 323–324
processing order
  ad hoc initializer 33
  definition 134
  global definition 24
  initializer 23
programmatic interface, component mapping to 247–248
programmers. See teams.
Prompt for class mappings (Build Options menu) command 303, 305
proposed revisions
  described 280
  revision numbers of 293
proxies
  building automatically for referenced SOM classes 303
  defined 301
  for SOM typedefs 301
proxy classes
  package of classes building 308
  special SOM 308
  superclass class definitions for 307
proxy map files 305, 306
PVCS Configuration
  local 60
  PVCS configuration
    master configuration 58
PVCS repositories
  described 52
  vs. File-Based 219
PVCS VCONFIG.EXE utility
  setting up master configuration files 58–60
PVCS Version Manager
  configuring access control 61
  conflicts 60
  purpose of 55
R
reading revisions 292–293
ReadStream, class 279
rebuilding images 236–237
Recategorize... (Method menu) command 128
recomputing
  definitions 120
  unresolved references 171
  versions 158
  reconnecting to repositories 51
recording
  current versions of package definitions 25
  image saves 321
  package reuse and role in applications 22
  relationships between modules 41
  revisions using configurations 41
  updated specifications 47
Recover / Reset Log menu command 319
Recover menu 320–321
RECOVER.LOG file
  building baseline images 239
  vs. CHANGE.LOG file 319
RECOVER.LOG file. See also recovery logs
Recovery Log Workspace
  opening 320
  Recover menu 320–321
  recovering changes using 321
recovery logs
  described 319
  format of 321
  versions of definitions and 27
red text 103
references
  See also unresolved references
circularities and forward 232
deleting from class to pool 130
recomputing unresolved 171
remaining after pool removal 317
reopening closed repositories 288
repository, entity, revision 282–283
resolving 134
searching for specific 123
to SOM classes and typedefs 303
  to undefined globals 171
unresolved 53
regular text in tools 104
relocating repositories 295
Remove From List (Definition menu) command 160
removing
  classes from packages 97–98
  definitions from Definition Group browser list 160
  specifications from browsing tools 39
Rename Category... (Method menu) command 127
Rename... (Package Browser Variable menu) command 130
Rename... command
  Annotations menu 149
  Definition menu 145
  Global/New submenu 122
  Module menu 117
Index

renaming
  annotations 149
  globals 122
  modules 117
  variables 130, 145
reordering definitions 187–188
reorganizing packages 185–186
replaying recovery log changes 321
repositories
  access control 52, 221–225
  accessing open 289
  backing up 221
  class RepositoryServices and 280
  closing 288–289
  committing
    modules to 119
    packages to 25, 121
creating
  Create Repository dialog 141
  procedure 220–221
  repository services, using 287
detaching from 228–229
determining what kind to use 219
distributing 238
identifying 280–284
inaccessible
  detaching from 323
  status line indication 323
--WORK-- directory 323
instances of Repository representing 279
loading module revisions from 132–135
locking 225
opening 288–289
opening module revisions in 132
operations on 51
operations performed on 284–286
optimistic versioning 225
overview 51
pessimistic versioning 226–227
references to, obtaining 282–283
relocating 295
setting default type 69
sharing code without access to common 227
types of 52
working without 324
writing custom tools for 246
Repositories dialog 220
  opening 75
Repository Access dialogs 135
repository components
  accessing 286
  container 285
  creating 279–280
defined 278
  matching multiple 283–284
  properties 284
repository entities
  access control 224
repository entity 138
repository field, module specification 203
repository interface
  operations performed using 279
  overview 245
repository management dialogs 135, 136
Repository Preference Editor 69
repository system paths
  messages creating 281–282
  parts of 280
  resolving 282–283
repository types, setting 287
Repository, class
  instances of
    creating 281
    iterating through 289
    repositories and 279
    returning 288
  as repository interface class 280
RepositoryEntity, class
  accessing repositories using classes 290
  creating
    entities 290
    instances of 281
  as repository interface class 280
  entities and instances of 279
repositoryFromPath: message 288, 289
RepositoryRevisionNumber, class
  retrieving instances of 291
  revision numbers and 280
repositoryServices variable 287
RepositoryServices, class
  accessing instances of 286
  as repository interface class 280
RepositorySystemPath, class
  creating instances of 281–282
  identifying entities or revisions 280
  Reset Log (Recover menu) command 319, 321
resolution
  described 40
resolve message
  creating Repository, RepositoryEntity,
    EntityRevision instances 282–283
  entity revision, accessing single 292
resolveMatching message
  accessing all entries in repository 290
  wild-card resolutions 283–284
resolving conflicts
  See also conflicts
described 230
editing packages without loading 229
Existing and Multiple Definition conflicts 313
Existing Instances Conflicts 317
Existing Subclasses Undeclared Superclass
  Conflicts 314
Indexable Type Conflicts 316
Multiple Specification Conflicts 318
overview 312
Pool Reference Conflicts 317
Type Mismatch 313–314
Undeclared Global Conflicts 314
Undeclared Pool Conflicts 317
Undeclared Superclass Conflicts 314
Unresolved Specification Conflicts 318–319
Variable Name Conflicts 315–316
restoring previous versions of methods 86
return statements 331
reusability
  of code from other packages 232
  component 184
dividing packages for 21
revision control 47
revision field, module specification 203
revision numbers
defined 37
  identifying module revisions and 213
  incrementing 88, 119, 121
  proposed revisions 293
RepositoryRevisionNumber class instances
  representing 280
  restrictions to 49–50
retrieving entity 291
as revision attributes 50
revision variable 292, 294
Revision, class 281
.RevisionNumber annotation 213
revisionNumberList message 291
revisionNumbers message 291
revisions
  See also entities
accessing and committing 279
accessing entity 291–292
attributes of 50
committing
  clusters 39
  module 211
  packages 25
comparing package 216–218
copying module, to repositories 119
creating 25, 88–89, 293–294
described 48–50
examining older package 214–215
identifying 213, 280–284
instances of EntityRevision representing
  279
integrating with packages 235
loading new 100
locking and unlocking 295
migrating to
  different 119, 133–134
new 25, 100
opening 93
operations performed on 284–286
overview 23
reading 292–293
recording, using configurations 41
references to, obtaining 282–283
reverting to
  different module 218
  older definitions 216
  specifying comment strings 285
tracking 207–211
version histories 208–211
viewing
  list of 77
  module 213
  package 79, 166
Right menu
  Definition Comparison Browser 165
  Version Comparison Browser 162
S
Save mapping file... (Build Options menu) command 305
Save... (Preferences menu) command 69
saving
changes to modules to disk 211
packages. See committing packages
preferences settings 69
proxy maps to file 305
variables and comments 117
searching
for category containing selected method 127
for specific references 123
with wild cards 120
semantic interface
  component mapping to programmatic
  interface 247–248
described 245
Semaphore directive 59
Semaphore Dir directive 59
Index

semaphores
See also Semaphore directive;
SemaphoreDir directive
consistency of settings for multiple, for same archive 60
semicolons (;) 332
Senders command
  Definition Group Browser Definition menu 160
  Method menu 127
senders, browsing for in Definition Group Browser 160
in Package Organizer 146
sequence types, proxies for SOM 301
Service Registry, Smalltalk/V 286
Set package... (Build Options menu) command 305
setting
  conditional inclusion flags 202
  package for generated proxy classes 305
  repository types 287
  revision contents 293, 294
  user names 56
  VCSID in VCS.CFG file 60
sharing code without common repository
  access 227
Show -> Modified (Definition Group Browser Definition menu) command 160
Show only defined methods (SOM Class Browser Class menu) command 306
simple expressions 329
Skip Do Its (Recover menu) command 320
.SLL files 100
Smalltalk libraries
  building from modules 239–241
  installing SOM tools from 297–298
Smalltalk Link Libraries
  building 169–170
  delivering Map Browser as 100
  opening Library Build dialog 119
Smalltalk menu
  additions to base 103
  Browse -> Unresolved 53
  Browse Packages 74
  Browse Packages command 92, 105
  Do It command 87
  Open -> Cluster Organizer 80
  Open -> Cluster Organizer command 90
  Open -> Definition Comparison Browser 163
Smalltalk/V for platform package 18
Smalltalk/V Service Registry 286
SOM api method category 304
SOM api wrappers method category 304
SOM Base Cluster 307–308
SOM Bootstrap Code package 308
SOM Browsers package 308
SOM Build Manager package 308
SOM Class Browser
  Class menu 306
  described 299–300
  purpose 297, 298
SOM Class Builder package 308
SOM classes
  building proxy classes for 306
  mapping
    names to Smalltalk class names 302, 305
    Proxy Map 308–309
  proxy methods for 304
  referenced, automatically building 303
  viewing all, in SOM repositories 299
SOM Control package 307
SOM Dlls package 307
SOM implementation packages
  overview 307
  SOM Base Cluster 307–308
  SOM Proxy Map 308–309
  SOM Tool Cluster 308
SOM Interface Repositories
  defining which can be browsed 299
overview 297
SOM Proxy Base Classes package 307
SOM Proxy Core Skeleton package
  clamp package lists 304
  described 307
  method for defining classes in 308
SOM Proxy Enhancements package 307
SOM System Additions package 308
SOM Tools
  browsers, See also SOM Class Browser;
  SOM Typedef Browser 298
building Smalltalk proxy classes 301–304
definitions of classes supporting 307
installing
  loading from clusters 298
  methods compared 297
  from Smalltalk libraries 297–298
SOM Class Browser 299–300
  Type Definition Browser, SOM 300–301
SOM Type Definition Browser
described 300–301
  TypeDef menu 306
SOM type definitions, viewing 300
SOM Typedef Browser 297, 298
SOM Typedef Builders package 308
SOMArray, class 301
SOMClassBuildSpecification, class 309
SOMControlManager, class 307
SOMEnumeration, class 301
SOMIR environment variable 299
SOMPointer, class 301
SOMProxyClass, class
  inheritance of proxy classes from 302
SOMProxyClassBuilder, class 308
SOMProxyTypeDef, class 307
SOMSequence, class 301
SOMStructure, class 301
SOMUnion, class 301
Sort alphabetical (SOM Class Browser Class menu) command 306
Sort hierarchical (SOM Class Browser Class menu) command 306
source code
  adding to packages 187
  browsing cluster 79
  in committed revisions 48
  creating ad hoc initializers containing 148
  reusing, from other packages 232
  sharing without common repository access 227
  tool and browser 245
  viewing
    definition 93, 144
    for packages or definitions 143
    inherited, from superclasses 82
    method 112
Specification Editors
  creating specifications using 37
  editing loaded specifications 155
  opening 154
  overview 102
  purpose of 154
Specification menu
  described 152
specifications
See also Cluster Organizer; clusters;
  complete specifications; partial specifications
clusters
  adding to 36, 200–201
  as collections of 35
  completing 153
  creating open 152
  deleting from clusters 152
  described 35–37
  designating as optional 201
  editing
    loaded 155
    expanding and contracting 204
    interpreting module 203
    minimizing 155
  modifying 43–44
  multiple contradictory 318
  optional 44–45
  overlapping in clusters 34
  primitive 327
  removing from browsing tools 39
  resolution of 40, 42–43
  resolution to specific packages 135
  specifying as optional 153
  in "top level" cluster 39
  tracking
    described 46
    tutorial 80–91
  unresolved 318–319
  viewing cluster 80–81, 150
square brackets ([ ])
  in blocks 328
statements
  assignment 331
  expressions 329–331
  format in code body 328
  return 331
  types of 329
structs, proxies for SOM 301
subclasses
  of abstract classes 183
  conflicts caused when deleting superclasses of 314
  viewing class 113
superclasses
  conflicts caused by undefined 314
  resolution of references to 24, 134
  viewing
    in hierarchy 113
    methods inherited from 106–108
system packages
  constantly present in development environment 18
  restrictions 194
T
  teams
    advantages of revision control to 47
    developing applications using 176–178
    dividing work among members 21–22
TeamVInterface, class 247–248
  teamwork
    detaching from repositories 228–229
    editing packages without loading 229
    extending classes defined elsewhere 234
    integrating revisions and packages 235
    loading packages
      multiple 232–234
      rebuilding image 236–237
resolving conflicts between packages
reusing code from other packages
sharing code without common repository
access
temporary declarations
text
 displayed in tools
The 66
time stamps
 repository component creation
as revision attributes

toby:do: message, formatting

to:do: message, formatting

tokens

tool box class

tools

See also SOM (System Object Model) tools;
entries for specific tools
drag and drop operations
overview
source cluster
specifying colors for
TeamVInterface

text display conventions
writing custom

Top (Package Organizer Definition/Move submenu) command

** top level ** cluster

described
moving specifications to

tracking
revisions
specifications
described

tutorial

See also Map Browser
tasks performed in 73–74
type definitions
generating Smalltalk classes for
package containing classes for SOM
proxies for SOM
viewing SOM

type field, module specification

type mismatch conflicts

TypeDef menu (SOM Type Definition Browser)

U

unary expressions

unary message selectors
in method headers
in unary expressions

unbound names, compiler response to

" unclassified " category

searching for specific categories

Uncluster (Specification menu) command

Undeclared Global Conflicts
Undeclared Pool Conflicts
Undeclared Superclass Conflict

underlined text

unions, proxies for SOM

Unload (Module menu) command

unloading
modules
packages

discarding changes to definitions by

inaccessible repositories

removal of definitions from image

unlock message

unlocking

modules
revisions

" unpackaged " package
globals created by ad hoc initializers
restrictions on

contents of 18

Unresolved References Inspector

Variable menu

unresolved references. See also Unresolved References Inspector

Unresolved Specification Conflicts

Up (Definition/Move submenu) command

Update command

Module menu
SOM Class Browser Class menu
Variable menu
Version Comparison Browser Compare menu

Version menu

updating

configuration clusters

Map Browser Configuration Cluster

Package Comparison Browser panes

SOM Class Browser class list

user identification

VCS.CFG file

VCSID variable

user names

master configuration file source settings

setting

V

V.EXE file

Variable (Global/New submenu) command
Variable menu
  Package Browser 129–131
  Unresolved References Inspector 171–172
Variable Name Conflicts 315–316
variable names
  in assignment statements 331
  simple expressions as 329
variables
  see also class instance variables; class variables; global variables; instance variables; pool variables
  defining 130
  deleting from image 130
  inspecting 131
  renaming 130, 145
  saving 117
  viewing all definitions referring to 131
Variables menu (class definition editor) 117
VCS.CFG file 60
  setting VCSID in 60
  user identification 56
VCSID directive 60
VCSID environment variable 56, 60
version button
  contents 56
Version Comparison Browser
  Compare menu 163
  Left and Right menus 162
  method definitions 146
  opening 127, 158
  panes 161
  version history 208–211
Version menu
  Compare All command 86
  described 158
  Install command 87
versions
  comparing 158
    method 160, 162
  of source-based definitions 215–216
  definition changes and 17
  definition types having 209
  of definitions 26–27
  identifying definition 109
  installing selected, as active in image 158
  recomputing 158
  recomputing definition 163
  restoring previous methods 86
  reverting to previous 210–211
  viewing
    previous method 85–87
View Entire Log (Recover menu) command 320
View Unsaved Changes (Recover menu) command 320
viewing
  annotations 142, 149
  available repositories list 98
  class definitions 79, 96
  class hierarchies 113, 123
  cluster annotations 153
  cluster specifications 80–81
  definitions
    original versions and histories of 209
    referring to specific variables 131
    within packages 144
  image changes in recovery logs 321
  initialization expressions for globals 123
  lists of modules and revisions 77
  methods
    in categories 106
    as inherited from superclasses 106–108
    previous versions of 85
  modules
    revisions of 213
    older package revisions 214–215
    organization of methods into categories 106
    package list in Package Organizer 142
    package revision definitions 166
    package revisions 166
    Recovery Log Workspace contents 320
    SOM classes 299
    SOM to Smalltalk class mappings 305
    source code
      inherited from superclasses 82
      specification annotations 154
W
wild cards
  category sort order 64
  category sorting preferences and 65
  repository system path matches 283–284
  searching for modules using 120
  variables, searches for 123
writeStream message 293
WriteStream, class
  accessing revision physical data 279
  setting revision contents 293, 294
Reader Comment Sheet

Name: ____________________________________________

Job title/function: ____________________________________________

Company name: ____________________________________________

Address: ____________________________________________

Telephone number: ( ) - Date: / / 

How often do you use this product?  Daily  Weekly  Monthly  Less

How long have you been using this product?  Months  Years

Can you find the information you need?  Yes  No

Please comment. ____________________________________________

Is the information easy to understand?  Yes  No

Please comment. ____________________________________________

Is the information adequate to perform your task?  Yes  No

Please comment. ____________________________________________

General comment: ____________________________________________

___________________________________________________________

WE STRIVE FOR QUALITY

To respond, please fax to Larry Fasse at (513) 612-2000.