Visual Smalltalk Enterprise™ ™

User's Guide

P46-0205-00
## Contents

### Preface
- Overview ................................................................. vii
- System Requirements ................................................ viii
- Getting Started ......................................................... viii
- About this Manual .................................................... ix
- Related Documentation ............................................. ix
- On-line Help and Documentation ................................. xi

### Chapter 1  Introduction to Visual Smalltalk
- Overview ........................................................................ 13
- Orienting Toward Objects ............................................. 14
- Objects ........................................................................... 15
  - Classes and Instances .............................................. 15
  - Methods and Messages .......................................... 16
  - Events ........................................................................ 17
  - Components ............................................................ 18
- The World of Visual Smalltalk ......................................... 19
- Smalltalk ....................................................................... 19
- Parts Workbench .......................................................... 21
- Visual Smalltalk Enterprise Extended Environment ...... 23

### Chapter 2  Programming in Visual Smalltalk
- Organizing Application Source Files .............................. 30
  - Organizing Code in Visual Smalltalk Enterprise .......... 30
  - Organizing Code in Visual Smalltalk .......................... 31
- The Transcript .................................................................... 32
- Service Manager .......................................................... 32
- The Workspace ............................................................ 34
- The Parts Workbench ........................................................ 34
  - Placing a Part on the Workbench ................................. 37
  - Creating Links .......................................................... 38
  - Saving a Part File ..................................................... 40
  - Launching a Part ....................................................... 40
  - Closing a Workbench ................................................. 41
- Package Browser .......................................................... 41
  - Parts of a Package Browser ....................................... 41
  - Working with Packages and Repositories .................. 43
  - Working with Classes and Methods ......................... 49
Class Hierarchy Browser .......................................................54
Parts of a Class Hierarchy Browser ........................................54
Working with Classes .......................................................56
Working With Methods ....................................................59
Method Editing Tools ..........................................................60
Method Browser ................................................................60
Message Browser ............................................................61
Inspectors ........................................................................63
Opening an Inspector .......................................................64
Editing Objects in an Inspector .........................................64
Using Special Inspectors ..................................................65
Debugging Tools ................................................................66
Walkback .........................................................................66
Debugger .........................................................................68
Saving Smalltalk Source Code ..........................................70
Saving the Smalltalk Image ...............................................70
Saving Packages ................................................................71
Smalltalk Link Libraries ...................................................71
File Out ...........................................................................72
Object Filer ......................................................................72

Chapter 3 Maintaining Visual Smalltalk
Overview ...........................................................................73
Saving the Image ...............................................................74
Recovering from an Image Crash ........................................75
Recovering in Visual Smalltalk ..........................................75
Recovering in Visual Smalltalk Enterprise ..........................77
Maintaining Log Files ........................................................79
Compressing the Change Log ...........................................79
Compressing the Recovery Log ........................................79
Managing Memory .............................................................80
Finalization .......................................................................80
Removing Instances ..........................................................80

Chapter 4 Building an Application
Overview ..........................................................................83
Designing an Object-Oriented Application ..........................83
Object Roles ......................................................................83
Component Architecture ...................................................85
Visual Smalltalk Application Framework .............................86
Building the Application View ..........................................88
Exposing the View Interface .............................................89
Contents

Form Generation ............................................................... 135
Adding a Create form parts... Button ......................... 136
Invoking Form Generation ........................................... 137
Customizing Form Generation ..................................... 139
Specifying Characteristics of Generated Parts ............ 141
Implementing partEditProperties ............................... 143

Glossary 145

Bibliography 165

Periodicals ................................................................. 166

Index 167
Preface

Welcome to Visual Smalltalk and Visual Smalltalk Enterprise, your complete object-oriented programming environment.

At the heart of Visual Smalltalk is the Smalltalk programming language. Smalltalk is the pure object-oriented programming language that has been revolutionizing enterprise-wide application development. Several key technological advantages, such as code reusability and component architecture, are becoming a reality based on the expressive power of Smalltalk. Visual Smalltalk provides a complete set of Smalltalk development tools, including class and code browsers, object inspectors, and workspaces for testing your work.

Visual Smalltalk includes the PARTS Workbench visual application assembly tool. Visual application assembly takes object-oriented programming to new and intuitive levels. It encourages thinking of an application program as an assembly of business objects, the processes and goods that you work with in your daily business. Application assembly becomes a simple matter of dropping the necessary objects onto a workbench and drawing links between them to describe the interactions.

Visual Smalltalk Enterprise extends Visual Smalltalk with an enhanced tool set that promotes modularized development and enables team development. By incorporating Intersolv’s popular PVCS version control system, Visual Smalltalk Enterprise helps you to distribute the work load among even large development teams and to coordinate the results. In the first chapter, we introduce the elementary syntax and semantics of the Smalltalk language. In the process, we will discuss the basic concepts behind object-oriented programming. This lays the foundation on which the remainder of this manual builds.
System Requirements

Visual Smalltalk requires the following minimum system resources:

- A personal computer with a 80386 or higher processor
- Microsoft Windows 3.1 or later, Windows 95, or Windows NT version 3.5.1 or later, or IBM OS/2 Warp
- Windows 3.1 must be running in 386 enhanced mode with virtual memory enabled, and Win32s support. (Win32s is provided with Visual Smalltalk.)
- A minimum of 16MB of RAM
- A minimum of 25MB free disk space for Visual Smalltalk, 50MB for Visual Smalltalk Enterprise
- A 3.5 inch, 1.44 MB floppy diskette drive or CD-ROM drive
- A mouse

The following resources are optional, but recommended:

- Additional RAM
- Math coprocessor

Getting Started

The first thing you need to do is to install Visual Smalltalk or Visual Smalltalk Enterprise. Installation instructions are contained on a separate card.

NOTE: For simplicity, we will generally refer to Visual Smalltalk and Visual Smalltalk Enterprise jointly as “Visual Smalltalk.” When we need to refer to one product but not the other, we will make the reference clear.

Once you have installed Visual Smalltalk, you can start the development environment either by clicking on the startup icon, or by executing VDEVVW.EXE on Windows systems or VDEVO.EXE on OS/2 systems.
About this Manual

The Visual Smalltalk User's Guide, the manual you are currently reading, is your gateway to the resources and tools provided in Visual Smalltalk and Visual Smalltalk Enterprise. The manual assumes no prior knowledge of Smalltalk or object-oriented programming, but does assume you will be referring to the related documents along with this one (refer to “Related Documentation” below).

Chapter 1 introduces the basic concepts of object-orientation and object-oriented programming. It then introduces more specific aspects of the Visual Smalltalk environment.

Chapter 2 introduces the windows, browsers and editors that you use to create Visual Smalltalk applications.

Chapter 3 explains some important issues about maintaining your Visual Smalltalk environment, including how to maintain source code for your application using the tools explained in chapter 2.

Chapter 4 discusses basic ideas behind object-oriented design, and how to implement a design using Visual Smalltalk. Applications consist of interacting objects, and this chapter gives some guidance in selecting and implementing appropriate objects.

Chapter 5 explains how to finish up an application and prepare it for delivery to a customer as a stand-alone runtime application.

Chapters 6 and 7 cover more advanced topics related to building new parts in Smalltalk to optimize use of the workbench for application assembly.

This manual does not talk about specifics of the Smalltalk language or give detailed instructions for using the PARTS Workbench. Refer to “Related Documentation” to find this information.

Related Documentation

Visual Smalltalk is a large topic, and it cannot be covered in one manual. Visual Smalltalk includes a complete library of documents to help you quickly and efficiently develop your applications.

Documents in addition to this programmer's guide are:

Getting Started
  This card tells you how to install and start running Visual Smalltalk.
Visual Smalltalk Enterprise Tutorial
An extended introduction to the Visual Smalltalk environment and tour of many of the key components of the system. The tutorial also walks you through building a complete application, the Contact Manager, from design through construction.

Visual Smalltalk Enterprise Language Reference
This reference manual discusses many of the classes in the Smalltalk language component of Visual Smalltalk, describing the class structure and how to use the class to solve programming problems. This book will help you master the Smalltalk language.

Visual Smalltalk Enterprise Encyclopedia of Classes
A summary of all public classes, methods, and events in Visual Smalltalk.

Visual Smalltalk Enterprise Workbench User’s Guide
In-depth explanation of the workbench, including instructions for constructing composite parts from existing parts. This is the book that will help you master the workbench.

Visual Smalltalk Enterprise Parts Reference
A reference discussion of every component in the workbench catalog.

Visual Smalltalk Enterprise Tools Reference
This manual explains in detail each tool in the extended Visual Smalltalk Enterprise programming environment, including such topics as code management and team programming. A chapter on the tool API also helps you to extend and customize the tool set to fit your special needs.

This manual describes how to build data base reports in the workbench using the Report Writer parts.

Additional resources are listed in the bibliography at the end of this manual.
On-line Help and Documentation

Visual Smalltalk and Visual Smalltalk Enterprise include on-line, context-sensitive help for menus and parts. While working, select any part or menu and press the F1 key.

The printed documentation is also available on-line while you are running Visual Smalltalk or Visual Smalltalk Enterprise. From the help menu or any help screen, select Library to view a list of documents.

The Encyclopedia of Classes is also on line. In any workspace or text pane, highlight the name of a class and press F1.
Environments for object-oriented programming have been developing since the early 1960’s. Visual Smalltalk and Visual Smalltalk Enterprise represent the latest, most powerful, and easiest to use environment available today.

In this chapter, we introduce the basic ideas behind object-oriented programming, as well as some more recent additions, such as using events to begin sequences of messages. If you are new to object-orientation, this chapter will help orient you to objects, introducing you to a new way of thinking about programming. If you are already familiar with these concepts, you may still benefit from at least browsing through the material.

Beyond the basics of object-orientation, Visual Smalltalk provides an integrated approach to thinking about objects and application programs. There are two basic perspectives you can take toward objects. On the one hand, you can think of objects in abstract terms, as instances of classes with certain specified behavior. This is the level where much application design work is done.

At the application assembly stage, on the other hand, it is useful to work with concrete objects, the instances of the classes. Visual Smalltalk and Visual Smalltalk Enterprise encourage you to think in these more concrete terms by representing objects visually, and allowing you to assemble an application out of these concrete objects on a workbench.

Accordingly, after we have discussed the general ideas of object-oriented programming, we describe how Visual Smalltalk and Visual Smalltalk Enterprise embody these ideas.
Orienting Toward Objects

We think and talk about the world as consisting of objects of all kinds. The world, for example, includes a variety of concrete objects, such as people, buildings, animals, trees, and desks. There are also more abstract objects, such as institutions, bank accounts, reports, numbers, collections of other objects, and on and on.

Objects have properties. An object has some properties by virtue of being the kind of object it is: balls are round, bank accounts store money, tables have legs, ice is cold. In addition to these general properties, individual objects have properties that make them individual. A person may be short or tall, fat or slim, have a social security number, and so on. A block has specific dimensions, weight, and color.

Some objects do things; they can perform tasks. So, birds can fly and calculators can add. Objects also interact with each other in a variety of ways. Money goes into and out of bank accounts. People join and leave organizations. Reports collect, calculate, and report data.

Visual Smalltalk is a programming environment for describing objects and the interactions between them. The end result of such a description is a Visual Smalltalk program. Depending on what we want the application program to do, the objects we describe may be quite concrete, highly abstract, or a combination of both.

As you learn to use Visual Smalltalk, keep this picture in mind. Think of your programming project as describing objects and what they do.
Objects

The problems that application programs are designed to solve are becoming increasingly complex. The problems involve complex interactions between data, business processes and rules, system users, customers, vendors, and so on. Over time, the interactions become more, not less, complex, and there are more factors, not fewer, involved.

Object-oriented programming is an approach towards managing this complexity by abstracting information about the business problem, and representing the business objects involved. The program that helps you manage your business is designed to explicitly represent the kinds of objects your business deals with.

The results are that the application is understandable in the same terms as your business. As the business expands, such as by adding another business unit, the application expands in the same way, by adding an object representing the new business unit.

In object-oriented programming, an application program is constructed as a collection of objects interacting with each other. By working at the higher level of abstraction represented by objects, the complexity is much easier to manage.

Accordingly, the most fundamental concept with which we deal in Smalltalk is the object. Objects are what we attribute properties to and describe interactions between.

An object can be anything, as suggested by the list of assorted objects given above, including Visual Smalltalk itself. As we continue, we will see just how pervasive this idea of objects really is.

Classes and Instances

When dealing with objects, we typically deal with individuals. We drive only one car at a time, not a class of cars. We can carry on a conversation with a few people at a time, not the class of people in general. Individuals are really what make up the world.

On the other hand, we very seldom think of an object merely as an individual. Usually, it is a kind of something; it is a car, or a person, or a desk, or a bird. Most of what we know about an individual object we know because the individual is just one of many instances of a class of objects. We know that an individual bird flies, not because we happen to know that each individual bird flies, but because we know that birds as a group (penguins notwithstanding) are the kind of things that fly.
In doing object-oriented programming, we work both with classes and with individuals, like two sides of the same coin. We seek to identify classes of objects and to describe behavior and relationships that are common to all instances of that class. If we do our job well, we can be sure that every member of a class has the properties we describe. The classes we identify are those whose instances we intend to use in building our application.

While all members of a class have many characteristics in common, each must also have something distinct from the others. This is what makes an individual unique. Accordingly, we need to provide a means of distinguishing individual instances of a class. Visual Smalltalk does this for us when we ask for a new instance of a class.

In Visual Smalltalk, classes are represented in two ways. In the Smalltalk language, a class is represented by a collection of textual descriptions of the class and the behavior of its instances. Some classes are further represented by an icon in the PARTS Workbench catalog. Both are essentially the same kind of thing; a source of new individuals.

Methods and Messages

Objects know how to do things, as we mentioned above. People know how to write letters, drive cars, and discuss national policy. Exercising a little more imagination, we can think of inanimate objects doing things. Pens “know” how to draw lines, and bank accounts “know” how to calculate interest and report their balance. Interactions between objects are also a matter of one object knowing how to do something at the request of another.
Knowing how to do something, in Visual Smalltalk, is accomplished by methods. Methods are essentially equivalent to the procedures written in procedural programming languages. They embody algorithms, the sequences of instructions required to carry out a task.

An object requests another object to do something by sending it a message. A common request, for example, is for the target object, or receiver of the message, to return some information to the sender.

A message has two parts: the name of the corresponding method and any required arguments. The name of the method is called the message selector. For a receiver to be able to respond to a message, it must define or inherit a method corresponding to the message.

When one object sends a message to another object, the sender is asking the receiver to perform the operation it specifies, and possibly to return some information as a result. When the receiver receives a message it recognizes, it performs the requested operation as described in its methods.

Objects can also trigger events. This is a newer addition to object-oriented programming that greatly increases object independence and interchangeability.

Events are a general mechanism that an object uses to signal that it has done something that might be of interest to other objects. Another object may need to perform some action, or start a sequence of actions, whenever such an event occurs. The simplest example is a button object in a visual screen. When the user positions the mouse pointer and clicks on the button object, the button will trigger a “clicked” event. Other objects may need to respond to the event, such as a window that responds by closing itself.

Events allow a looser relationship between objects than do messages. When an object sends a message to another object, it is demanding that the receiver respond in an appropriate way. If the receiver cannot respond, or doesn’t exist, an error occurs. On the other hand, an object can trigger an event without caring whether any other object is interested. The event is more of an announcement than a demand.

If an object is interested in an event triggered by another object, it registers an event handler with that object that says, “When you trigger such-and-such an event, do this action.” The action usually
consists of sending a message. The message must be defined for a specified receiver, but that is not the responsibility of the object triggering the event.

Events are used extensively in Visual Smalltalk, especially in the workbench. When you connect objects in the workbench, you draw links between objects, connecting an event triggered by the source object to a message sent to the target object. This approach is a good general model of object interaction, as is promoted in Visual Smalltalk.

**Components**

Another popular concept in object technology is the component. While the idea is fairly loose and broad, certain features of components are emerging.

A key idea is that a component is designed to be reused. A component encapsulates some important piece of programming logic that can be packaged and used across several applications. For example, if a spell checker were designed as a component it could be reused by any application needing such a feature. A programmer who needs to add spell checking to a new word processor or database product could simply “plug in” the component and use it by sending a few messages.

Components can vary greatly in size. They are not usually as small as calculating a single function, nor as large as an entire application, though they could be. Individual functions doing related work, on the other hand, especially if that work is required
by several applications, could make a good component. And a complete application, such as a word processor, might make a good component if it is designed to be pluggable into a suite of other applications.

An important feature of a component is that it has a well designed interface. This is necessary so that a programmer wanting to incorporate the component needs to know very little about its internal workings, but does know precisely what messages it will respond to and what kind of information to expect back from it. Care taken in designing the interface can either encourage or discourage its reuse.

Visual Smalltalk supports building components for use within a Visual Smalltalk application, as well as for using components built according to emerging standards. For example, on Windows platforms Visual Smalltalk can access components built according to Microsoft's COM standards, using OLE2 functionality. On OS/2 Visual Smalltalk can use and enhance objects built using according to the SOM standard.

In Visual Smalltalk you can also build components for distribution to and use by other Visual Smalltalk programmers. A component can be delivered as a Smalltalk library containing the Smalltalk code for a piece of functionality, or as a part built specifically for use in a workbench. Both strategies are used in a large number of Visual Smalltalk applications, and enhance the general picture of application assembly.

The World of Visual Smalltalk

Visual Smalltalk and Visual Smalltalk Enterprise are complete object-oriented programming environments. Both systems include Cincom's implementation of the Smalltalk programming language and the Parts Workbench object assembly tool. Visual Smalltalk Enterprise extends this environment with additional tools designed specifically to support large projects and group development.

Smalltalk

Smalltalk is the most popular pure object-oriented programming language, and the foundation of Visual Smalltalk. Smalltalk is a text-based language, using an easy to understand syntax with a rich vocabulary.
You use Smalltalk to define the basic objects used in your application by first creating a new class for each new kind of object. For instance, to create a customer object you would create a class called `Customer`. The class definition includes any variables required by the class and its instances, such as the instance variables that store properties for instances of the class. In a Class Hierarchy Browser, the definition might look like this:

```smalltalk
Object subclass: #Customer
instanceVariableNames:
  'custName custAddress custPhone custNumber'
classVariableNames: ''
poolDictionaries: ''
```

This expression specifies the new class to be a subclass of `Object`, and identifies four instance variables. There are no class variables or pools (pool dictionaries) identified.

For more information about class definitions and variable types, refer to the Visual Smalltalk Enterprise Language Reference.

Once you have created a class, you define methods to describe the behavior of the objects in the class and of the class itself. Methods correspond to both public and private messages that the class and its instances can respond to. In general, public messages are those that an object expects other objects to send to it, typically requesting information or action. Private messages, on the other hand, are messages that an object only expects to receive from itself or its constituent objects. Private messages can be thought of as representing the internal workings required to respond to a public message.

For example, most classes respond to a `new` message, implemented by a `class` method, that causes the class to create a new instance. Other class methods typically set class characteristics or define characteristics of an instance before creating it.

Instances of classes, on the other hand, respond to `instance` messages, which are implemented by `instance` methods. For example, each customer object should be able to respond to a message `custName` requesting the customer’s name. This message will simply return the value of the `custName` variable defined in the class. In a browser the definition will look like:

```smalltalk
custName
  "Return the value of custName"
^custName
```
This kind of a method, which gets property information, and methods that set properties, are referred to as accessor methods. Full instructions for writing methods are given in Visual Smalltalk Enterprise Language Reference.

**Parts Workbench**

In contrast to the Smalltalk language, which is a powerful text-based language for defining new objects, the Parts Workbench is a visual editor used to assemble existing objects into other objects.

The workbench allows you to construct bigger, more complex objects from smaller, simpler objects by dragging component objects to a workbench window and connecting them with links. The constituent objects can be dragged from a catalog of parts or created directly from Smalltalk classes and dropped on the workbench.

The workbench fully implements and gives a visual look to two of the basic concepts behind object-oriented programming: encapsulation and information hiding. When a component object is placed on the workbench, its own internal workings are hidden, with only its public interface exposed. The next higher level of application construction proceeds exclusively at the level of the public interfaces of the component objects involved at that level.
This aspect of object-oriented programming has great advantage in building complex applications. As long as the object’s interfaces remain unchanged, any changes made to the component objects, to add or improve its functionality, do not break functionality at the assembly level. Also, when assembling the application, the programmer is completely shielded from any underlying complexity, making it much easier to conceive and develop the higher level application.

There are two general categories of parts: visual and non-visual. Visual parts are the objects that you use to build a graphical user interface (GUI). The main visual part is the window part. When building a new visual interface for an application, you begin by dropping a window on the workbench, and then add to it other visual parts, such as buttons, display fields, and entry fields.

Non-visual parts represent “business objects,” non-visual objects representing data, processes, people, or anything else your business works with. Non-visual parts perform a variety of tasks, such as holding values, performing arithmetic operations, or accessing external data. They are dropped directly on the workbench, rather than in a window.

All parts trigger events and respond to messages. You design the interactions between parts by drawing links between one part’s event and another part’s message. For example, you might create a window that has, among other things, a button to close the window.

The workbench defining this much of the window looks like this:
The effect of drawing the link is that when a part triggers the event, it sends any messages connected to that event to the designated message receivers. In the above example, when the user clicks on the Close button, the part triggers a **clicked** event. That event, because of the link, sends the **close** message to the window.

**Visual Smalltalk Enterprise Extended Environment**

Visual Smalltalk Enterprise extends the power of Visual Smalltalk by adding tools and resources to help manage and structure your development projects. While the extensions are beneficial in any full-scale development environment, they are particularly valuable in team development environments.

The primary organizational structures are **packages** and **clusters**, jointly referred to as **modules**. Modules are used to organize your work by grouping related pieces of Smalltalk code.
Modules also give a convenient way to distribute and share work among development team members. Responsibility for individual packages or clusters can be assigned to individual team members, who store their work in shared repositories. The whole team has access to work in progress without having to share a single development image. When it is time to integrate the system, the modules are assembled and built into the running application.

Additional browsers and tools enhance the development environment. The central tool is the Package Browser, in which you define classes and methods. From the Package Browser you also access other tools to organize packages and clusters, review the history of your work, and so on. A set of revision control mechanisms further allows a team to manage and coordinate the process of change and growth.

**Packages and Clusters**

A package is a unit of Smalltalk source code, consisting of a set of definitions in a specified order. A package typically defines classes and their methods, global variables, pools and pool variables, and other objects. A package can also define individual methods (called loose methods) for classes defined in other packages.
A cluster consists of packages and other clusters, identified by specifications. Each specification describes a module in terms of:

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

Specifications may be either partial or complete:

- A **partial** specification includes at least the module name, but leaves some other information unspecified (for example, “the module named **Map Browser**”).
- A **complete** specification includes all specification information (for example, “revision 3.0 of the package named **Cartography** in the repository **Tutorial**”).

Specifications also have other properties, which are described in the **Visual Smalltalk Enterprise Tools Reference**.

Modules are saved in repositories. When you have made changes to a module, it is marked as “modified” and is listed in blue, italic type in the package browser. When you decide you want to save your changes, to make the changes permanent or available to others, you commit the package to its repository. The module is then saved to disk, assigned a new revision number, and the “modified” flag is removed.

The Visual Smalltalk Enterprise development image is itself contained in modules. Two modules are always present in your development environment:

- The **Visual Smalltalk Enterprise** package, which contains all the classes and definitions in the base image.
- The **unpackaged** package, which contains all the new definitions you have created outside of any other package, for example if you create new classes or methods using the Class Hierarchy Browser.

### Dividing Work Among Team Members

If large, complex applications are to be finished in a timely manner, the work must be done by a team. To work efficiently, each member must be responsible for one or more well-defined units. Visual Smalltalk Enterprise’s packages and clusters organize the individual’s work within a team.

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

Since all code is contained in packages, the application is completely contained in the collection of packages. When it comes time to integrate the application, each programmer contributes his or her packages, which are then combined to make the executable application.

If conflicts occur between packages, for example, if two programmers create a method with the same name in a class, these conflicts are reported when the packages 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.

**Initializers**

Visual Smalltalk Enterprise also adds the concept of an initializer. Since an image state is not recorded in packages, initializers perform operations on the code in a package necessary to prepare it to work.

An initializer is either a class initializer or an ad hoc initializer. A class initializer is an expression that initializes a class and sets initial values for class variables and class instance variables. An ad hoc initializer is an expression that initializes several global definitions whose initialization is not suitable to any single definition.

Initializers are evaluated when the package containing them is initialized.

**Repositories**

Repositories provide permanent storage for revisions of packages and clusters. Repositories are identified by a name in the Visual Smalltalk Enterprise environment, and exist on disk as a file system directory containing the data files that store the repository information.

You will normally use several different repositories in your work. Some may be private, containing your work in progress. Others are shared with team members, containing work that all team members must share. Shared repositories must be created in a
public directory on a shared file system. Private repositories can be created anywhere, such as on a local disk or a private network directory.

Visual Smalltalk Enterprise supports two kinds of repository:

- **PVCS repositories** store revisions of a module in a single file. PVCS repositories provide access control mechanisms to limit revision access to a package. For efficient disk usage, a package is recorded as a base version and a series of differences from the base.

- **File-based repositories** do not have access control mechanisms. Each revision is stored completely in its own file, so a full set of revisions uses more disk space than in a PVCS repository.

The type of repository is specified when it is created.

**Revision Control**

A critical need, especially in team development environments, is the ability to control the way revisions are entered into the common system. Visual Smalltalk Enterprise provides this control by using repositories.

Each time you commit a module that has been modified, a new revision is created for the module, with a unique revision number. Revision control is useful for individual programmers who may need to keep a history of their work. Revision control is even more useful in team programming. A revision control system is useful to managers, team leaders, and system integrators, allowing them to trace the history of a change. Other members of the team can consult previous revisions in order to determine when a change was made and who made the change.

When you open or load a module, you select one revision of that module. As you modify a module, those changes are local to your local Smalltalk image. To save those changes to a new revision of the module, you **commit** the module to its repository. The new revision stores the state of the module when it was committed.

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 you cannot change the contents of a specific revision once it is committed.

You can **open** a revision to examine and edit it without **loading** it into development image. If a revision is open, it is visible in the browsers, and can be edited and compared to other revisions, but it is not executable. To execute source in the revision, you must load it into the development environment.
The Visual Smalltalk environment provides a wide variety of tools for building large scale, object-oriented application programs. In this chapter we introduce the essential tools and show how to use them to perform basic programming tasks.

Visual Smalltalk Enterprise includes tools that are not available in Visual Smalltalk. For the most part, these differences revolve around the package/cluster structure available in Visual Smalltalk Enterprise. Where a difference affects the way you work in Visual Smalltalk, we will mention the fact. Many of the tools are not discussed in this section. Refer to the Visual Smalltalk Enterprise Tools Reference.

Since programming in Visual Smalltalk is essentially tied to the tools, we will take a tools approach in this chapter. Within the key tools, namely the workbench, package browser, and class hierarchy browser, we describe the basic operations.

We do not attempt here to describe every option in every tool. In particular, there are a number of menu options that we do not discuss. In the case of menu items, use the on-line help text to view a description by highlighting the item and pressing F1. The major topics we will cover are:

- Organizing Application Source Files
- The Transcript
- The Service Manager
- The Workspace
- The Parts Workbench
- Package Browser
- Class Hierarchy Browser
- Method Editing Tools
- Inspectors
- Debugging Tools
- Saving Smalltalk Source Code
Organizing Application Source Files

Before beginning to develop your application, you should look ahead to delivering that application. Some preparation for delivery should be made at the very beginning of the development process.

A Visual Smalltalk application consists of several interacting files:
- The Visual Smalltalk image file (V.EXE)
- Visual Smalltalk runtime libraries
- Application specific part files
- Application specific Smalltalk libraries

Of these, the hardest to deal with are the application specific Smalltalk libraries. These files contain the Smalltalk code, the class and method definitions, that specify most of your application logic.

To facilitate building your Smalltalk libraries, it is necessary to keep a record of your code, so that exactly the right code can be built into the Smalltalk library.

Organizing Code in Visual Smalltalk Enterprise

If you are using Visual Smalltalk Enterprise, this task is simplified by the fact that you already organize your application code into packages. The Build libraries... menu option in the Package Browser builds Smalltalk libraries automatically from the packages.

Before you begin, create a new package:

1. Open a Package Browser.
   - Press CTRL+P, or select the Smalltalk / Browse Packages menu item.

2. In the Package Browser, select Module / New / Package.
   - A dialog opens allowing you to enter a name for your package.

3. Enter a name for the package, then click OK.
   - Your package’s name is added to the top left pane of the Package Browser. It is shown in blue italics type until you commit it to a repository.
As you work, it is easy to create new packages, move class definitions between packages, and delete packages as needs change. This will be particularly true if you are working in a team, where pieces of the main programming task are divided up among several developers.

Some objects, such as global variables and class variables, require initializers to set the object’s values. Initializers are executed when the package is loaded into the image, ensuring that the object has the necessary values.

The packages you use during development might not be the same as the packages you ultimately use to build Smalltalk libraries. If delivery packaging is specified later in the project, some reorganization of your module structure may be necessary.

**Organizing Code in Visual Smalltalk**

If you are using Visual Smalltalk, without the Enterprise extensions, there are no packages to help you organize your work. Instead, you need to do a little more work to keep your code organized in preparation for building Smalltalk libraries. Briefly, you need to:

1. Keep a record of your code
2. Periodically build Smalltalk libraries containing your code.

The simplest approach for keeping track of your code is to open a new workspace and include a minimal Smalltalk library build script. To start, you can include the following lines:

```smalltalk
sll := SmalltalkLibraryBuilder new: 'myapp.sll'.
sll add: ; " include class names here "
add: ;
windowFeedback;
writeFile.
```

Each time you create a class, add an `add:` line with the class name. You can use as many `add:` lines as you need. If you redefine a method in one of the base classes, you must add it also. Make sure you save your script.

A full description of library building operations is provided in the [Visual Smalltalk Enterprise Language Reference](#).

You may want to divide a large project into more than one library, similar to the way a team using Visual Smalltalk Enterprise uses multiple packages. In this case, you can use several workspaces or use multiple scripts in a single workspace, each script building a different Smalltalk library. As you define classes and methods, simply add them to the appropriate script.
The Transcript

The Transcript is the Visual Smalltalk system window. When you start Visual Smalltalk, the Transcript is the first window to open, and it remains open as long as Visual Smalltalk is running. From the Transcript you can access all the other Visual Smalltalk development tools.

Other than a starting point, the main use for the Transcript is as a place to write system messages. For example, while binding Smalltalk libraries, Visual Smalltalk writes status and progress messages to the Transcript.

You can use it to display your own informative messages, which is often useful in debugging. To write to the Transcript, you can refer to it by its global name, Transcript. Using WriteStream protocol, you can write strings to the Transcript. For example, enter in a workspace:

```
Transcript show: 'This is a test'; cr
```

and evaluate it a few times using Do It. You can often test the output of a part of your application by sending it as a string in this way.

For information about WriteStream protocol, refer to the Visual Smalltalk Enterprise Language Reference.

Service Manager

Visual Smalltalk includes a large collection of components in addition to the standard, or base components. The Service Manager makes it easy to install and uninstall these optional components as needed.

The optional components are referred to as services, suggesting that they provide additional functionality. There are several types of functionality supported by the Service Manager, including additional tools, classes providing support for GUI controls, examples, and so on.

To open the Service Manager, select Smalltalk / Browser Services in a development system window (Developer / Browser Services in a workbench). The Service manager opens displaying a list of available services and a list of services already installed, if any.
To install a service, select the name of the service in the Not Installed list, then click the Install>> button. Since many of the services have dependencies on other services as well, the Service Manager may install more services than you selected.

To uninstall a service, select the name of the service in the Installed list and click <<Uninstall. If the selected service is required by some other installed service, and information dialog is displayed. Click OK to close the dialog. Also, if a service is installed because of a dependency by a service you have chosen to uninstall, a dialog will ask you whether or not to uninstall the additional service.

The Service Manager is used throughout the Visual Smalltalk documentation for installing optional functionality. Since the services are saved as Smalltalk libraries, you may also chose to install them using the Smalltalk Library Binder. For information about the library binder, refer to the Visual Smalltalk Enterprise Language Reference. The file name of the Smalltalk library for each service is included in the service description, displayed in the lower text pane of the Service Manager browser window.
Programming in Visual Smalltalk

CHAPTER 2

The Workspace

Workspaces are the scratch pads of the Visual Smalltalk system. Workspaces allow you to try out code while you are developing it. Each workspace is independent of all other windows in the system.

To open a workspace, select Smalltalk/New Workspace, or press Ctrl + W.

Workspaces are ideal for creating objects and defining variables to which you can send successive messages. Workspace variables are temporary variables local to the workspace. To use a workspace variable, simply assign it an object, just as you do to a temporary variable in a method. You do not declare it first. The variable remains available, and the object assigned to it persists, until you close the workspace.

The contents of a workspace can be saved as a text file and reopened later. This provides a convenient mechanism for creating test scripts, and storing any collect of expressions you may want to evaluate at a later time.

The Parts Workbench

The Parts Workbench is the visual programming component of Visual Smalltalk. It provides a high-level, interactive, graphical tool for assembling your application out of parts. Parts are simply objects that are either provided in Visual Smalltalk, that you design and create yourself, or obtain from a third party part provider.

The workbench environment consists of:

• Workbenches — a layout surface in which you place objects and connect them to build an application

• Catalogs — notebooks containing collections of parts

A new workbench is automatically opened when you select Smalltalk/New Workbench from any Visual Smalltalk window. You can use this workbench to begin developing your application. You open additional workbenches as needed to create additional parts of your application or to edit existing parts.

To open a workbench for an existing part file, Choose File/Open... from the workbench menu bar. A file selection dialog opens. Select a file and click OK.
You can also open a workbench for a part in the catalog by double-clicking on the part. This allows you to view and edit the definition of that part.

**Parts of the Workbench**

A workbench window opens looking like this:

At the bottom of a workbench is an optionally displayed text area, for displaying hints regarding parts, menu items, and links. You will want to leave this enabled while you are learning Visual Smalltalk.

To disable or enable the hint line, select **View / Show hints**. This is a toggle menu item that shows a check mark when the hint line is enabled, and removes the check mark when disabled.

**Workbench Cursors**

The workbench uses different cursor shapes to visually indicate system status. For example, the hour glass (Windows) or clock (OS/2) cursor indicates that a process is taking place, such as a file being read. The cursor also takes on the appearance of the part icon when one is being copied from the catalog. Here are some other common cursors:
Catalogs

Catalogs are notebooks containing pre-built parts, and are one of your primary sources for parts as you develop an application. They float on top of the active workbench to make them more easily accessible.

You use catalogs to build an application by dragging parts from a catalog and dropping them onto the workbench. The standard part catalog is displayed when you launch open a new workbench.

Opening a Catalog

You may open as many different catalogs as you like. By convention, catalog files have a .CAT extension, but this is not required.
To open a catalog:

1. Select View/Open catalog...
   
   If a catalog is already open, you can click the File... button in the open catalog and select Open...
   
   An Open Catalog dialog is displayed.

2. Locate and select a catalog file, then click on OK.

   The catalog is displayed.

Placing a Part on the Workbench

There are two sources of parts: the catalog and Smalltalk classes. A catalog part typically has its own icon, which represents the part in the catalog and on the workbench. A Smalltalk class does not have an icon associated with it, and so is represented on the workbench by a nested part icon.

To place a part on the workbench from a catalog:

1. Click on the desired part to select it.

2. Press and hold the mouse drag button.
   
   This is typically the left button on Windows systems and the right button on OS/2 systems.

3. Move the mouse cursor to the workbench.

4. Release the mouse button.

   The part will be placed on the workbench, with a name for the instance of the part shown below it.

Notice that some parts can only be placed directly on the workbench, such as the window and computation parts, and others can only be placed inside of a window part, such as a button part. If you attempt to drop a part where it doesn't belong, the operation will simply fail.

To place an instance of a Smalltalk class on the workbench:

1. Select Developer/Add object in the workbench menu bar.

   The Add Object dialog opens.
2 Select a class in the class list, or enter the instance creation expression in the text pane. Click the appropriate radio button, then click OK.

Selecting the class name adds the instance by sending new to the class. For most classes, this is the appropriate way to add an instance.

For some classes, a more complex expression may be required. For example, to add an instance of Color with an RGB mix, you might enter in the text pane:

\textbf{Color red:10 green: 14 blue: 8}

If the expression is correct, the dialog closes and an icon is displayed over the workbench.

3 Move the icon to a location on the workbench, and click the mouse button.

The instance is placed on the workbench, and you can work with it as with any other part.

\textbf{Creating Links}

When you have placed a part on the workbench, you activate it by drawing links between events it triggers and messages that other parts can answer. You do this, like most operations in the workbench, with the mouse.
For example, open a new workbench. On the window page of the part catalog, select the window part and drag it to the workbench. Notice the green line drawn between the border of the workbench and the border of the window part. That is the link between the open event and the window’s open message.

![Workbench Screenshot]

Click on the Show all labels button on the button bar to see the link labels. (If you don’t know which is the show labels button, watch the hints). Events are shown in a rectangular label, and messages are shown in the hexagonal labels.

In this case, when the application is launched, it triggers it’s open event. The link causes it to send the open message to the window, causing the window to open.

Now drag a button from the catalog and drop it onto the window part. No links are automatically drawn this time. Press the secondary mouse button (usually the right button) on the button part and drag the mouse to anywhere on the window part. A dialog opens listing the button part’s events and the window part’s messages. Select the button’s clicked event and the window’s close message, then click OK. The green link is now drawn. The next time you launch this part, you can click on the button to close the window.
To finish off the button, add a label to it. You can do this in the properties dialog, or directly. To add text directly, press and hold the Alt key, and click on the button. When the text cursor is displayed on the button, type “Close” and press Enter.

**Saving a Part File**

When you build a part in a workbench and want to save it for further editing later, you save it as a .PAR file. This is the standard format for part files until you are preparing your application for delivery.

To save a part using the current file name:

> Choose File/Save from the workbench menu bar.

To save a part to a new filename:

1. Choose File/Save as... from the workbench menu bar.
   
   A file selection dialog opens.
2. Select a destination directory.
3. Select PAR format.
4. Click the OK button.

**Launching a Part**

When you are ready to test your application, you simply launch it; no time consuming compilation is required. When you launch it, the application part’s open event is triggered.

A typical application has its open event connected to the open message of a window containing your application's user interface. You must link the application’s open event to some message if you intend to launch the application directly from the workbench or as a stand-alone EXE.

A launched application is independent of the parts in the workbench. If you change your application in the workbench, an already launched application will not show the changes.

To launch a workbench application, either:

> Choose Test/Launch from the workbench menu bar.

or

> Click on the toolbar's Launch button.
Closing a Workbench

To close a workbench:

> Double-click on the workbench control menu box

or

> Choose File/Close from the workbench menu bar

or

> (Windows 95) click the close button.

Package Browser

The main tool for working with packages is the Package Browser.

To open a package browser, either select Smalltalk / Browse Packages in a menu bar, or press Ctrl+P.

Parts of a Package Browser

The Smalltalk programming operations can all be performed from a package browser. A package browser, shown below, has the following parts.

The module list displays the names of all the modules (both clusters and packages) currently loaded in the system.
The *globals list* displays all global definitions (classes, global variables, and pools) contained in the selected module. When a cluster is selected, the globals list contains all of the classes and globals defined in the packages in the cluster.

The *category list* shows the categories into which the methods of the selected class are organized. Categories are useful for grouping related methods, such as accessor methods, initialization methods, or any other functional group. If the class contains methods that have not been assigned to a category, it shows the category **unclassified**. The category **all** shows all methods, regardless of their category.

Above the category list, two radio buttons, *instance* and *class*, let you choose whether to view instance method categories or class method categories.

The *method list* displays the names of all the class or instance methods in the currently selected category.

Above the method list is the *inheritance box*, a drop-down list showing all superclasses of the currently selected class. Select a class from this list, and the method list shows the methods belonging to the selected class and all of its superclasses.

Below the method list is the *implementor box*, a drop-down list of all classes in the inheritance list that implement the selected method. Initially, the immediate implementor of the method is displayed. You can view overridden method definitions by selecting another class in the implementor list.

The *contents pane* changes its format depending on whether the item selected in the upper panes is a module, a class or global, a category, or a method. For each of these, toggle items in the *Global* menu set whether the pane displays the comment, definition, initialization, or description for the selected item.

We will describe some of these options in the following sections. For a full description of all displays, refer to *Visual Smalltalk Enterprise Tools Reference*.

The *context button* shows the name of the package containing the selected definition. If the module name on the button is different than the module selected in the module list, you can press this button to switch the currently selected module to the package that contains the selected definition.

The *version button* displays version information for the selected definition. The button is left blank if there is no relevant version information. If there is revision information, the button will show version information, usually a creation date, time, and author for
the definition. You can click on the version button to open a
history browser on the selected definition, letting you view its
previous versions, if any. (History browsers are described in the
Visual Smalltalk Enterprise Tools Reference.)

Text Display Conventions
Visual Smalltalk Enterprise uses several format conventions for text
items in the extended tools, providing additional information
about the current state of a module, definition, or specification.
These conventions are summarized in the following table:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue Italic text</td>
<td>The item has been modified since the module containing it was last committed. If the item is a module, its contents have changed since it was last committed.</td>
</tr>
<tr>
<td>Black Bold text</td>
<td>Items shown in bold face are defined in the selected module. Items in regular face are defined in another module.</td>
</tr>
<tr>
<td>Underlined text</td>
<td>A class or category is shown underlined in the Package Browser if only some of its contents are defined in the selected module.</td>
</tr>
<tr>
<td>Regular text</td>
<td>Modules and definitions shown in regular text have not been modified since they were last committed.</td>
</tr>
<tr>
<td>Parentheses</td>
<td>A module name that appears in parentheses in an organizer is open but not loaded into the image.</td>
</tr>
<tr>
<td>Bold Green text</td>
<td>In module comparison browsers, green text indicates items that are in one module but not the other.</td>
</tr>
<tr>
<td>Red Italic text</td>
<td>In comparison browsers, red italic text indicates differences between the items being compared.</td>
</tr>
</tbody>
</table>

You can change the colors in the Preference Group editor, by selecting Options / Preferences, and then selecting Color.

Working with Packages and Repositories
The package browser provides support for the packages, clusters,
and repositories that Visual Smalltalk Enterprise uses to organize
the source code for a development project. While developing your
application, you create classes and methods in packages. The
classes have their usual place in the Smalltalk class hierarchy as well, but packages represent bundles of code, usually as functional units.
When your work is at a stage that you want to save it as a loadable module, rather than in your development image, you commit the package to a repository, a disk storage structure. Repositories provide permanent storage for your source code. Each time you commit your package to its repository, you assign it a new revision number, in this way making it easy to trace your work.

In the following sections we describe basic procedures for working with these organizational structures.

Creating a Package

In Visual Smalltalk Enterprise, you store all of your Smalltalk code in packages. Accordingly, before beginning to program, you should create a package for your work.

You can also use clusters to form groups of packages.

To create a new package or cluster:

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

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

The contents pane displays the module definition editor with the name of your newly created package or cluster added. The name is in blue italics, indicating that it has not yet been committed to a repository.
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 may be valuable to other users. 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. Information is added after you have committed a new revision of the module.

4 Select File / Save to save the comment.

Committing a Package

To permanently save the code in a package, you commit the package to a repository. The repository is a system of directories and files on disk. The repository must be created before committing packages to it, but you are given the opportunity to create the repository during the procedure.

Note that committing a cluster does not also commit the packages or other clusters it contains. To save your work, you must commit the individual packages, or use Module / Commit all.

To commit the changes you have made to a package:

1 Select the package in the module list.

2 Select Module / Commit...

The Package Commit Dialog opens, as shown below:
3 Select a repository for the package.

If the package has previously been assigned to a repository, it is shown in the Repository list. Otherwise, select a repository from the Repository list. You cannot commit packages to the Visual Smalltalk Enterprise system repositories.

If the repository you want to use for the package is not in the list, then you either need to connect to it or create it. You can click on the Repository button to access the Repository dialog. Procedures for connecting to and creating repositories are described in later sections.

4 Enter a comment for the package.

5 Edit the Revision number, if necessary.

A default revision number is shown in the space, incremented to the next valid revision number for this package. You can edit this number, but the revision number must be larger than the last revision number, or represent a branch.

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

The Release radio button increments the major revision number and changes the minor number to zero, indicating a major release of your package. For example, Revision 2.1.0.1 becomes 3.0.

6 Click OK to commit.

The changes to the package are then committed to the repository, and the dialog closes. The package and its contents are returned to normal type in the package browser.

**Loading a Package**

To be able to execute the definitions in a package, you must load the package or a cluster containing it. This adds the contents of the package to the development image for editing and for use in your application.

**To load a package:**

1 Select Module / Load... in the package browser.
2 Select the Repository, Module, and Revision for the package you want to load.

If you need to connect to a repository, click the Repository button and select the repository directory. Then select the module to load.

The comment for the module revision is displayed.

3 Check the **Initialize after load** check box, if necessary.

This check box allows you to control when the initializers for the module are run. For more information about initializers and controlling initialization, refer to the *Visual Smalltalk Enterprise Tools Reference*.

4 Click the **Load** button.

The module is added to the module list in the package browser, and you can access its contents.

**Creating a Repository**

You need to create a repository before committing a package or cluster to it. Repositories are located on disk. This procedure creates the necessary files on disk.

**To create a repository:**

1 Select **Smalltalk / Open / Repository Browser**.

The Repository Browser, shown below, opens.
You can also open the Repository Browser by clicking on the Repository button in the Commit dialog.

2 Select Repository / New... .

The Create Repository dialog opens.

3 Enter a name and directory path for the repository.

The repository name must be unique on the specified directory path. Note that the repository name does not correspond to a directory name, but is stored in a file.

Also select a file type (PVCS or File-based) and a locking type (Optimistic or Pessimistic). For full descriptions of these options, refer to Visual Smalltalk Enterprise Tools Reference.

To select a directory in a browser, click on the Set Path... button.

4 Click Create to create the repository.

The files and directories are then created on the directory path you specified. The repository is connected to the development system, so you can immediately commit packages to it.

Connecting to a Repository

Before you can use an existing repository, you must connect to it.

To connect to a repository:

1 Select Smalltalk / Open / Repository Browser.

The Repository Browser opens.

You can also open the Repository Browser by clicking on the Repository button in the Commit dialog.

2 Select Repository / Connect... .

A directory selection dialog opens.

3 Find and select the directory containing the repository, then click OK.

The Repository Browser is updated to show any repositories in this directory. All repositories shown are connected.

You can now load any modules in the repository or commit modules to it.
Working with Classes and Methods

The core of your work in Visual Smalltalk consists of defining objects. The following sections discuss the basic procedures for working with classes and methods using the package browser.

Finding Definitions

Several tools allow you to search for package, class, global, and method names, and permit the use of wild cards and partial names. The following table shows which menu choices permit wild-carded or partial names.

<table>
<thead>
<tr>
<th>Menu Choice</th>
<th>Wild-cards</th>
<th>Partial Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module / Find Module... (in Package Browser)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Global / Find Class (or Global)... (in Package Browser)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Smalltalk / Browse / Implementors of...</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Smalltalk / Browse / Senders of...</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

A wild card includes an asterisk in the pattern to match any number of characters, including none. A partial name is treated just like placing a wild card character at the end of a string. Case is ignored when finding matches.

For example, entering “add:*” in Smalltalk / Browse / Implementors of... finds all definitions that implement add:, add:after, add:afterIndex, add:before, and so on.

Defining Classes

To define a class:

1 Select the package in which you want the class to reside.

   You may have to expand clusters if the desired package is not displayed. Click on a cluster expansion icon showing a ‘+’ sign to expand its display.

2 From the Global menu, choose New / Class.

   The contents pane displays the class definition editor.
3 In the **Class** field, type the name of the new class.

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

   If a class name is selected in the class/globals list, this is provided as a default superclass name. You can use it or replace it with any other class name.

5 Change the selection in the **Indexable** combo box, if necessary.

   The **Indexable** combo box lets you specify whether instance variables are indexable and, if so, whether they are object or byte indexable. Refer to the *Visual Smalltalk Enterprise Language Reference* for an explanation of these options.

   The initial selection is *none*. This is the most common setting, so you can usually leave it alone.

6 Specify variable names for the class.

   Classes can have four kinds of variables: instance variables, class variables, class instance variables, and pool variables. Most classes have instance variables, typically storing property information for the instances of the class. The others are used less often. Refer to the *Visual Smalltalk Enterprise Language Reference* for a full description.

   If you already know some variables the class will need, you can define them now. If you don't know, wait until later and add variables as you find you need them.
Select a variable type, and enter one or more variable names, then press Enter. The variable name is added to the list.

For pool variables, the pool must already be defined before you can add its name to a class definition.

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

If a variable is selected, the comment applies to that variable; otherwise the comment applies to the class.

8 Save the definition.

When you are finished entering the class definition, choose File/Save to save the definition. Your new class appears selected in the global list.

Editing a Class Definition

As you work on your program, you may need to modify the class definition, usually to add or remove variables or to update a comment.

To edit a class definition, select the class in the class/global list. Under the Global menu, select Definition to display the class definition. Then edit the definition as when creating the class.

When you are finished, save the definition. Its name will be displayed in blue italic type, indicating that it has been modified but not committed to a repository.

Creating a Method Category

As the number of methods in a class grows, it becomes helpful to be able to group methods into functional categories. The Visual Smalltalk Enterprise supports method categories in the Package Browser. While categories have no effect on the way your application operates, using category names can make your application more understandable and maintainable.

Categories can be hierarchical. For example, you may have several subcategories of private methods. When creating subcategories, you specify the main and subcategory names, separated by a dash (-). The following figure shows two hierarchical categories, the private category having been created with these category names:

- private-accessing
- private-displaying
- private-initialization
A hierarchical category has an expansion indicator to its left.

![Package Browser](image)

To create a method category:

1. Select the package and class for the category.
2. Select Method / New Category... in the package browser menus.
   A category name prompter opens.
3. Type a descriptive name for the new category.
   To enter a subcategory, type the main and subcategory names, separated by a dash (-). If the main category doesn’t already exist, it will be created as well.
4. Click OK.
   The category is added to the package browser category list for the selected class.

Creating a Method

To define a method:

1. Select the package and class in which to define the method.
2. Click on the radio button to set the method type.
   Methods are either class methods or instance methods.
3. Select Method / New in the package browser menu.
   The new method creation dialog opens.
If you selected a category in step 2, the method editor window is displayed in the Package Browser, showing a method definition template.

4  Edit the template to define your method.

   Replace the selector, comment, temporary variable declaration, and statements with the text defining the new method.

5  Click on OK to save your method definition.

   The new method is added to the selected package, and is displayed in the package browser method definition window.

   You can also create a new method by editing an existing method, changing the selector and statements as needed, and saving it. The new method is created in the same class and category as the method you edit.

**Editing a Method**

To modify a method:

1  Select the package and class in which the method is defined.

2  Select the method category in the category list.

   You can select the special ** all ** category to show all methods.
3. In the method list, select the method you want to modify. The method source appears in the contents pane.

4. Edit the method to make your changes.

5. Press Alt + S, or select File / Save to save your changes.

### Class Hierarchy Browser

The Class Hierarchy Browser is the browser that lets you examine the relationships among the classes within Visual Smalltalk, and to edit their definitions and methods.

The Class Hierarchy Browser is the main class and method editor in Visual Smalltalk, and provides many of the functions served by the package browser in Visual Smalltalk Enterprise. Using the class hierarchy browser, you create and edit classes and methods. Its menus provide a wide variety of tools for viewing definitions. This is the tool you will use most while working with Smalltalk code.

If you are using Visual Smalltalk Enterprise, the class hierarchy browser is still useful as an auxiliary view on your work, making it easy to see the overall class hierarchy. If you edit classes and methods using the class hierarchy browser, your changes are made in their packages. However, if you create a new class or method, it is placed in **unpacked**. You can move it to your own package later.

To open a class hierarchy browser on the whole class hierarchy, either:

- Select Smalltalk / Browse Classes, in Visual Smalltalk,
- Select Smalltalk/ Open/ Class Hierarchy Browser, in Visual Smalltalk Enterprise, or
- Press Ctrl+B

### Parts of a Class Hierarchy Browser

A class hierarchy browser is divided into five panes:

The **class hierarchy list** contains the names of all of the classes in the system, presented in a hierarchical order, in alphabetic order within a level. **Object** appears first in the list as it is at the top of the hierarchy. Indentation is used to indicate that a class is the subclass of the preceding level.
A class name followed by ellipses (...) indicates that the class has subclasses that are not being shown. Double-click on the class name to expand levels, showing all subclasses. Double-click on an expanded class to collapse all levels below it.

The *variables list* shows the variables defined for the class selected in the class hierarchy list plus those that are inherited from its superclasses. Variables defined in the selected class appear at the top of the list, followed by variables inherited from the direct superclass, then by variables from the superclass of that class, and so on, ending with the variables inherited from class *Object*. Either class or instance variables are listed, depending on which radio button is selected.

The *methods list* pane displays a list of either the instance methods or the class methods, depending on whether the *instance* or *class* radio button is selected. If a variable is selected in the variables list pane, the methods list shows only those that reference the selected variable.

The *contents pane* is a text pane that allows you to edit class definitions and method source code. When you select a class from the class hierarchy list, its class definition is displayed in this pane. When you select a method from the methods list, its code appears in this pane.
CHAPTER 2

Working with Classes

Finding a Class
To select a class to browse, scroll through the class hierarchy list until you find the class you are looking for. Move the cursor into the class hierarchy list and click on the name of the class you wish to examine or modify.

Since the class hierarchy is quite large, it can be difficult to find a specific class. You can find a class quickly using Class / Find... . In the prompter, enter the class name and click OK. The find feature is not case sensitive.

Examining a Class
When you select a class, the browser displays the class definition in the contents pane, a list of variables in the variables list pane, and a list of methods implemented by the class in the methods list pane.

The Class menu contains the functions you use to maintain the hierarchy, including adding and removing classes, updating the hierarchy pane after modifications, opening a Class Browser on a selected class and writing out the definition and methods of a class to a file.

Adding Classes
You can create a new class as a subclass of any other class.

To create a new class:

1. Select the class in the class hierarchy list that is to serve as the superclass.

2. Select Class / New...

   A dialog box opens, prompting you for information about the class. The box shows the selected class as the superclass of the new class.
3 Enter a name for the new class.

The name you enter must not be the name of any other class.

4 Set the instance variable type radio buttons.

The two pairs of radio buttons specify two properties of the instance variables for this class:

- Setting Indexed to Yes makes the class have indexed instance variables. The default is No, so the class has only named instance variables.

- Indexed instance variables can contain either byte values or pointers. Named instance variables can only contain pointers.

Indexed instance variables are not often used, so set Indexed to No unless you know you need indexed instance variables. Refer to Visual Smalltalk Enterprise Language Reference for more information about instance variables.

5 Click on OK.

The new class is then added to the class hierarchy with its definition displayed.
CHAPTER 2  Programming in Visual Smalltalk

Editing a Class

The only editing you can do on a class is to change its variables. The class definition identifies the class’s instance variables, class variables, pools, and class instance variables.

To edit a class definition:

1. Select the class in the class hierarchy list.
   Visual Smalltalk displays the current class definition in the contents pane.

2. Set either the Instance or Class radio button.
   With the Instance button set you can edit the lists of instance variables, class variables, and pools. With the Class button set, you can edit the list of class instance variables.

3. Edit the variable lists in the contents pane.
   Add new instance variable names, class variable names, and pool dictionary names between the single quote marks under the appropriate heading. You can also delete names to remove them from the class definition.

4. Select File / Save.
   The edited class definition is saved and displayed in the contents pane.

When you change a class definition, Visual Smalltalk recompiles all of the methods in the class and its subclasses. In addition, the new class definition is written to the change log. Changes to a class definition take effect immediately. All future instances of the class have the new structure.

If any errors occur while recompiling, the nature of the error and the affected method are shown in the Transcript.

CAUTION: Do not redefine or edit classes used by the system. Low-level portions of the system may depend upon a certain implementation for those classes, so redefining them risks causing a system failure.

Removing Classes

You can delete a class from the class hierarchy as long as it has no active instances or subclasses.

To remove a class:

1. Select the class in the Class Hierarchy browser.
2 Select Class / Delete.

A confirmation dialog is displayed asking you to confirm your choice.

3 Click on Yes.

The class is then deleted from the class hierarchy, unless the class has instances or subclasses.

If the class has instances or subclasses exist, a walkback window will open. Close the walkback window and fix the problem.

Working With Methods

The class hierarchy browser includes full method editing capabilities. When you select a class, the selectors for methods in that class are displayed in the methods list pane. Either class methods or instance methods are displayed at one time, based on whether the Instance or Class radio button is selected.

To view the source code for a method, simply select the method selector in the methods list. The source text is then displayed in the contents pane.

The variable list displays the variables defined for the class. By selecting a variable, you can filter the method list to include only those methods that reference the selected variable. The Variable menu contains three items, controlling the filtering:

- Assigned displays only those methods that assign a value to the selected variable.
- Used displays only those methods that use the selected variable.
- Both (the default selection) displays all methods that either assign or use the selected variable. Note that for class variables both are always shown.

Adding a New Method

To add a new method:

1 Select the appropriate radio button: instance to add an instance method or class to add a class method.

2 Select Method / New.

An editable method template appears in the contents pane.

3 Select and type over the placeholder template items to create a new method.
When the method appears as you require, select Save from the File menu to invoke the compiler and install the new method.

You can also create a new method by editing an existing method, changing the selector and statements as needed, and saving it. The new method is created in the same class.

**Editing a Method**

To modify an existing method:

1. Select it in the methods list.
2. Edit the source code in the contents pane, but not the message selector in the first line.
3. When the method appears as you require, select Save from the File menu to recompile the method.

If the compiler detects an error when you save a method, a message displays in the method text where the error was detected. Delete the message and edit the text to correct the error. Then save the method.

**Deleting a Method**

To remove a method from the selected class:

1. Select the method selector in the methods list.
2. Select Method / Delete from the menu.

The methods list is immediately updated to verify that the selected method has been deleted from the class.

**Method Editing Tools**

The following two tools are useful for working with methods. They behave the same for both Visual Smalltalk and Visual Smalltalk Enterprise.

**Method Browser**

The Method Browser lets you browse and edit a list of methods.

To open a Method Browser:

1. Select the method in the Package Browser or Class Hierarchy Browser methods list.
2. Select any of the menu items from the Method menu: Senders, Implementors, Local Senders, or Local Implementors.

You can also select Smalltalk / Browse / Senders of... or Implementors of..., and enter the method name in the dialog that opens.

Method browsers have two panes. The method list pane displays a list of methods identified by class and message selector. When you select a method, the contents pane displays the source code for that method. The contents pane allows you to view and edit the source code for a selected method.

Message Browser

The Message Browser displays a list of all messages sent in a method. For each selector, you can then view a list of either senders or implementors of that message.

The selectors list shows all the selectors of messages sent in the method on which the browser was opened. The methods list identifies either the senders or the implementors of the chosen selector, depending on whether the senders or implementors radio button is set. When a method is selected, the contents pane displays the source code for the method.
To browse selectors:

1. Select a class in the Class Hierarchy Browser or Package Browser.
2. Select a method from the methods list.
3. Select Messages from the Method menu.
4. Click either the Senders or the Implementors radio button.
5. Click on a selector in the selectors list.

The method defined for the associated with that selector is displayed in the contents pane. In addition, the expression containing the selector is highlighted. This can be helpful when trying to identify keyword messages where arguments are interspersed with components of the selector.
Inspectors

Inspectors are used to examine objects in the system. They are an important tool for understanding how the objects in your application are behaving.

There are several different kinds of inspectors. The following figure shows a general inspector, used for any kind of object that does not have a special inspector.

Inspectors have two panes:

- The instance variables list shows all the instance variables of the object being inspected. For most objects, the first item in the list is self, the object being inspected. Named instance variables, if any, are listed next, followed by indexed instance variables, if any.
- The contents pane displays the value of the object selected in the instance variables list.

Specialized inspectors generally add menus or items under the Inspect menu item. The special inspectors are described briefly in the following sections.
Opening an Inspector

There are several ways to open an inspector.

- Send the message `inspect` to any object. For example, evaluate this expression with `Do It` to open an inspector on the point:
  
  \[(2 @ 3) \text{ inspect}\]

- Select an object expression in a text pane and select `Smalltalk/Inspect It` or press Ctrl+I.

- In an inspector, select an object in the variable list, then select `Inspect / Inspect` or double-click on the object to open an inspector on the object.

These methods open a special inspector if there is one for the object, so it’s not necessary to specify the kind of inspector you want.

Editing Objects in an Inspector

When you select one of the variables from the list, its current value is displayed in the contents pane. If you select the `Inspect` item from the `Inspect` menu, or double-click on the instance variable, a new Inspector opens on the selected instance variable.

The contents pane is a text pane similar to a workspace. You can type in this pane, select your expressions, and use relevant items from the `File`, `Edit`, and `Smalltalk` menus as required. As you do so, bear in mind that:

- Any expression that you evaluate is compiled in the scope of the object being inspected. This means that you can use the names of all of the instance variables in your expressions.

- If you select `File/Save`, the entire contents of this pane are compiled and evaluated. The result replaces the current value if an instance variable is selected.

- If you select `Edit/Restore`, the inspector places the current value of the selected instance variable into the contents pane.

**WARNING:** If you are inspecting an object used by the development environment, do not change its value or the values of any of its instance variables, unless you are absolutely sure how your change will affect the system. In any case, save your image before doing so.
Using Special Inspectors

Inspecting Dictionaries

A Dictionary object associates keys with values. The Dictionary Inspector has two panes: a keys list pane and a contents pane. When you choose a key in the keys list, the associated value is displayed in the contents pane.

The Dictionary menu items are:

- **Inspect** opens an inspector on the selected key.
- **Add...** allows you to add a new element to the dictionary. The system prompts you for the new key. The associated value is **nil** until you change its value in the contents pane and save the change.
- **Remove** deletes the selected item from the system.

Inspecting Ordered Collections

Instances of the class **OrderedCollection** are indexable, extensible collections whose elements are ordered. The Ordered Collection Inspector has two panes: an index list pane and a contents pane. The index list pane lists the indices of the collection’s elements in the instance variables list. When you choose an index in the list, the associated value is displayed in the contents pane.

When you select one of the variables from the list, its current value is displayed in the contents pane. If you select the **Inspect** item from the **Inspect** menu, or double-click on the instance variable, a new Inspector opens on the selected instance variable.

Inspecting Byte Arrays

The Byte Array Inspector is similar to the general inspector in most respects, with the following differences:

- When **self** is selected from the instance variables list, a byte-oriented dump of the object is shown in the contents pane. This dump is columnar, displaying the offset, followed by up to 16 bytes of data in hexadecimal format, followed by the same data in character format. Nonprintable characters are displayed as “.”.
- Select **Inspect/Radix** to choose the radix (base) in which the numeric data column is displayed: binary, octal, decimal, or hexadecimal (default).
The Byte Array Inspector may also be used for any other kind of object, as long as it understands the `dumpStringBase: aRadix` message.

### Inspecting Fields

The Field Inspector allows you to inspect the fields and contents of external data structures. The Field Inspector works with instances of `ExternalBuffer`, `SelfDefinedStructure`, and other objects that respond to the `inspectorFields` message.

You can also open a Field Inspector using the `openOn:` method:

```smalltalk
FieldInspector new
openOn: anObject
```

The object specified must respond to the `fields` message.

### Debugging Tools

Two kinds of windows are available to help you debug your program: Walkback windows and Debugger windows.

Smalltalk automatically opens a Walkback window when a run-time error occurs. You can also force a Walkback by interrupting program execution. The Walkback displays the sequence of messages and conditions that caused and immediately followed the error condition. Often, this is enough information for you to fix the problem.

When you need more information than is provided in the Walkback, you can request a Debugger window by clicking the `Debug` button in the Walkback. Using the Debugger, you can investigate the state of the application by examining the values of variables and the messages used. You can also edit the program right in the Debugger, correcting programming errors on-the-fly.

### Walkback

Visual Smalltalk opens a Walkback window automatically any time a run-time error occurs. Common run-time errors include:

- stack overflow
- a message is not recognized by the receiver
- subscript out of range

Some of these errors can be handled by the program. If an error handler isn’t defined, the Walkback is opened.
Besides the automatic Walkback, you can request a Walkback in several ways:

- By sending the **error** message to any object, including a string argument describing the error. For example:

  ```smalltalk
  self error: 'Index is outside of collection bounds'
  ```

- By sending the **halt** message to any object. For example:

  ```smalltalk
  self halt
  ```

- By pressing CTRL + Break while the program is running. You are likely to do this to interrupt an unintentional loop.

This screen shows a walkback produced by attempting to access the eighth character in the string **Hello**, an obvious error.

![Visual Smalltalk Transcript]

The window title gives a brief description of the error.

The text pane of the window contains a **method walkback**, showing the message-sends that led up to the error and opened the Walkback. Each line in the method walkback represents a single message-send, with the most recent listed first.

Each message send begins with the class of the receiver. If the method is inherited from a superclass, the class defining the method is shown next in parentheses. The message selector is then shown as a string following ">>."

A variation looks like this:

```smalltalk
[ ] in ClassName >> methodName
```
This indicates that an error occurred while evaluating a block in the method **methodName** defined in the class **ClassName**.

When you see a walkback, you can respond in one of the following ways:

- If the walkback occurred as a result of a **halt** message, a CTRL + Break, or a resumable exception, you can click on the **Resume** button to continue from the point where execution was interrupted.

- If you can determine the problem from the information contained in the Walkback, close the Walkback. Close the walkback by double-clicking on the walkback’s system menu, or by clicking the **Terminate** button. You can then fix the problem and start execution over at the beginning.

  When you close a Walkback, the associated process is terminated and any **ensure blocks** associated with the process are executed. If bugs in the ensure blocks cause recursive walkbacks, use the **Debugger/Drop Process** option (see **Debugger** below).

- If you need more information, click on the **Debug** button. The Walkback is replaced with the Debugger window.

### Debugger

The debugger gives you an expanded view of the walkback, and allows controlled execution of a process. The debugger window has four panes and five buttons.

The top left pane shows a list of message-sends. When you select a list item, the other panes display additional information about that item.

The top left pane shows the same list of message-sends as the walkback window. When you select a message-send line, the method is shown in the bottom text pane with the affected expression highlighted.

The top center pane lists the receiver, arguments, and variables of the selected method. If you select one of these, the **contents pane** on the right displays the value of the selected item. You can open an inspector on any item by double-clicking on it.

You can edit the method in the bottom pane and save it, just as you do using the Class Hierarchy Browser. When you edit and save a method, any ensure blocks for methods later in the walkback sequence are run, to ensure a consistent state.
The *Hop*, *Skip*, and *Jump* buttons can be used for resumable processes. They work as follows:

- *Hop* sends the highlighted message, but returns control to the debugger before the method executes. *Hop* is equivalent to *Skip* for methods that are implemented by a primitive or do not send a message.

- *Skip* sends the highlighted message and executes the resulting method before returning control to the debugger.

- *Jump* executes up until control returns to the highlighted method.

The following special debugging items are available under the *Debugger* menu.

- *More levels* adds ten more items to the walkback list each time it is selected.

- *Filter stack* shortens the message send list by showing only the first and last message send to a single object, hiding intervening message sends.

- *Resume* continues execution after a *halt* message. The debugger closes and execution continues from the point of interruption. You cannot resume if you have changed a method in the walkback list.
• **Restart** closes the debugger and restarts execution by resending the message in the selected walkback entry.

• **Terminate** ends the current process and closes the debugger window. Process termination runs any ensure blocks associated with the process.

• **Drop Process** terminates the current process without executing any of its ensure blocks, and closes the debugger window. Normally, you only use this option in situations where there are bugs in the ensure blocks that would cause recursive walkbacks. When you select this option, Visual Smalltalk prompts for confirmation before continuing. Note that this option should only be used when an error occurs while terminating a process or when you know that ensure blocks should not be executed.

**Saving Smalltalk Source Code**

There are a variety of methods for saving Smalltalk source code in Visual Smalltalk. The following sections briefly describe the methods available and when they are most appropriate.

**Saving the Smalltalk Image**

The traditional method of saving a Smalltalk program, including all source code, is to save the Smalltalk image. You do this either by selecting *File / Save Image...* in any Smalltalk menu, or by clicking *Yes* in the prompter dialog when exiting Visual Smalltalk. This is an easy and convenient means for saving your work.

There are complications with this approach, however. The main problems are that the image can become very large, and cleaning up dependencies prior to delivering the complete application can be difficult.

Regardless of these difficulties, saving the image is often a useful interim method for saving your work, especially if you are not using Visual Smalltalk Enterprise. The advantages of working with packages lead some users of Visual Smalltalk Enterprise to never or only rarely save their Smalltalk image.

When you save the image, the current state of your development image is saved. For this reason, if you know your development image is in an error state, any condition causing a Walkback, *do not save the image*. Fix the error condition before doing a save image.
Saving Packages

Packages are the main storage mechanism for Smalltalk source code in Visual Smalltalk Enterprise. While working, you select a package in which you create your classes and methods. At appropriate times you commit your packages to repositories to save the source in a permanent location.

By committing your packages to their repositories before exiting Visual Smalltalk Enterprise, you can avoid ever having to save your Smalltalk image. At the beginning of a new session, you simply load your packages and continue working.

When you are ready to deliver your application, tools are available to build the necessary delivery files directly from your packages. This makes application delivery a very simple process.

Procedures for creating and working with packages are described elsewhere in this chapter and in the Visual Smalltalk Enterprise Tools Reference.

Smalltalk Link Libraries

Smalltalk link libraries, also called simply “Smalltalk libraries,” provide a good alternative to packages for users of Visual Smalltalk (not Visual Smalltalk Enterprise). Some more set-up work is required, however, and you need to create your own library build scripts.

The general idea is that you write every Smalltalk class and method you create to a Smalltalk library file before exiting Visual Smalltalk. This is simplified by creating a script in a workspace and evaluating it when you are ready to update your library. To access your application code, you bind the library to your Smalltalk image and continue editing.

There are no tools to keep track of your code in Visual Smalltalk as there are in Visual Smalltalk Enterprise, but you can accomplish much of the same effect by creating a library build script at the beginning of your project. As you proceed, simply add each new class to your library build script. If you create methods for classes in other libraries, you add them to your script also. Similarly for any other objects that you create for your application, such as pools or globals.

This strategy also simplifies building your final application. Since your application code is usually delivered in Smalltalk libraries, having constructed scripts to build them along the way simplifies the final build process.
File Out

Filing out your classes and methods is a way of writing just source code to a text file. This method used to be the main approach for sharing application source between programmers and porting between different operating system platforms.

File out format is not used as much any more as it used to be, but is still a useful way to save your code. It is also a portable format between Visual Smalltalk and Visual Works.

The following menu items file out the source for classes and methods in your Smalltalk image:

- Class / File Out… writes the definition of the selected class and all its instance and class methods. The file is in change log format, and the file name has a .CLS suffix. Subclasses are not filed-out.
- Class / File Out All… is like File Out…, but includes all subclasses and their methods.
- Method / File Out… writes the definition of the selected method. The file is in change log format, and the name has a .MTH suffix.

Object Filer

The Object Filer provides another mechanism in Visual Smalltalk for storing objects and exchanging them among Smalltalk images. You can store and exchange code by filing out the source code of classes and methods. The Object Filer provides an equivalent mechanism for storing and exchanging the data represented by objects in your image.

The Object Filer performs these basic operations: dump an object, load an object, and produce a report describing a binary filed object or its classes.

For more information about the Object Filer, refer to the Visual Smalltalk Enterprise Language Reference.
Overview

The current state of the Visual Smalltalk environment is recorded in three synchronized files. Both Visual Smalltalk and Visual Smalltalk Enterprise use these two files:

- **V.EXE**
  - The Smalltalk image file

- **CHANGE.LOG**
  - The Smalltalk change log

In addition, Visual Smalltalk, but not Visual Smalltalk Enterprise, uses:

- **SOURCES.SML**
  - The source code for the image

Visual Smalltalk Enterprise uses:

- **RECOVER.LOG**
  - A history of changes made to the image

As you work in Visual Smalltalk, the current state of the Smalltalk image changes. When you save the image, the current state of the image is written to the V.EXE image file. The image file reflects the entire state of Visual Smalltalk at that moment, including compiled code for methods.

The source code for any additions or changes you make to class and method definitions is saved in the CHANGE.LOG file. Visual Smalltalk maintains pointers from the compiled methods in the image to the source code in the change log. Both Visual Smalltalk and Visual Smalltalk Enterprise use the change log, but record slightly different information.

SOURCES.SML, which is used only by Visual Smalltalk, stores a condensed set of source code for methods in the image. When the change log gets large, you can compress sources to create or update the SOURCES.SML file and initialize the change log. The change log then continues recording new code and changes to source recorded in SOURCES.SML.
RECOVER.LOG is used by Visual Smalltalk Enterprise to record a history of changes to the image, including expressions you have evaluated. The history gives a way to “play back” changes you have made to the system, restoring the image to its condition prior to the crash.

On each system, the three files are closely synchronized. When you back up your image, make sure you copy all three files. Mismatched files may prevent the system from starting, or at least cause a loss of source code.

For this reason, never directly edit CHANGE.LOG, SOURCES.SML, or RECOVER.LOG. Changing the files may also cause the system not to run. Instead, use the procedures described in this section to maintain these files. In some cases, however, it is useful to open the files as text files, to view their contents.

**Saving the Image**

The Smalltalk *image* determines the state of the Smalltalk environment, including all the active objects, both compiled code and data. The image is read from the V.EXE file and bound Smalltalk library files when the system starts up. All objects are loaded into virtual memory including any open windows.

Since Visual Smalltalk is an interactive and modifiable environment, the image is constantly being changed as you use and modify Visual Smalltalk. These modifications are not written on the disk until you save the image.

You can save the image by either selecting the File/Save Image... menu item or by selecting Yes in the Save image? dialog when exiting Visual Smalltalk. The next time you start up the system, the Visual Smalltalk environment resumes exactly as it was when the image was last saved.

If you choose not to save the image, changes you had made during a session are still recorded in the change log, so you can still selectively recover these changes. Just open the change log, and copy the code you want from the change log to the method editor.
Recovering from an Image Crash

Visual Smalltalk is very resilient, but disasters can and will happen. The automatic logging features of Visual Smalltalk make it possible to recover from most disasters with little or no loss of work.

To maximize your chances of recovery, take the following precautions:

- Always have a backup copy of all Visual Smalltalk files, including all .SLL and .DLL files.
- Always have a reliable backup copy of the V.EXE, CHANGE.LOG, and SOURCES.SML or RECOVER.LOG files. Remember, these files work together, so always back up and restore them as a set.
- Save your work before trying something that might crash the system. Either commit your changes to their repositories, save the image, or update your Smalltalk libraries.
- If you have any reason to believe you have damaged the system, do not save the image. Instead, exit without saving your work, then start up Visual Smalltalk again and start over.
- If Visual Smalltalk crashes, don’t panic. Make a copy of your V.EXE, CHANGE.LOG, and SOURCES.SML or RECOVER.LOG files. Usually, you can recover your work, and if you make a copy, you can try repeatedly.

The procedures described in the following sections will help you recover your work following an image crash. The procedures are different for Visual Smalltalk and Visual Smalltalk Enterprise.

Recovering in Visual Smalltalk

Visual Smalltalk uses the change log to record changes you make to the system. It records classes and methods you define and expressions you evaluate in “file out” format, so you can restore work by selecting a chunk (delimited by exclamation points) and filing it in.

You can use the following procedure any time you need to recover work that you did not save.
To recover lost work, do the following:

1. **Copy your V.EXE, CHANGE.LOG, and SOURCES.SML files.**
   
   You want to make sure you have a copy of your latest work before proceeding, to avoid overwriting your last “good” image and source files.

2. **Try to start Visual Smalltalk as usual.**
   
   In most cases, the development system will start up simply using the last saved image. If so, go on to step 4.
   
   If Visual Smalltalk does not start, reboot the operating system and try again.

3. **If Visual Smalltalk still does not start, use your backup copies.**
   
   Often it is enough to delete V.EXE and rename V.BAK to V.EXE, and try again. If this doesn’t work, then copy your most recent backup copy of the V.EXE, CHANGE.LOG, and SOURCES.SML, and try again. These three files work together, so restore all three as a set.

4. **When you have Visual Smalltalk running, open the change log.**
   
   Use the File/Open... item to open the CHANGE.LOG file that you were using when the system crashed. The CHANGE.LOG file has all of the modifications that you made and all of the expressions that you evaluated.

5. **Find the point in the CHANGE.LOG file where you saved the image that you are currently running.**
   
   Every time that the image is saved, Visual Smalltalk writes a comment into the CHANGE.LOG file giving the date and time when the V.EXE file was saved. Scan the CHANGE.LOG file contents backwards, if you were using a recently saved image.
   
   Your lost work is recorded between the last image save and the end of the file.

6. **Select a portion of code you want to restore to the image.**
   
   The CHANGE.LOG file is formatted as a series of chunks. To restore your work, select one or more consecutive chunks of code. You must select complete chunks, including the exclamation points (!). For methods, be sure to include the leading information identifying the class and method type.
   
   Choose the chunks that you install carefully! One of them may be the error that caused your system to crash.
Choose Smalltalk / File It In.

If a chunk is an expression, it will be evaluated. If a chunk is a method definition, it will be recompiled and installed into the system.

Repeat steps 6 and 7 until you have recovered your work.

When you have finished recovering what you want, save your image and make a backup.

Recovering in Visual Smalltalk Enterprise

Visual Smalltalk Enterprise maintains RECOVER.LOG specifically for recovering from system failure. The CHANGE.LOG is still used to record source code and versioning information for all methods and other permanent source pointers, but is not used to recover changes from an unsaved image.

The RECOVER.LOG file contains a history of every change you make to your Smalltalk image, including all expressions you have evaluated with Do It or Show It. You use the RECOVER.LOG file to “replay” changes you have made since the last time you saved your Smalltalk image.

You can use the following procedure any time you want to recover work that you did not save.

**To recover lost work, do the following:**

1. Copy your V.EXE, CHANGE.LOG, and RECOVER.LOG files.

   You want to make sure you have a copy of your latest work before proceeding, to avoid overwriting your last “good” image and source files.

2. Try to start Visual Smalltalk Enterprise as usual.

   In most cases, the development system will start up simply using the last saved image. If so, go on to step 4.

   If Visual Smalltalk Enterprise does not start, reboot the operating system and try again.

3. If Visual Smalltalk Enterprise still does not start, restore your backup copies.

   Often it is enough to delete V.EXE and rename V.BAK to V.EXE, and try again. If this doesn’t work, the copy your most recent backup copy of the V.EXE, CHANGE.LOG, and RECOVER.LOG, and try again. These three files work together, and so restore all three as a set.
4 Choose Smalltalk / Open / Recovery Log to open the Recovery Log Workspace.

5 Select the text between the “saved image on:” line and the end of the recovery log.

By default, only information since the last image save is shown. If you do not regularly save the image, installing code from packages instead, you only need to find the changes you made during the last session.

*Choose the chunks that you install carefully!* The end of the file may include a change or an expression that caused the image to crash.

The default is to not perform *Do Its*, which are the typical cause of a system failure. To change this setting, select *Recover / Skip Do Its*.

6 Choose the Smalltalk / File It In menu item in the Recovery Log Workspace to replay selected changes.

The File In process invoked from the Recovery Log Workspace is different from the process invoked from other browsers and windows, and is required for this procedure.

When processing completes, all changes have been reinstalled in your image.

7 Save your work.

Once you’ve restored the system to the point before the image crash, either save the image or commit your packages, as appropriate to your working method.

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.

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. If the image crashes again while replaying, start over, but avoid the last change or expression in the recovery log.
Maintaining Log Files

Visual Smalltalk and Visual Smalltalk Enterprise each record development system activities in two log files:

- Visual Smalltalk uses CHANGE.LOG and SOURCES.SML
- Visual Smalltalk Enterprise uses CHANGE.LOG and RECOVER.LOG

These files require occasional compression to reduce their disk space use.

Compressing the Change Log

Since all changes you make are recorded in the change log, the file can become very large. To reduce its size, you compress the change log, which cleans out old definitions, keeping only the latest version.

In Visual Smalltalk, compress the change log by evaluating the following expression in a workspace:

SourceManager current compressChanges

You can also clean up the change log by writing the latest source to SOURCES.SML by evaluating this expression:

SourceManager current compressSources

To compress changes in Visual Smalltalk Enterprise, select Smalltalk / Team / Compress Changes.

Compressing the Recovery Log

Since RECOVER.LOG records every change you make, it can grow quite large over time. When it gets too large, you will want to reset the log.

Do not delete or edit the recovery log file. The image remembers the place in the file that marked the last image save.

Instead, use the Recover / Reset Log menu item in the Recovery Log Workspace to discard the contents of the log file. The recovery log is also modified when you condense changes.
Managing Memory

When an object is created, it occupies some of the computer's memory. When the object is no longer needed, when there are no more references to it, the object is deleted and the memory occupied is reclaimed by the garbage collector.

In the normal course of things, this is all handled automatically by Visual Smalltalk. There are a few instances, however, where more direct involvement is required.

Finalization

While an application program should always release all memory resources it consumes, this is sometimes difficult. Especially in the case of external resources, mistakes occur.

Visual Smalltalk supports a backup system, called finalization, which programmers can build into their classes to assure that resources are released when they are no longer needed. While a programmer should take every care to release any objects and external resources, using the finalization feature is useful to defend against mistakes.

Finalization support is described in the Visual Smalltalk Enterprise Language Reference, in the “System Classes” chapter.

Removing Instances

It is occasionally necessary to remove remaining instances of a class. This is necessary, for example, when you need to delete a class that has lingering instances.

CAUTION: If the class is a user interface component, the system may require instances of it. Removing all instances may leave your image in an inconsistent state. To avoid losing work, save your image before proceeding.

To remove instances of a class:

1. Save your image.
2. In a workspace, type the expression, substituting the appropriate name for MyClass:
   
   MyClass allInstances

3. Select the expression and inspect it to open an inspector on an array of all the instances of the class in your image.
In the text pane of the inspector, type the expression:

```smalltalk
(self at: 1) allReferences
```

Select the expression and inspect it to open an inspector on an array of all objects that refer to the first instance.

From here, you must determine the best way to proceed based on the individual circumstance. You might choose to redefine a method or explicitly set the reference to `nil`, but you must somehow remove the reference.

After you have removed all references to the first instance, repeat this process with each other instance.

When all references are removed, you can safely remove all instances. When all instances are removed, you can remove the class. If you are sure you have removed all instances, but you get another error when try to remove the class, execute the expression:

```smalltalk
Smalltalk unusedMemory
```

This expression forces garbage collection.
Overview

Visual Smalltalk applications are typically interactive applications that make extensive use of a graphical user interface (GUI). Behind the graphical interface is a great deal of application logic that manipulates the information used by your business.

Visual Smalltalk makes it easy to develop both the GUI and the application logic separately, and then to join them into a fully functional application. The resulting application is highly maintainable and extensible.

Designing an Object-Oriented Application

It is a fundamental principle of object-oriented design that an application should be composed of encapsulated objects, with each object being responsible for its own behavior and state. Good application design leads to objects, or components, that can be developed and maintained individually, and assembled into a full application.

There are a wide variety of methodologies in use to promote good object-oriented design. Some of these are described in the books and periodicals listed in the Bibliography. For comprehensive discussions of object-oriented design, please refer to these works.

For purposes of our discussion, and to help you get started in the right way, we will briefly describe a few, general principles of good design.

Object Roles

A useful way to think of constituent objects is in terms of their role in the application. Thinking of an object in terms of the role it plays in an application helps clarify its responsibilities.
Two major categories of such roles are user interface and business model. Identifying any given object as an interface object, for example, makes it clear that it should not be burdened with information processing that properly belongs to the business objects.

A slightly more detailed analysis of roles leads to a four layered architecture, with each layer corresponding to a different role for the application objects:

**Four Layer Architecture**

Presentation objects are typically part of the GUI, the collection of windows and data display fields. Their primary responsibility is to interact with the user by displaying information and prompts, and collecting information from the user.

The business objects are primarily responsible for modeling the kind of information and logic to do the processing required by the business. For example, if the data collected from the user constitutes an order, the business object will represent the order as well as the processing required to fill the order and collect payment. In an accounting application, the business logic will involve a great deal of calculation. In a manufacturing application, the business model may specify assembly rules, such as the order in which components are installed. Rules for moving things in and out of stock will be a key part of the business model in a warehousing application.

The server objects represent resources typically provided by an external resource, such as a data base located on a server. The external resource is represented in Visual Smalltalk by “wrapping” it, exposing just enough of its interface to send information to and receive information from the other program.

The application objects can be thought of as standing between these other objects, providing any logic specific to moving data to and from the GUI and business elements, and controlling any work flow required by the application.
Consider a simple random number generator program. We need a user interface that allows the user to enter a seed value and read a generated value. We also need some logic that performs the actual calculation based on the input seed value. And we need some application logic that describes what happens when the user changes the seed value or requests a new random value. If we design different components for each of these levels, the responsibilities of each is clear and yields a good object-oriented design.

More complex applications will have many presentation objects, business objects, server objects, and application objects.

**Component Architecture**

A well-designed object has a clearly defined public interface, consisting of the messages it can receive from other objects and events it can trigger. Some objects are designed for a single, specialized purpose within a given application. Other objects, especially objects that represent essential pieces of business logic and data, can be designed with a very general interface making them easy to reuse in many applications.

The idea behind *component architecture* is that some objects can be designed in such a way as to maximize their reusability. The application development environment, in turn, must then be able to use such components, accessing them through their interfaces for building the larger application.

Visual Smalltalk provides the environment for building applications out of components, as well as the mechanisms for building reusable components. The task of designing and implementing good, reusable components, however, often requires the efforts of a large group of information specialists.

Good components tend to be:

- Fairly large objects, in the sense that they represent items that show up in many contexts.
- “Coarse grained” in their interface definitions. The messages a component answers should be few and flexible.
- Designed specifically to be reused by many applications. As few assumptions as possible should be made about the context in which a component will be used.

As an example, a customer object may make a good component, representing a data structure with specific data items that is used throughout a set of business applications. The interface can
consist of messages to get and set customer information items, restricted to defining its features directly relevant to being a customer. The customer’s name, address, customer number, and other items belong in this component. Additional information, such as outstanding orders placed by this customer, or payment status, do not define anything essential to being a customer, and so belong somewhere else in the system.

**Visual Smalltalk Application Framework**

A good *application framework* helps make you a more productive application programmer by simplifying many of the problems involved in assembling an application. In particular, a good framework should relieve many responsibilities of application startup and shutdown, as well as provide good mechanisms for the many parts of the application to work together. The framework should also allow you to develop your application using the same structure you used in designing the application.

Visual Smalltalk provides an application framework that supports the kind of architecture we have been describing. At the simplest level, the framework allows you to clearly differentiate the presentation layer and the business layer. Traditionally in Smalltalk programming, these have been referred to as the *view* and the *model*, respectively. In Visual Smalltalk, the view is most easily built using the Parts Workbench, while the business model is typically built using Smalltalk code.

The model and view are held together by another interfacing component, called variously a coordinator, a controller, a manager, or other terms, suggesting an intermediary role. In the remainder of this document we will refer to this component as a *coordinator*.

![Diagram](attachment://diagram.png)

The coordinator represents the application logic layer, being responsible for starting up and shutting down the application, and describing the flow of information between the presentation and business logic layers.
The coordinator may also perform other services, such as conditioning any information passed between the model and view. Verifying formats and other processing that may not be appropriate for either the model or the view to be responsible to perform can be done by the coordinator.

Coordinating the model and view can be done either in a workbench or in Smalltalk, depending on what makes the most sense for your application.

Coordination makes extensive use of events and messages. Events essentially broadcast a notification that something has happened, usually a state has changed in the object triggering the event. In response to the event, one object sends a message to another.

Events can be thought of as starting something happening, a sequence of actions in the application, while messages complete the sequence. In a user-driven application, for example, when the user does something in some part of the interface, such as clicking on a button, the application must respond. The response begins by triggering an event, saying in effect “I’ve been clicked.” This sets off a sequence of message sends to accomplish whatever effect clicking that button is designed to do. Some of these messages may themselves trigger events, and so the action can become quite complex.

The Parts Workbench is the easiest coordinator, allowing you to drop view and model objects onto a workbench. Each object on the workbench has a set of events that it can trigger and a set of messages to which it can respond. You connect the objects simply by drawing links from one object’s event to another object’s message.

To use Smalltalk as the coordinator, you create a subclass of a coordinator class, such as the ApplicationCoordinator, in which you define methods that create instances of the model and view and register appropriate event handlers to mediate between them. For more information about using the Application Coordinator, refer to the Visual Smalltalk Enterprise Language Reference.

In the remainder of this chapter, we will use the Contact Manager application from the Visual Smalltalk Enterprise Tutorial to illustrate the design and framework principles just described.
Building the Application View

The basic GUI element is the window consisting of the main window and all of its panes (subwindows), controls (buttons, scroll bars, and so on), and dialogs. The visual nature of the workbench makes it natural for building the view for your application.

With visually oriented, interactive applications, the view is a natural place to begin working on your application. By working with the view, you can think through how the application's user will access and enter information, and you can gain insight into how better to handle the information processing.

The view will consist of one or more windows. A main window will open when the application is launched. It provides some pretty self-evident idea of what the program is doing and what the user should do next. For example, the Contact Manager opens by displaying the window shown below.

This view has been built in workbench, as described in the tutorial, and will not be described further here. It is a fairly complex part, containing standard parts (the push buttons, text pane, and notebook part), as well as additionally constructed parts (the filter panel and phone information panel in the notebook).
Exposing the View Interface

Within the application, we want to deal with this whole window as a single view object, without having to know anything about its internal workings. The interface to the window itself is fairly simple, triggering very few events and responding to very few messages.

Because parts are typically compositions of smaller, constituent parts, with one part being used within another, within another, and so on, exposing the interface can be tricky. The Parts Workbench simplifies the process by providing mechanisms in the Nested Part for exposing interfaces of interior parts. By making the entire part a Nested Part, a single, easy to access and use interface can be exposed.

The Nested Part essentially “passes through” interfaces, relaying messages and events between the underlying part and the external objects.

The interface for the view object identified in the tutorial is:

- **Messages**
  - **open** - to display the window when requested, specifically at the beginning of the application
  - **updateView** - to refresh the display when the display data has changed

- **Events**
  - **close** - signalling when the window is closed
  - **editContact** - signalling when the user clicked the Edit Contact button
  - **newContact** - signalling when the user clicked the New Contact button

To specify the exposed interface, open the properties dialog for the view part, and click the *Interfaces* button. The interfaces dialog opens, allowing you to enter new event and message names for the nested part.
The `open` message is the only interface item provided by default, so we have to add the others. We add events for outgoing interfaces and messages for incoming interfaces. Since these are simply pass-through interfaces, we only add event and messages names; there is no processing to specify.

To add an event, click the `Events` radio button and click `New...`. A dialog prompts you for the name of the new external event. Similarly, to add messages, click the `Messages` radio button and click `New...`. A dialog prompts you for the name of the new external message.

By examining the interfaces list for the FVIEW.PAR part, you can see these interfaces in the interface list.

With the interfaces defined for the part, you can assign the part as the content of the nested part and connect the interfaces. To do this, drag a new Nested Part to a new workbench, and open its property dialog. Click the `Change file...` button. In the file selection dialog, select the part file and click `OK`. Now if you drag a link from the Nested Part, the link dialog lists the events we added to the underlying part's event list. Similarly, if you drag a link to the Nested Part, the link dialog will list the messages we added. This is how the top level workbench of the Contact Manager, CNTCTMGR.PAR, was constructed. The result looks like this:
Building the Business Model

The main work of the application is done in the business model. It describes the objects representing the things your business works with: customers, payroll, processes, documents, whatever. Each object that your business operates on has characteristics and behavior, the information processing required by the application.

Business objects are not typically visual in nature, but highly conceptual. Accordingly, this part of your application is best represented in Smalltalk. Building an application in Smalltalk involves defining classes and methods for those classes.

When you have identified a piece of your business to represent as an object, you need to define a class for the object. An object that communicates with other objects using only messages can be created anywhere in the class hierarchy. Objects that trigger events, which includes any objects that interact with the workbench, should be created as subclasses of EventManager.

Defining Business Objects

The business model for the Contact Manager is contained in seven classes, all defined as subclasses of EventManager:

- EventManager
- Address
- Contact
  - BusinessContact
  - PersonalContact
- ContactHolder
- InitiatorInformation
- PhoneInformation

The EventManager hierarchy provides the mechanisms necessary for registering and triggering events, which provide the main interface between the workbench and Smalltalk objects. Objects that do not need to trigger events can be defined anywhere in the class hierarchy.

Contact is the abstract superclass of all contacts types, and defines the behavior and attributes common to all contacts. BusinessContact and PersonalContact define the specific behavior of these two specialized contacts.

Address instances know and process the mailing address of a contact. Similarly, instances of PhoneInformation know the phone number information of a contact.
InitiatorInformation knows the contact initiation information of a BusinessContact. Instances store information about who initiated the first contact and when, and who is scheduled to initiate the next contact and when, and so on.

Consider the two classes Contact and BusinessContact as examples of the business model. Contact defines a base set of instance variables used by all contacts, firstName, lastName, address, comment, phonInfo, and generatedId, and a set of accessor methods for them. BusinessContact adds three more instance variables, together with accessor methods for them.

Contacts are primarily information holders, and so the accessor methods make up most of their “logic.” In addition, however, contacts trigger events to notify other application components when an information item has changed.

An object must declare what events it can trigger before it can trigger them. Events are declared when a new instance is instantiated by reimplementing the constructEventsTriggered class method. For Contact the method is defined as:

```smalltalk
constructEventsTriggered

"Private - answer the set of events that instances of the receiver can trigger."

|items|

items := self fieldEvents.

^(super constructEventsTriggered

   add: #update:

   addAll: items

   yourself)
```

A constructEventsTriggered method typically invokes the default events list defined in Object and then adds additional events, as shown here. The default method creates the list as a Set and adds the single event changed.

The reimplemented method above then adds the single event update and all the event names listed in the fieldEventsList class method. (The sequence of message sends begins with self fieldEvents.) The events included in fieldEventsList are triggered together in the appropriate circumstances, by sending the message triggerFieldEvents. This message is sent by a coordinator object described in the next section. It is convenient and an aid in maintenance to keep them in a list.
The contact objects are simple objects, and have quite simple interfaces. Other business objects may provide more sophisticated services, including calculations, database access, process or data verification, and so on. More Smalltalk code will be required to perform these services.

Once the business logic has been represented by appropriate business objects, additional methods are required to expose the interfaces for use by the other application components.

**Exposing the Model Interface**

The business model must also expose an interface to allow it to interact with the rest of the application. As far as Visual Smalltalk is concerned, the interface is defined entirely by the messages it can send and receive, and the events it can trigger. There is also a convention that methods labeled “Private” in their comment are not for general use, that is, other objects should not invoke these methods.

The workbench works a little differently, in that more control is available over what interfaces are exposed on the workbench.

As it turns out, the contact objects never interact directly with the workbench, so no additional methods are needed. (Interaction with the workbench is handled through a coordinator object described below.) This is frequently, but not always the case, however, so we will describe how interfaces are exposed to the workbench.

Two methods are used to expose the interface of an object on the workbench: `partMessages` and `partEvents`. The interfaces specified by these methods determine which messages and events are listed in the interfaces dialog for the object when it is dropped on a workbench.

The default implementation of these methods, defined in `Object`, is to list all events, and to list all messages that do not begin with “Private” in their method comment. This makes it easy to use arbitrary Smalltalk objects in a workbench.

You can exercise more precise control by redefining the `partMessages` and `partEvents` methods for your class. In these methods you then specify exactly the messages and events you want to surface in the workbench interfaces dialog. In the Contact Manager application, the `ContactEditManager` class implements a `partMessages` method, and `Query` implements a `partEvents` method. A single class can also implement both or neither of these methods.
A list of messages and a list of events to expose are stored in instances of `PARTSInterfaceList`, as shown in the `partEvents` message in `Query`:

```smalltalk
| items |
items := #( update:).

^PARTSInterfaceList new
  items: items
```

As illustrated here, the easiest way to add events to the list is to create a new instance of `PARTSInterfaceList` and specify the items in an array. The same is done in `partMessages` for `ContactEditManager`.

You can also concatenate lists or append items to lists inherited from the superclass. Refer to the Visual Smalltalk Enterprise Encyclopedia of Classes for additional protocol.

Coordinating the View and Model

The application coordinator uses the events and messages exposed by the view and model components to put it all together. It has the primary responsibility of specifying the objects that make up the application and establishing the communication links between them.

The easiest to use coordinator is the Parts Workbench. A view object typically exchanges information with model objects, either to display or update information as required. If you design the interfaces to business objects well, it is easy to drop them onto a workbench and draw links between their events and messages, just like any other part.

The workbench works as an editable graphic representation of the links between the various application components. The links are static, however, and so you want the workbench to contain objects that exist throughout the application run.
Connecting View and Model Objects

The simplest method of coordinating the view and model is to simply have both objects on the same workbench, and connect them with links. Any business object that triggers events can send a message to any other object on the workbench. Similarly, a view object can send a message to a business object on the workbench.

This is illustrated in the Contact Manager application by the nested part containing the filter panel of the main view part. Open the properties dialog for this panel and click on \textit{Edit contents}... . The workbench that opens shows the panel constructed in a window and a \texttt{Query} object on the workbench.

The \texttt{Query} object is essentially a collector and holder of conditions on a contact database query, and of the query results. Its conditions relate directly to the controls in the filter panel, as you can see by displaying the links. Coordination between the filters pane and this piece of the model logic is completely defined in the workbench by these links.

It is easy to place an instance of a Smalltalk object on the workbench. Simply select the \textit{Developer / Add object} menu item, and select the class of the object you want to add. Refer to the \textit{Visual Smalltalk Enterprise Workbench User's Guide} for additional information about adding Smalltalk objects to a workbench.

Simple Controllers

A common and important element of the application logic consists of Smalltalk objects that move information between the view and the model, possibly performing some processing on the object. These objects, referred to as controller objects or controllers, are particularly important for business objects that do not themselves appear on the workbench.

Introducing controllers helps keep application-specific logic separate from either the view or the model objects. Especially in the case of business objects, this separation helps to generalize the objects for reuse in other applications.

The \texttt{ContactQueryController} is a good example of one of these controller objects. It coordinates both input and display in the main view and accesses the database interface (\texttt{ContactDatabaseInterface}) according to conditions set in the \texttt{Query} object.
An instance of **ContactQueryController** is placed on the workbench and linked to the relevant events and messages of other objects on the workbench. In Smalltalk, it stores an instance of the database interface, and exchanges information at that level.

### Representing Dynamic Objects

Dynamic objects, such as a customer that might be added or deleted, or simply focused on and then focused away from, should not be directly added to the workbench. Instead, a **holder** for that object should be defined, and the holder can then take individuals as its value as required while the application is running.

In the Contact Manager application, this is exactly the situation. Contacts, like customers, are kept in a database. The application does not have a single contact object which can be dropped on the workbench, but must be able to work with any number of contact objects, one at a time.

To do this, we use a **ContactHolder** object, which acts as a container for transient contact objects. The single instance of **ContactHolder** placed on the workbench is itself a static object, but its contents change as we have different contacts in view. Links are drawn to the contact holder object, which provides a set of standard interfaces relevant to contact objects. The holder then has the responsibility for exchanging messages with the contained contact object to achieve the necessary work.

The interesting work done by the holder is in connecting the events triggered by the application and by the contact object. In its **partEvents** method, **ContactHolder** surfaces all events triggered by all subclasses of **Contact**. It also identifies these events as events it can trigger, in the **constructEventsTriggered** method. The declared events are used so that links can be created on the workbench in the absence of a “real” contact object.

When a contact object is identified, either by calling it from the database or by creating a new one, it is assigned to the contact holder's **contact** variable. (Actually, a copy of the contact object is assigned.) The contact object contains the logic for triggering the events surfaced by the contact holder. The connection between these is established in the **initializeContactEvents** method, where the handler defined for an event in the contact holder is reassigned to the event triggered by the contact itself.
The logic implemented for **ContactHolder** is taken directly from the logic implemented for **PARTSValueHolderPart** and its subclasses. You can look at these classes for additional examples of handling dynamic objects in the workbench.
Delivering a Visual Smalltalk Application

Overview

Once you have completed your application, you need to prepare it for delivery. During development, you created your application as a collection of parts and Smalltalk code, with the code stored in either packages or Smalltalk libraries. Now you need to prepare the application to run stand-alone, apart from the Visual Smalltalk development environment.

Your delivered application will consist of a set of files, including:

- A single executable program file (.EXE) to launch the application
- One or more executable part files (.PAX)
- One or more Smalltalk library files (.SLL)
- A single bind file (.BND)
- A collection of run-time support files

In this chapter, we break down the process of preparing your application into three primary tasks:

1. Preparing the application to run as a stand-alone program:
2. Preparing application run-time files:
3. Packaging the run-time files:

**NOTE:** The License Agreement governs distribution of Visual Smalltalk applications. Be sure to read this agreement, which is printed and provided with the distribution media, before distributing any applications. This will help you to ensure that your application complies with Cincom’s licensing requirements.
Preparing a Stand-alone Application

There are a few tasks that an application must perform when running stand-alone that it does not do in the development environment. The most important of these is handling its own start up. These tasks mainly relate to polishing the presentation of your application, but they have real value to your customers.

Starting Your Application

When your application is started by executing the Smalltalk image file, V.EXE, the SessionModel triggers the `startUpApplication` event. You need to register a handler for this event that will invoke your application code.

In Visual Smalltalk Enterprise, you can register the handler by adding an initializer such as the following to one of your packages:

```
SessionModel current
  when: #startUpApplication
  send: #open
  to: MyApp
```

In Visual Smalltalk you can register the handler using a bind action for a Smalltalk library. The library must be bound to the image during startup, not statically bound. There are a variety of ways to do this. A flexible approach is to build a `bound` class method in your application class that registers the handler:

```
bound
  "Register handler for #startUpApplication event"

SessionModel current
  when: #startUpApplication
  send: #open
  to: self .
```

You can add additional startup processing to this method also. Then include this message as the bind action for your SLL, using an expression like this in the build script:

```
mySLL bindAction:
  ( MyApp class compiledMethodAt: #bound )
```

If you do not have a library that binds at startup, you can edit the `SessionModel startUpApplication` method, and directly register the handler there. To avoid having to make the change every time you save your image, you can add this to your library as a loose method.
Preparing a Stand-alone Application

Creating a Shutdown Dialog

You may want to provide a shutdown dialog for your application, to verify that the user wants to quit before exiting the application. To display a shutdown dialog, define a close: method in the application framework subclass for your application.

The following close: method displays a simple confirmation dialog:

```smalltalk
close: aTopPane
"Private - Close the receiver"

Smalltalk isRuntime
ifTrue:
  [( MessageBox confirm:
     'Are you sure you want to exit?' )
    ifTrue:
      [ ^Smalltalk exit ]
    ifFalse: [ aTopPane abortClose. ^self ]]
ifFalse: [ ^self close ]
```

Creating a Splash Screen

When you load Visual Smalltalk, the system looks for a bitmap file (.BMP) with the same name as the runtime .EXE and, if it exists, displays the bitmap as the startup screen.

If you add a startup screen, this copyright notice must be included:

```
Portions copyright (c) Cincom Systems, Inc. 2000, All rights reserved.
```

Adding a Sign-on Dialog

As an alternative to creating a bitmap, you can create a DLL with code to open and close your dialog. This DLL must contain two methods to examine the file VWSIGNON.C in the SAMPLE\SIGNON directory:

```
OpenStartupDialog()
    Displays the runtime sign-on dialog

CloseStartupDialog()
    Closes your runtime dialog
```
The name of the DLL must be the same as the name of your executable. For example, if you rename the image from V.EXE to MYAPP.EXE, your sign-on DLL must be called MYAPP.DLL. If the runtime application finds a Smalltalk library with the same name as the executable during startup and does not find a corresponding bitmap file, it will open that DLL and call the two exported functions to open and close your sign-on dialog.

The following Cincom copyright notice must be included in the sign-on dialog of any application developed using Visual Smalltalk:

Portions copyright (c) Cincom Systems, Inc. 1995-2000, All rights reserved.

Capturing Command Line Arguments

Your Visual Smalltalk application can read and process command line arguments by using the `getCommandLine` method in class `SessionModel`. This method returns an array containing the contents of the command line.

To make use of command line arguments, you must edit the `startUpApplication` method in `SessionModel`, and remove the default command line handling.

Then either add this method as a loose method to one of your application packages or libraries, or save the image with this change made.

Changing Application Icons

Two icons are commonly used to represent an application on the desktop: the application icon and the minimize icon. If a minimize icon is not defined, the application icon is used for both.

Icons can be created using any icon editor for the appropriate operating system platform, saved in an ICO file or a DLL resource file. Visual Smalltalk includes an icon editor, packaged as a Smalltalk library, in the SAMPLE\ICONEDIT directory.

Setting the Application Icon

A default Visual Smalltalk icon is provided in the V.EXE image file. You probably want to replace the icon with one for your application.
Preparing Application Run-time Files

To assign an icon to your application:
1. Create the icon and save it as a .ICO file or in a resource .DLL file.
2. Specify the icon in the *Save Image As...* dialog when you save the image.

Refer to “Building the Application Image File” later in this chapter for more information on saving the image.

Setting the Minimize Icon
To set the minimize icon, send the `icon:` message to your application top pane in the `open` or `openOn:` method. If the icon is in an .ICO file, send:

```smalltalk
self icon: (Icon fromFile: 'myicon.ico')
```

If it is in a resource .DLL, send:

```smalltalk
self icon: (Icon fromModule: 'mydll' id: 'myicon')
```

Icons can be specified for additional views by sending similar messages to the view.

Preparing Application Run-time Files

While working in the development environment, your application consists of Smalltalk code stored in packages and various part files. Once you have the application running correctly, you need to build these files into executable parts and Smalltalk libraries.

Creating Executable Parts

To distribute any parts required by your application, you must convert them to executable parts.

To convert your parts to executable parts:
1. Open the application part in a workbench.
2. In the workbench menu, select `File/Save as...` and save as a .PAX file.

A file dialog opens for you to select the file name and type.
3. Repeat steps 1 and 2 for each part used in the application.

When a part is saved as an executable part, it includes any nested parts as well as parts directly used on the workbench. So, you do not need, for example, to save separately any buttons or window panes used in the workbench, nor any parts built as nested parts.
CHAPTER 5

Delivering a Visual Smalltalk Application

On the other hand, any parts referenced by a Part Accessor part must be saved as separate executable part files. Open each part referenced by a Part Accessor part and save it as a .PAX file as well.

Building Application Smalltalk Libraries

Smalltalk libraries will contain all the code defining your application, including classes, methods, globals, and so on.

In Visual Smalltalk, you have to write a script to build each library. A simple application will require only a single library to which you add all the classes you have defined. You can also add other objects, such as individual methods. A simple script can begin with the following:

```smalltalk
sll := SmalltalkLibraryBuilder new: 'myapp.sll'.
sll add: ; " include class names here "
add: ;
windowFeedback;
writeFile.
```

On the `add:` lines, include the name of a class. Additional protocol for including other objects is described in the Smalltalk/V Programming Reference.

In Visual Smalltalk Enterprise, you build Smalltalk libraries directly from the packages you have already created. This step is simplified by the fact that your application is already contained in one or more modules.

If all of your application code is in a single package, you will produce a single library. If your application is in multiple packages you can build it into a single Smalltalk library by putting the packages in one cluster, and then building the Smalltalk library from that cluster.

To build your application as several Smalltalk libraries, you can build from each package separately, or group the packages as desired into several cluster, and build each cluster separately.

To build a Smalltalk library from a module:

1. In the Package Browser, load the module(s) containing your application.

2. Select a module, then select Module / Build Library... .
   A dialog opens displaying the description of the current module.

3. Click Customize... to edit the library build script.
Preparing Application Run-time Files

Here are a few lines you may want to edit.

- **Author:**
  Replace "editThisName" with your name, as a string.

- **info name:**
  Check the default file name, and change if necessary.

- **info targetDirectory:**
  The default directory is the Visual Smalltalk Enterprise root directory. If you are collecting files in a different directory, place that directory name here.

- **info sourceInFile**
  The default is to include source in the library file. If you want to write the source into a separate file, edit this line to `info sourceSeparate`.

You may want to make other changes to the build script as well.

4  Select the entire build script and evaluate it using Do It.

A text window opens and displays progress messages as the library is being built. The text will indicate either that the library was successfully built, or that an error occurred causing the build to fail.

**Collecting Run-time Support Files**

So far your application directory contains executable part files and the application Smalltalk libraries. Other files are required to support the application.

Several files are always required by a Visual Smalltalk application. The following files are platform specific:

<table>
<thead>
<tr>
<th>Windows</th>
<th>OS/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBAS\nnW.SLL</td>
<td>VBAS\nnO.SLL</td>
</tr>
<tr>
<td>VVM\nnW.DLL</td>
<td>VVM\nnO.DLL</td>
</tr>
<tr>
<td>PWWRBX\nn.SLL</td>
<td>PWORBX\nn.SLL</td>
</tr>
<tr>
<td>PWWHST\nn.SLL</td>
<td>PWOHST\nn.SLL</td>
</tr>
<tr>
<td>PWWRUN\nn.SLL</td>
<td>PWORUN\nn.SLL</td>
</tr>
<tr>
<td>VCOFF\nnW.SLL</td>
<td>VCOFF\nnO.SLL</td>
</tr>
<tr>
<td>VOFLR\nnW.SLL</td>
<td>VOFLR\nnO.SLL</td>
</tr>
<tr>
<td>VOFSP\nnW.SLL</td>
<td>VOFSP\nnO.SLL</td>
</tr>
</tbody>
</table>
The following files are common to all platforms:

- PWCBASnn.SLL
- PWCBVRnn.SLL
- PWCRBSnn.SLL
- VODMPnn.SLL
- VOFSCnn.SLL
- VOSWnn.SLL

You must also include the following two .MAP files, which provide logical file name mappings to system resources:

- V.MAP
- VNOGUI.MAP

Depending on the optional resources you used in your application, you may also need to copy additional files to your application directory. The following table lists some of the most common additional files. File names for further libraries you may have used can be determined by examining the description of the particular option in the Service Manager.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWCDARnn.SLL</td>
<td>BtrieveAccessor part, the DLLAccessor part, or the CStructure part.</td>
</tr>
<tr>
<td>PWCDRNn.SLL</td>
<td>DDE parts</td>
</tr>
<tr>
<td>PWCSBRnn.SLL</td>
<td>ScrollBar parts</td>
</tr>
<tr>
<td>PWCTPRnn.SLL</td>
<td>TablePane part</td>
</tr>
<tr>
<td>PWpNBKnn.SLL</td>
<td>Notebook parts</td>
</tr>
<tr>
<td>PWpNPRnn.SLL</td>
<td>Notebook parts</td>
</tr>
<tr>
<td>PWpRESnn.DLL</td>
<td>Notebook parts (Win32S)</td>
</tr>
<tr>
<td>PWpRWRnn.SLL</td>
<td>Report Writer parts</td>
</tr>
<tr>
<td>PWpSPKnn.DLL</td>
<td>Speaker part on Windows 3.1 (Win32S)</td>
</tr>
</tbody>
</table>
Preparing Application Run-time Files

If you have used any Parts Wrappers in your application, you need to include the runtime files for these wrappers as well. Refer to the application delivery section of the user guide for each wrapper for a list of additional files to include.

### Bind Options for Smalltalk Libraries

When building the application image file, you will have to know which Smalltalk libraries you want to bind statically to the image, which you want to bind when the application is loading, and which the application dynamically binds and unbinds.

Advantages of each binding method are discussed in the *Visual Smalltalk Enterprise Language Reference*. Briefly,

- Statically bound libraries load faster, but increase the image size and cannot be updated individually.
- Load-time bound libraries reduce the image size and can be updated individually, but load more slowly.
- Dynamically bound and unbound libraries reduce image size, and are only bound when needed, but make the application responsible for binding and unbinding.

Your final decision may require some testing, to find a combination to best meet user requirements.

### Selecting Libraries to Statically Bind

Based on the advantages and disadvantages of static and non-static bindings, the following suggestions may help:

**Bind:**

- Any library containing startup code
- Any library containing core application logic

**Do not bind:**

- Libraries that your application dynamically binds and unbinds
- Libraries that you want to update without replacing the image

If these guidelines conflict, you need to balance the advantages and disadvantages in making your decision.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDDEnlp.SLL</td>
<td>DDE parts</td>
</tr>
<tr>
<td>WBTRCALL.DLL</td>
<td>BtrieveAccessor part</td>
</tr>
</tbody>
</table>

If you have used any Parts Wrappers in your application, you need to include the runtime files for these wrappers as well. Refer to the application delivery section of the user guide for each wrapper for a list of additional files to include.
Create the Bind File

If you have Smalltalk libraries that you did not bind to the image and that are not dynamically bound by the application, they need to be bound to the application during startup. To do this, you create a bind file for the libraries.

The bind file is a plain text file containing the names of the libraries to bind at startup. Each library name occurs on a separate line. Any Smalltalk library can be bound by this file, including the Visual Smalltalk runtime support libraries.

The file name must be the same as the application executable, but with a “.BND” extension.

You can reference another bind file within your application bind file by putting the name of the other file on a line preceded by an “@” sign. The “@” sign must be the first character on the line, with no space between the sign and the bind file name.

Building the Application Image File

While you can use the clean runtime image file (V.EXE) and dynamically bind all SLLs to it during startup, your application will start up faster if you statically bind at least the SLL(s) containing the main program logic. You need to build the startup image file from a clean image to make sure all development resources and non-essential code are removed.

To build the image file, do the following:

1. Create a new directory to contain your application.
2. Copy a clean V.EXE into the application directory.
   A clean V.EXE was copied into the VBACKUP directory during installation.
3. Copy the following development files to your application directory:

<table>
<thead>
<tr>
<th>Windows</th>
<th>OS/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRESnW.DLL</td>
<td>VRESnW.DLL</td>
</tr>
<tr>
<td>VDEWEXE</td>
<td>VDEVO.EXE</td>
</tr>
<tr>
<td>VDEV3AW.SLL</td>
<td>VDEV3AO.SLL</td>
</tr>
</tbody>
</table>

4. Copy your application Smalltalk Libraries to the application directory.
5 Copy the Visual Smalltalk development image file to the application directory.

You need to copy VDEVW.EXE for Windows, or VDEVO.EXE for OS/2. You do not need any other development files for this procedure. In particular, you do not need any additional workbench files.

6 In the application directory, start the development image.

Run VDEVW on Windows platforms, VDEVO on OS/2.

7 Bind the Parts Workbench runtime files to the image.

Select File/Install, and choose PWWRUNnn.SLL (Windows) or PWORUNnn.SLL (OS/2) in the file dialog. This library file binds all of the standard workbench runtimes.

8 Bind any optional workbench runtime files required by your application.

9 Bind your application specific Smalltalk libraries to the image.

You can use File/Install in the transcript to select and bind each Smalltalk Library.

You can bind all of your libraries, or bind only those that are needed at startup. Refer to “Selecting Libraries to Slightly Bind” earlier in this chapter for guidelines on selecting libraries.

*Do not bind* any libraries that your application dynamically binds and unbinds.

10 Select File / Save Image as ... to save the image.

The Save Image As dialog box opens. This dialog gives you the option of setting the application icon and modifying the application startup method.

11 Enter the application name.

Replace “V.EXE” with a name for the application executable file. The name should be something meaningful, suggesting the application title. The extension must be “.EXE”.

12 Set the application icon.

If you have created an application icon, as described earlier in this chapter in “Changing Application Icons,” click the Change Icon button. In the file dialog, select your icon, then click OK.
13 Save the image and exit Visual Smalltalk.

Click OK to save the image. Once the image has been saved, exit Visual Smalltalk. You do not need to save the image again.

The resulting executable is your application image file.

Building a Workbench Executable

In a few situations, specifically if your application is designed to launch from a workbench, a simpler approach to building the runtime application is available. Using this approach you build the executable image directly in the workbench.

While this method for creating the executable application image is simpler than the standard method, you do not have as much control over your application, for example in such areas as startup and shutdown processing.

The general application delivery strategy remains the same. You still need to build Smalltalk libraries for your application code, as described above. You will deliver a Smalltalk image file and a bind file, but these are automatically generated for you.

The files produced are:

- a “clean” image file, consisting of the original V.EXE renamed for your application
- the main part file saved as a .PAX file
- a bind file including minimally required runtime SLLs and PWCSSTR31.SLL, which contains only a bind action to load and run your application part.

To build the image file:

1 Start the development system.

2 Bind all application runtime SLLs to the image.

   You must have built these already as described earlier in this chapter.

3 In a workbench, open the part file that is the starting point for your application.

4 Select File / Save As... in the workbench menu bar.

5 In the file dialog, select .EXE as the file type.

6 Select your application directory and name the file.

   The name will become the file name used to start the application.
As stated above, this saves the part as a .PAX file, creates a bind file (.BND), and saves the Smalltalk image file (.EXE), all with the name of your application.

The PWCTR31.SLL file loads and launches the part, so you don’t need to create a startup method in your application SLL.

All other instructions in this chapter still apply. For example, you can provide a startup bitmap.

### A Simple Example

For a quick example, we will prepare the Contact Manager example as a stand-alone, runtime application. The Contact Manager is already contained in a Smalltalk library and several part files.

**To prepare and package Contact Manager:**

1. Create a directory for the application.
   
   For this example we’ll use the directory C:\CNTCTMGR. This is where we’ll collect the pieces we need for our application.

2. Copy and rename a clean image file to your runtime application name.
   
   Since the application will be completely contained in a Smalltalk library, we can use the original image file. You should have a copy in the VBACKUP subdirectory.

3. Copy the Visual Smalltalk runtime support files to the application directory.
   
   Copy the required runtime files listed in “Collecting Run-time Support Files” from the Visual Smalltalk LIB directory. Also copy the NoteBook support files.

4. Copy the Contact Manager SLL.
   
   The library file is LIB\FINCONMA.SLL.

5. Start your Visual Smalltalk development system.

6. Load and save the application part files as .PAX files in the application directory. Ensure the directory TUTORIAL\PARTS is in your PARTS Settings search path.

   You will need to bind FINCONMA.SLL, or load the package. Then load each of the following parts and save as .PAX files:

7. Click OK.

   As stated above, this saves the part as a .PAX file, creates a bind file (.BND), and saves the Smalltalk image file (.EXE), all with the name of your application.
Delivering a Visual Smalltalk Application

For purposes of this example we need to build a new class and SLL to handle starting the application.

Visual Smalltalk applications typically are started by sending a message, such as `open`, to the main application class. The Contact Manager differs slightly in this respect, because it was designed to launch from the workbench. This is not unusual, especially for relatively simple applications, and actually makes Contact Manager a good candidate for building the image directly from the workbench.

However, to illustrate the more typical application startup preparation, we will emulate the normal behavior in a new `ContactManager` class. Normally you would include corresponding methods in your application class.

7 Create a `ContactManager` class as a subclass of `Object` with the following methods.

We need two methods, one to register an event handler for the `startUpApplication` method, and one to launch the main part file, FCNTCTMG.PAX. These methods are both implemented as class methods.

The `bound` message will be sent when the startup SLL is bound.

```
bound
    SessionModel current
    when: #startUpApplication
    send: #open
to: ContactManager.
```

The `open` message will be sent when `startUpApplication` is triggered.

```
open
    (PARTSAplicationHolder on: 'c:\contactmanager\fcntctmg.pax')
    openApplication.
```

8 Build a new library for the `ContactManager` class.

This simple script will create the required Smalltalk library. Notice the bind action, which registers the start up event handler.
**Preparing Application Run-time Files**

**SmalltalkLibraryBinder bindTo: 'vslb31'**.

```smalltalk
sll := SmalltalkLibraryBuilder new:
    'c:\cntctmgr\CMbind.sll'.
sll add: ContactManager;
    bindAction:
        (Message receiver: ContactManager
selector: #bound);
    windowFeedback;
    writeFile.
SmalltalkLibraryBinder unBind: 'vslb31'.
```

In Visual Smalltalk Enterprise, you would probably build the `ContactManager` class in a package. Then include as an initializer:

**ContactManager bound**

When you build the library using *Module / Build Library...* the bind action is defined for you.

**9 Build the application bind file.**

In a workspace, create the following bind file and save it as CNTCTMGR.BND in the application directory. The bind file for Windows systems should be:

```plaintext
@PWBRUNW.BND ; Parts runtime support
PWWNBK31 ; NoteBook part support
PWWNPR31 ; NoteBook part support
FINCONMA ; Contact Manager support library
CMBIND ; ContactManager startup library
```

For OS/2, the first three file names are PWBRUNO.BND, PWONBK31, and PWONPR31.

The files CNTCTMGR.BND and PWBRUNW.BND (for Windows) or PWBRUNO.BND (for OS/2) should be in the application startup directory.

**10 Exit Visual Smalltalk.**

When prompted whether to save the image, click *No*.

The application is now complete. The CNTCTMGR directory should contain the Visual Smalltalk runtime support files, plus:

- CNTCTMGR.EXE
- CNTCTMGR.BND
- FINCONMA.SLL
- CMBIND.SLL
- FCNTCTMG.PAX
- FEDIT.PAX
• FEDITC.PAX
• FVIEW.PAX

Use the usual methods to select and run CNTCTMGR.EXE. You can add an icon to the desktop to make it easier to start. You can also enhance this application by adding a bitmap splash screen as described earlier in this chapter.

Packaging the Run-time Files

At this point your application directory contains nearly everything required by your application, and very little that is not required. Notably, you should delete these files from the directory:

<table>
<thead>
<tr>
<th>Windows</th>
<th>OS/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>VRES/nW.DLL</td>
<td>VRES/n0.DLL</td>
</tr>
<tr>
<td>VDEVW.EXE</td>
<td>VDEVO.EXE</td>
</tr>
<tr>
<td>VDEV3AW.SLL</td>
<td>VDEV3AO.SLL</td>
</tr>
</tbody>
</table>

These should be the only development files that you copied to the application directory.

You must also copy to the application directory any additional resource files required by your application, such as startup and shutdown resource files and/or bitmaps:

• icons
• startup logo bitmap
• auto-bind SLL text file
• Smalltalk library name map file
• DLLs you have created and accessed from within the application
• DLLs from other vendors that you have used (verify that their licensing agreements have been satisfied)

You can then distribute the files remaining in this directory.
Overview

As noted in earlier chapters, you can place an instance of a Smalltalk class on the workbench and use it like a part. There are ways of improving the object definition, however, to improve its behavior on the workbench. This chapter and the next describe these improvements. The main difference is in preparing an object specifically to interact with other objects in the workbench.

The steps required to create a new part in Smalltalk are:

1. Create a part class in Smalltalk that implements the behavior of the new part.

2. Use the workbench to create a property edit dialog for the part.

3. Implement a property dialog class that interfaces with the property edit dialog part.

4. Build Smalltalk libraries separating runtime and workbench support classes.

If you want to add the part to a catalog, you must also do:

5. Create a .PAR part file containing an instance of this class.

If you already have a class definition, but want to make it more usable as a part, you can do so simply by adding the methods required to support the workbench editing and execution behavior, and build a part file using this modified class.
Random Number Generator Example

The techniques discussed in this section are illustrated in a simple random number generator example. The end product is a non-visual part suitable for adding to a part catalog. The files included in the example are:

- **RANDOM.SLL** Contains the code for Random Number Generator.
- **RANDOM.PAR** Contains the completed random number generator part, suitable for adding to a catalog.
- **RANDOM.ICO** Contains the icon for the random number generator part.
- **PPRANDOM.PAR** Contains the property dialog for the random number generator part.

To load the part files and run the example, you must install:

- Property Sheets, in the Services Manager
- the Smalltalk library, PWCPMW31.SLL

Creating a Part Class

You begin defining a part like you do any Smalltalk object, by defining the classes and methods that describe the part’s state and behavior. Whether the part is visual or non-visual determines where in the Smalltalk class hierarchy you should define the part classes.

For a nonvisual part, define your part class as a subclass of `EventManager`. This superclass provides the essential event table support needed by the event system, without having to replicate all that behavior in your class.

For a visual part, such as a new kind of window pane, define the part class as a subclass of `Window`. This allows your part to inherit some behavior it needs to be a visual part, and also provides the event handling support.

Once you have created a class for the part, define its behavior as usual, by defining methods.
Remember to define methods that trigger events to notify other objects of changes in the part. For instance, if the part stores data values, it should trigger an event any time the value of a data item is changed. A visual part should also trigger an event any time the user acts on the part.

The rest of this section discusses additional methods necessary to make the object work as a part and interact with the workbench.

**Exposing the Object’s Interface**

An object’s interface on a workbench consists of the events that it triggers and the messages that it responds to. Its public interface is usually a subset of its entire interface.

We want an object, when placed onto the workbench, to have its public interface exposed, but to leave its private interface hidden. We do this by creating two instance methods for the object: `partMessages` for the messages; `partEvents` for the events.

Both methods have default implementations defined in `Object`. The default implementation of `partMessages` returns an alphabetical list of all instance messages defined for the object whose comment does not start with “Private”. The default implementation of `partEvents` returns a list of all events the object can trigger.

Since the default implementations include all messages and events, you may want to reimplement the methods for your object, identifying exactly the messages and events you want the object to support in the workbench.

The messages `partMessages` and `partEvents` each create an instance of `PARTSInterfaceList` specifying, respectively, a list of messages and list of events. The interface list can also specify separator lines for the displayed list, and a default item.

Set the contents of a new `PARTSInterfaceList` using one of the following messages, depending on how many options you are specifying:

```plaintext
items: symbolArray
items: symbolArray
defaultItem: aSymbol
items: symbolArray
separators: IntegerArray
defaultItem: aSymbol
```

You should always specify a default choice.
For example, the random number generator part uses the message `next` to access the next value and the messages `lowerBound`, `setLowerBound`, `upperBound`, `setUpperBound`, `seed`, and `setSeed` to access and set the properties of the generator.

The `partMessages` method for the random number generator would be:

```smalltalk
partMessages
    " Answer the PARTS message interface of the receiver. "
    ^PARTSInterfaceList new
        items: #( next
            lowerBound setLowerBound:
            upperBound setUpperBound:
            seed setSeed: )
        separators: #( 1 5 )
        defaultItem: #next.
```

Since `next` is the most frequently used message, it is made the default. The separator lines are added to give some visual organization to the list.

The `partEvents` message is implemented the same way. For example, if the random number generator triggers the `next:` event whenever it receives the `next` message, you would implement `partEvents` as follows:

```smalltalk
partEvents
    " Answer the PARTS event interface of the receiver. "
    ^PARTSInterfaceList new
        items: #( next: )
        defaultItem: #next:
```

When there is only one item in the list, it should be designated as the default.

Rather than simply create a new list, you may want to modify the list generated by your class’s superclass. These messages create the list by concatenation:

```smalltalk
concatenateLists: anArrayOfInterfaceLists defaultItem: anItem
    This message creates an interface list by adding messages to
    the parts interface list returned by a superclass.

concatenateListsSeparated: anArrayOfInterfaceLists defaultItem: anItem
    This message concatenates the interface lists and separates
    each sublist with a line.
```
You can convert an inherited list to an ordered collection and modify it using that protocol. This is useful for removing as well as adding messages. For example:

```smalltalk
partMessages |
    standardMessages items |
    " get superclass partMessages "
standardMessages := super partMessages.
items := (standardMessages items
    asOrderedCollection
    " remove unwanted messages "
    remove: #add: ;
    remove: #addAll: ;
    " add new messages "
    addAll: #( at: at:put: includesKey: );
    yourself)
    asSortedCollection asArray.
" create new list "
\PARTSInterfaceList new
    items: items
    defaultItem: standardMessages defaultItem
```

### Enabling Cloning for a Part

A new part can be created by “cloning” an existing part, creating a new instance of the object with its full state preserved. This is done when a top level part is tested or saved to a file, or when a part is copied to the clipboard. A part is cloned by sending the message `partMirrorCopy:` to it.

Unlike `shallowCopy` and `deepCopy`, the mirror copy operation does a recursive copy, copying all of an object’s constituent objects. It copies the receiver itself, then copies the objects the receiver points to, the objects those objects point to, and so on until all objects are included.

The default implementation of `partMirrorCopyReal:` in class `Object` copies all the data in all the objects that can be reached from the receiver object, and is usually sufficient for nonvisual parts.

Reimplement `partMirrorCopyReal:` if you need to:

- selectively omit items that would otherwise be copied, or
- include data that is not copied by the superclass implementation.
For example, you must reimplement `partMirrorCopyReal:` in a visual part that has added instance variables. For a visual part, or for a nonvisual part that uses system resources, you need to reimplement `partMirrorCopyReal:` to filter some of the data.

This is important, for example, for any part that holds a system window or resource handle, since copying the handle doesn’t create a new copy of the system resource. So, for a subpane, `partMirrorCopyReal:` must be reimplemented to drop the subpane’s window handles.

Cloning a part is carried out by two methods that work together.

`partMirrorCopy: aPARTSMirrorCopyDictionary`

Looks up the receiver in `aPARTSMirrorCopyDictionary`, and either sends `partMirrorCopyReal:` to the receiver if it is absent from the dictionary, or simply returns the dictionary if the receiver is already present.

`partMirrorCopyReal: aPARTSMirrorCopyDictionary`

Performs the actual recursive copy operation by calling `partMirrorCopy: on each of its sub-objects.`

If you do need to modify the way your object is copied, you only need to reimplement `partMirrorCopyReal:`.

First send `super partMirrorCopyReal:` and then copy the rest needed by this class. For example, `PARTSDialPanePart` defines an instance variable, `dial`, and adds it to `partMirrorCopyReal:` like this:

```plaintext
partMirrorCopyReal: aDictionary
" Private - add a mirror copy of the receiver to aDictionary. Answer the copy. "
| copy |
copy := super partMirrorCopyReal: aDictionary.
copy dial: (dial partMirrorCopy: aDictionary).
^copy
```

Other examples exist throughout the system.

## Creating the Property Dialog

Most parts have properties that can be or need to be set to fully specify the part’s behavior on the workbench. These properties are set using the part’s property edit dialog.

Visual Smalltalk provides default property edit dialogs, which provide editing for basic properties. The default property dialogs are `PPVSLPRT.PAR` and `PPNNVPRT.PAR`, shown below. These are
the dialogs displayed for a visual part and a nonvisual part, respectively, if you open the property edit dialog for part without a custom dialog.

Creating a Custom Property Edit Dialog

If your part has more properties to set than those provided by the default dialogs, you need to create a custom property edit dialog. Visual Smalltalk provides a part that does most of the work for you. The part uses the Property Manager capability in Visual Smalltalk.

The procedure requires defining two new methods in your part class: the **constructPropertyManager** class method, and the **partEditProperties** instance method to identify the property editor. Then you use the Property Interface part to generate the custom property edit dialog.

The resulting dialog requires the Tab control on Windows platforms, or the Notebook control on OS/2. Make sure the appropriate control is installed in the image.

The **partEditProperties** method is simple:

```smalltalk
partEditProperties

" Open an editor on the PARTS properties of the receiver. "

( self propertyManager ) editFor: self
```
The `constructPropertyManager` class method defines the contents of the property edit dialog by adding properties to, and removing properties from, the default property manager, defined in `Object`. The method for the Random Number Generator is:

```

constructPropertyManager
"Construct accessors for receiver's properties."
^super constructPropertyManager
    add:
        (SelectorPropertyAccessor new
            name: 'Seed';
            get: #seed;
            set: #seed;;
            format: 'Integer');
    add:
        (SelectorPropertyAccessor new
            name: 'Lower Bound';
            get: #lowerBound;
            set: #lowerBound;;
            format: 'Float');
    add:
        (SelectorPropertyAccessor new
            name: 'Upper Bound';
            get: #upperBound;
            set: #upperBound;;
            format: 'Float');
    addGroupValidator: #validateValues;;
    addNames: #('Seed' 'Lower Bound' 'Upper Bound')
    toPage: 'Settings';
    addGroupEditor: 'pprandom.par';
    yourself
```

A `PropertyManager` is implemented as a subclass of `Dictionary`, and stores instances of subclasses of `PropertyAccessor`, a kind of `Association`. Since we're adding properties, the property accessor we want is an instance of `SelectorPropertyAccessor`. The name assigned will show up in the property edit dialog. The `get:` and `set:` messages identify the property accessor methods defined in the part class.

The `format:` message specifies the format of the value expected for the property. The supported formats are defined in the `PropertyManager supportedFormats` class method, which specifies the part to use to accept the value from the user. Examine this method to see what formats are currently supported.
addGroupValidator: identifies the validator method for the properties, if there is one. For Random Number Generator we have implemented the validateValues: method to do this. This method is invoked, with the value to be validated, by the property manager when properties are set in the edit dialog.

The property edit dialog may have more than one page, with each page indicated by a tab. The addNames:toPage: message specifies the page should contain the property fields. If the page already exists in the property dialog, defined in a superclass method, these fields are added to it. If the page does not already exist, the page is added. More than one of these messages may be necessary to place properties on appropriate pages.

addGroupEditor: specifies the name of the part file containing the property edit dialog. We will create this below.

Creating the Property Editor Part

We finish building the property edit dialog in the workbench. To do this:

1. Open a new Parts Workbench.
2. Drag a PropertyInterface part onto the workbench, and open its property edit dialog.
3. Specify the object whose properties are set by the new dialog.
   In the Object Expression: field, enter the instance creation expression for your part. For Random Number Generator it is: RandomNumberGenerator new.
4. Mark the Generate property dialog check box.
5. Click OK.

The new property dialog is generated by the Property Interface part, which uses the constructPropertyManager method to identify the dialog characteristics you have specified. It adds property fields to the dialog and draws the necessary links to validate and set the properties.

The resulting workbench for Random Number Generator looks like this:
6  Save the workbench as a part file.

The new part file is the property edit dialog for your part. When specifying the part name, save it as the name you specified for **addGroupEditor** in the
**constructPropertyEditor** method.

You now have a custom property edit dialog for your part.

**Building Smalltalk Libraries for a Part**

You generally build two libraries for a part: a **runtime** library and a **workbench** library.

The runtime library contains the code for the part class and other new classes and methods required for an instance of the part class to execute in a runtime environment. The runtime library also contains a **bind** method that binds the corresponding workbench library if the part is being used in the workbench.

The workbench library contains classes that are required only in the workbench environment, such as the property dialog class and any supporting classes and code.
Runtime and workbench code must be separated because the workbench code typically has dependencies on development libraries which are not available during runtime execution. Eliminating the workbench code also speeds up loading parts and reduces the size of the runtime application.

For this reason, when you are organizing your work you should separate the runtime code from code that will support the workbench. If you are using Visual Smalltalk Enterprise, simply put the code in separate packages. If you are not using Enterprise, write separate scripts for building runtime and workbench libraries.

**Part File Naming Conventions**

Since there may be several files associated with a part, you should use similar names for the files so you can tell how each file is related to the part by looking at the filename.

For example, in some situations there is specific code needed to support the workbench, but is not required for the runtime. This is not the case for the random number generator, but if it were, there is a convention for distinguishing the runtime and workbench related libraries.

The part file is named RANDOM.PAR. By convention, the runtime library is designated by the letter “R” appended to the base name of the part and the workbench file by appending “W”, so its two Smalltalk libraries can be named RANDOMR.SLL and RANDOMW.SLL. Employing this convention will help you understand which files you need for a given application.

**Workbench Bind and Unbind Methods**

The *bind* method for a Smalltalk library is the method that is initially executed when the library is bound. If you have separate runtime and workbench libraries, you need to include an expression to load the workbench library in the bind method for the runtime library.

The bind method should handle any user-specific operations that might be necessary after installation of either library, such as initializing any part values. The bind method for a part is a class method called `installPartRuntimeLibrary`.

We don't need to do this for the random number generator, but if we did, its bind method would look something like the following:
installPartRuntimeLibrary
" Install the RandomNumberGenerator part runtime library. Load the workbench support library if appropriate."
PARTSSessionManager
installWorkbenchSupportLibrary:
  self workbenchLibraryName
for: 'RandomNumberGenerator Part'.

The unbind method in a runtime library is the method that is executed when the runtime library is about to be unbound. This method should unbind the part's corresponding workbench-time library that was initially installed in the bind method. The unbind method should also handle any user-specific operations that might be necessary before the runtime (or workbench-time) library is unbound.

The unbind method for a part is called removePartRuntimeLibrary by convention and is a class method. The unbind method for the random number generator part would look like the following (again, we don't actually need this method):

removePartRuntimeLibrary
" Remove the RandomNumberGenerator part runtime library. Remove the workbench support library if it was installed."
PARTSSessionManager
removeWorkbenchSupportLibrary:
  self workbenchLibraryName.

In both examples, the class method workbenchLibraryName returns the workbench-time library name as a string object.

Creating a Catalog Part File

The part you have created is all you need to use it in building an application. An inconvenience is that we must separately bind the part's supporting libraries to the environment before we can use the part.

This next procedure makes the part easier to use by building into it information needed to automatically bind required libraries when the part is placed on the workbench. This makes the part more suitable for adding to a catalog, and so we will call it a catalog part file.

You can build the catalog part file either using the workbench or a Smalltalk script.
Creating a Catalog Part File

Building a Catalog Part in the Workbench

1. Open a new workbench.

2. Bind your part's runtime library.

   The runtime bind method will bind the workbench library RANDOM.SLL automatically.

3. Add an instance of your part to the workbench.

   Choose Developer / Add object... and in the dialog, select RandomNumberGenerator in the class list. Then click OK.

4. Click on the workbench background to place the new part.

5. Double-click on the workbench background to bring up the application properties dialog.

   Set the properties as desired. You should:
   - specify the part name
   - enter a one-line hint
   - specify the help file name, if any
   - edit the icon to represent the part in a catalog
   - set the part file reuse mode to "Ensemble of parts"

   Click OK when you are finished.

6. Choose File / Save as... to save the part as a .PAR file.

   You have now created a part file that will automatically bind the necessary library files when the part is used in an application.

Building a Catalog Part in Smalltalk

You can also create the part file programmatically by evaluating a Smalltalk expression. Creating the part file programmatically has two major advantages:

- It is easier to re-evaluate a script to rebuild a part than to go through the workbench procedure just described.
- You can explicitly describe library dependencies, whether those libraries are currently bound or not.

The script sends the following message to the PARTSAppliication class, supplying the necessary arguments:
writeNewFile: aPathName  
   name: aNameString  
   solePart: anObject  
   icon: applicationIcon  
   partIcon: partIcon  
   hint: aHintString  
   helpFile: aHelpFile  
   helpTopic: aHelpTopicName  
   nestingMode: nestingMode  
   libraryMappings: libraryMappings

The arguments of this message are:

aPathName A string containing the name of the .PAR file being created.
aNameString The part name.
anObject The initialized instance of the part to be stored in the part file.
applicationIcon An icon bitmap containing the file name of the icon to display in the catalog.
partIcon The icon displayed for the part in the workbench. It should be nil for a visual part, and the same as the application icon for a nonvisual part.
aHintString The hint string for the part.
aHelpFile A string specifying the name of the file containing help for this part.
aHelpTopicName A string containing the help topic in the help file for this part.
nestingMode A symbol, corresponding to the reuse option of the part. This should usually be #merge, which selects the “Ensemble of parts” reuse option. Other choices are #reference, which selects the “linked” reuse option, and #embed, which selects the “embedded” reuse option.
libraryMappings An IdentityDictionary whose key is a class and the corresponding value is the name of the library file that contains the class and should be bound when the part file is loaded.
The *libraryMappings* parameter lets you to build a catalog part file and specify the library dependencies of the part file, even if the image in which you are building the part file does not currently have the correct library dependency for your part file class. This means you can create a part file even if the libraries to support the part file are not bound to the current development image.

Let’s look at an example of creating the random number generator part using this `PARTSApplication` class method.

You need to create an icon file in the standard .ICO file format for the operating system platform before creating the part file. You can use a third-party icon editor or the workbench icon editor.

Class `PARTSFile` has messages to access the contents of an existing .PAR file. In particular, the `iconFromFile:` message can be used to load the icon from a .PAR file into the image for editing or reuse in another part file.

After you have created an .ICO file for your part, you can use the `fromFile:` message of `PARTSIcon` class to load the icon from the file. The random number generator example uses RANDOM.ICO.

To build the RANDOM.PAR file, you evaluate the following expression in a workspace:

```
" Create the RandomNumberGenerator part "
| icon |
icon := PARTSIcon fromFile:
   'smlltalk\tutorial\random.ico'.
PARTSApplication
   writeNewFile: 'user\random.par'
   name: 'RandomNumberGenerator'
   solePart: (PARTSRandomNumberGeneratorPart new)
   icon: icon
   partIcon: icon
   hint: 'A random number generator.'
   helpFile: "
   helpTopic: "
   nestingMode: #merge
   libraryMappings: ( IdentityDictionary new
      at: PARTSRandomNumberGeneratorPart
      put: 'RANDOMR';
      yourself )
```

You can place the resulting part in your Parts catalog using the catalog *File... / Insert Part File...* menu item.
When to Recreate a Part File

After a part file is initially created, further development may make it necessary to recreate the part. A part file should be recreated if any of the following occurs:

- Changes are made in the class definition for the part, such as the addition of a new instance variable.
- Library dependencies have changed, either requiring additional libraries or removing libraries that are no longer required.
- Changes in any of the initial settings or properties of the part are made.

A change in a library referenced by the part does not necessarily require rebuilding the part. Usually, only a change in library dependencies will make rebuilding necessary.
Overview

This chapter discusses a variety of advanced topics related to developing parts. The techniques discussed here may add to the value of a part, but are not required.

Dynamic Messages

It is occasionally useful for a part to change its interface dynamically based on settings in its property editor. For example, when you change the properties of the DLL accessor to define new procedures based on DLL entry points, new messages are automatically added to access those functions. The message selectors the part displays in its interface list depend on the part properties that are set by the part developer when the part is actually edited in the workbench.

To support dynamic, instance specific messages, your `partMessages` implementation must construct its message interface list dynamically, based on the instance data of the part. As a result, your part’s message interface might not have methods corresponding to all the message names that the part advertises. The alias link mechanism of the workbench handles routing such messages to a dispatching method in the part when a link is fired.

Dynamic message entries must be Smalltalk symbols. You must implement the methods `partLinkClassFor:` and `partRealSelectorFor:` in your part class, as well as one or more methods that handle the dynamic messages at execution time. Your `partRealSelectorFor:` method is used by the alias link mechanism to obtain the name of the dispatching method that handles a dynamic message selection send.

`partLinkClassFor: aSelector`

Sent to a part during link creation. The argument is a symbol from the `partMessages` items. Must answer the class `PARTSAliasLink` for a dynamic message selector, otherwise the class `PARTSLink` (best done using `super`).

Visual Smalltalk Enterprise User's Guide

131
partRealSelectorFor: aSelector

Sent to a part during link creation (edit time) or when a
dynamic message selector is sent to the part from a script
(application execution time). Must answer the message
selector of a two-argument method supported by the
receiver that can dispatch the dynamic message. The first
argument to your dispatching method is the dynamic
message selector, the second is the arguments array of the
message.

The PARTSArrayHolderPart part is an example of a part that
dynamically generates instance specific messages. The message
and event protocol that a PARTSArrayHolderPart displays in its
interface dialog is determined by the size of the array that it holds.

The message interface of a PARTSArrayHolderPart consists of
standard messages supported by all PARTSValueHolderPart
parts (such as value) and all arrays (such as at: and at:put:). These
message selectors all have corresponding methods in the classes.
In addition, a PARTSArrayHolderPart part dynamically generates
messages selectors for its interface list to get and set each element
in the array. These are dynamic messages for which there is no
method in the image.

For example, you can link to a PARTSArrayHolderPart
containing a three-element array with messages such as element2
and setElement2: However, there are no such methods in the
PARTSArrayHolderPart class.

The PARTSArrayHolderPart class has a method called
keyMessages, which dynamically constructs the set of element
accessing messages displayed in the part’s message list. These
dynamic message selectors are registered with the workbench
alias link mechanism by the partLinkClassFor: and
partRealSelectorFor: methods.

The partLinkClassFor: message is sent to the part at link-creation
time by the workbench link editor, and must answer class
PARTSAliasLink for any message selector generated by the
keyMessages method. (The normal PARTSLink class is answered
for the message selectors that correspond to real methods.)

The partRealSelectorFor: message is then sent to the part to
obtain the mapping from the dynamic message selector of the
alias link to the real message selector to send to the part when the
link is fired.

The PARTSArrayHolderPart has two methods for dispatching
dynamic message selectors. The element accessing messages such as
element2 are routed to the method lookup:arguments: for
Validating and Setting Part Values

As you have seen, a part can have a value that can be set and retrieved through its part messages. The EntryField and ValueHolder parts are examples of parts whose values are set and retrieved using the `setValue:` and `value` messages. When building a part that requires an external message interface for setting and retrieving its value, you should implement `setValue:` and `value` messages for the part, to maintain consistency with the parts provided with the PARTS Workbench environment. Users of the part will quickly recognize the familiar message names and their purpose.

A part that can set and retrieve its value should also trigger a `changed:` event whenever its value is changed. The `changed:` event can then be linked to other parts in the application when the application needs to know that the part value has been changed.

A part should validate arguments passed with the `setValue:` message to ensure that the argument is suitable for the kind of value the part is designed to use. If possible, a part should do its best to handle arguments which may not be in the same form as the internal representation used for its value. Thus, the validation process applied to arguments passed using the `setValue:` message may also include converting the argument into a form suitable for setting the part value. This makes for a much more usable part, particularly when it is not immediately obvious to a user of the part what type of argument is expected.

Suppose a part was designed to maintain an integer value. A user might connect the argument link of the `setValue:` message for the part to the `value` message of a String Holder part. So, for example, instead of providing the integer value 3 as the argument of the `setValue:` message, the message is sent with the string ‘3’. The `setValue:` method code should be able to validate the argument, and convert the string ‘3’ into the integer value 3, before using the message argument to change the part value.
Therefore, the `setValue:` message of a part has potentially four functions it must perform:

- Validate the argument sent with the message to see if it is in the form used by the part to store its value and convert the argument, if necessary, to the desired form.
- Determine if the new value is different from the current value.
- Set the part value to the new value.
- If the value has changed, trigger the `#changed:` event.

A part should also implement a message named `value:`, which can be sent from the `setValue:` message. Typical, the `value:` message should not be listed in the part’s message interface. Rather, users of the part should use the `setValue:` message to set the new value for the part. The `setValue:` method should contain all the necessary logic to validate and/or convert the message argument, trigger the `changed:` event, if applicable, then send the `value:` message to the part. The `value:` method implements the code necessary to actually change the part value internally.

The advantage of using the external `setValue:` and internal `value:` messages in combination is to separate the validation logic and user interface code from the internal implementation of the part used to access and change its value.

The generic `setValue:` method shown here demonstrates the four functions listed earlier. It is implemented as an example for a part which expects an integer value, but can also accept the string representation of an integer as well.

```smalltalk
setValue: aValue

"Set the contents of the receiver to the integer represented by aValue. Validate aValue, and convert if necessary. Finally, if the final value is different from the beginning value, trigger the changed: event.

Answer the final value."

|newValue |
newValue := ( aValue isInteger
    ifTrue: [ aValue ]
    ifFalse: [ aValue isString
        ifTrue: [ aValue asInteger ]
        ifFalse: [ nil ] ].

self value: newValue.
newValue ~= self previousValue ifTrue: [
    self triggerChanged;
    previousValue: value ].

^newValue
```
Parts with multiple values that can be set and retrieved should implement a three-method interface similar to `setValue`, `value` and `value:` for each value. The method names should be derived from the value name. For example, a part that stores and retrieves the path to a file might implement three methods called `setPathName`, `pathName` and `pathName:`.

**Form Generation**

You can add a property to a part that causes it to automatically add parts to the application and create links to the new parts.

Some nonvisual parts, such as the C struct and Btrieve accessor represent records or data structures that contain several data items. The visual equivalent for an aggregate of data items is a **form**. A form is a collection of visual parts that represent the values in the aggregate.

When a form is automatically created in the workbench with the `Create form parts...` property button, the visual parts corresponding to each data item are automatically placed in the Workbench, labelled with static text parts. Links are automatically created between the visual parts representing the data items and the messages and events in the data aggregate part that access the values of those data items.

The illustration below shows an example form created by a Btrieve accessor part.
These visual parts were created from the field definitions in the Btrieve accessor property dialog:

For more information about which visual parts correspond to various data types, see the Btrieve Accessor section in the Visual Smalltalk Enterprise Part Reference.

The PARTS Workbench classes PARTSFormGenerator and PARTSFormGeneratorItem contain the framework for creating visual parts and links under the control of another part.

The following discussion describes how to incorporate form generation behavior into a part.

**Adding a Create form parts... Button**

You’ll need to create a new part for the random number property edit dialog that has one additional button named CreatePart-Button and labelled Create form parts... as shown below.
You need to reimplement the `fileName` method in your part class to open the new dialog part file:

```smalltalk
(fileName " Private - Answer the name of the part file containing the property dialog. 
If no extension is given, open editable .par file; 
if not found, open non-editable .pax part."
'^PPAPP2.PAR'
```

**Invoking Form Generation**

In general, when the `Create form parts...` button is pressed, the message

```smalltalk
generatePartsFor: aCreatorPart 
fromDialog: aDialogWindow
```

is sent to an instance of `PARTSFormGenerator`.

The dialog class needs to retain the form generator after it creates the parts and links in case the property edit dialog is cancelled. In case of cancellation, the message `cancelChanges` to the form generator will cause the generated parts and links to be removed from the application.

In order to use the form generator, you need add the following items to the property dialog class:
1. A “forms” instance variable to hold a collection of form generators. The *Create form parts...* button might be pressed several times while the property dialog is open, creating a new form generator with each press.

2. Implement `initializeValues` to initialize the instance variables:

   ```smalltalk
   initializeValues
   " Private - initialize instance variables. "
   super initializeValues.
   forms := OrderedCollection new.
   ```

3. Add an accessor method to get the *Create form parts...* button:

   ```smalltalk
   createPartsButton
   " Private - Answer the Create parts button in the dialog window. "
   ^self partNamed: 'CreatePartsButton'
   ```

4. Reimplement `initializeParts` to register the `createParts` method to be called when the *Create form parts...* button is pressed:

   ```smalltalk
   initializeParts
   " Private - the dialog window application has been loaded. Register the event handlers and initialize the items in the property dialog based on the current values of the part. "
   super initializeParts.
   self createPartsButton
   when: #clicked
   send: #createParts to: self
   ```

5. Implement `createParts` to create an instance of `PARTSFormGenerator` and call `generatePartsFor:fromDialog:`.

   ```smalltalk
   createParts
   " Private - the Create Parts button is pressed."
   | aFormGenerator |
   aFormGenerator := PARTSFormGenerator new.
   aFormGenerator generatePartsFor: self part
   fromDialog: self dialogWindow
   aFormGenerator createdParts
   ifTrue: [ forms add: aFormGenerator ].
   ```

   The argument `self part` accesses the part whose properties are being edited, that is, the part creating the form. The
Form Generation

argument `self(dialogWindow)` is an accessor for the dialog window part of the property dialog itself. The dialog window argument is required in order to hide the property dialog so the workbench can be brought to the top and the parts can be placed. The `FormGenerator` message `createdParts` returns true if the user created any parts. If so, the form generator is added to the forms collection so the part creation can be undone.

6. Implement `cancelChanges` to undo the form generation if the user changes his mind:

```smalltalk
cancelChanges
    " Undo the link and part creation if cancelled. "
    super cancelChanges.
    forms notEmpty
        ifTrue: [
            self partEditor selectOffAll.
            forms do: [ :aFormGenerator |
                aFormGenerator cancelChanges ].
            self partEditor select: self part ].
```

The forms collection will be empty unless the user created one or more sets of form parts while the property dialog was open. If form parts were created, each form generator is asked to cancel its changes. The messages `selectOffAll` and `select:` are used to remove the selection from all parts (in case parts being removed are selected) and then reselect the part.

Customizing Form Generation

The part itself is responsible for defining what happens when a form is created for it. The form generator expects a part to implement two methods:

- `partFormGeneratorNames`  
  Answer an OrderedCollection of the part names created by the Create Parts dialog.

- `partFormGeneratorItemsFor: partNames`  
  Answer an OrderedCollection of PARTSFormGeneratorItem representing the specified partNames.

The `partFormGeneratorNames` method returns the names of the data items that parts that can be generated for. These item names are usually taken from the property fields, such as those of a C struct or Btrieve accessor.
For the random number generator example, the form item names are not determined by properties so they can be hard-coded:

```
partFormGeneratorNames
    "Answer an OrderedCollection of the part names created by the Create Parts dialog."

^OrderedCollection
    with: 'LowerBound'
    with: 'UpperBound'
    with: 'Seed'
```

These item names are presented to the user in the Form Item Layout dialog brought up by the form generator.

This dialog lets the user choose which of the possible data items to create visual parts for, and the order they will be placed in the user's application.

The names you select in this dialog are passed to `partFormGeneratorItemsFor:`, which maps each item name into an instance of `PARTSFormGeneratorItem`.

For the random number generator example, the mapping of an item name to an instance of `PARTSFormGeneratorItem` is done with a dictionary constructed by the `partDict` method.

```
partFormGeneratorItemsFor: partNames
    "Answer an OrderedCollection of
    PARTSFormGeneratorItem representing the specified partNames."

    | result |
    result := OrderedCollection new: partNames size.
    partNames do: [ : each |
        result add: (self partDictionary at: each)].

^result
```
Specifying Characteristics of Generated Parts

The **PARTSFormGeneratorItem** instance completely describes the visual part to be created, including which part file to use, the part name to give it in the application, and links to be created to and from the part.

An instance of **PARTSFormGeneratorItem** is created with the class message:

```
fromPartFile: aPartFileName
name: aName
extent: aPoint
staticTextLabel: labelString
partEvent: aPartEvent
partMessage: aPartMessage
sourceEvent: aSourceEvent
sourceMessage: aSourceMessage
```

The meaning of each of the arguments is described in the following table.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>aPartFileName</strong></td>
<td>A string containing the name of the .PAR file for the visual part to create for this item.</td>
</tr>
<tr>
<td><strong>aName</strong></td>
<td>The part name to be given to the part in the Workbench.</td>
</tr>
<tr>
<td><strong>aPoint</strong></td>
<td>A point representing the extent of the visual part.</td>
</tr>
<tr>
<td><strong>labelString</strong></td>
<td>A string containing the text to be placed in a static text part that labels the new visual part. If nil, no static text part is created for the item. The static text label is placed to the left of the visual part.</td>
</tr>
<tr>
<td><strong>aPartEvent</strong></td>
<td>Either nil or a symbol. If nil, no link will be created from the new part to the creating part. If a symbol, it names an event of the new part to be linked to <strong>aSourceMessage</strong> in the creating part.</td>
</tr>
<tr>
<td><strong>aPartMessage</strong></td>
<td>Either nil or a symbol. If nil, no link will be created from the new part to the creating part. If a symbol, it names the message of the new part to be linked to <strong>aSourceEvent</strong>.</td>
</tr>
</tbody>
</table>
We now know enough to implement the final, and probably most important method, `partDictionary`, that creates the `PARTSFormGeneratorItem` instances for the two data items:

```smalltalk
partDictionary
    "Answer a dictionary that maps part names to
     PARTSFormGeneratorItems."
    | visualPartFileName visualPartExtent visualPartChangedEvent |
    visualPartFileName := 'FRMTTDF.T.PAR'.
    visualPartExtent := 60@25.
    visualPartChangedEvent := #changed:

    ^Dictionary new
    at: 'LowerBound'
        put: (PARTSFormGeneratorItem
             fromPartFile: visualPartFileName
             name: 'LowerBoundEntryField'
             extent: visualPartExtent
             staticTextLabel: 'Lower Bound'
             partEvent: visualPartChangedEvent
             partMessage: nil
             sourceEvent: nil
             sourceMessage: #lowerBound: );

    at: 'UpperBound'
        put: (PARTSFormGeneratorItem
             fromPartFile: visualPartFileName
             name: 'UpperBoundEntryField'
             extent: visualPartExtent
             staticTextLabel: 'Upper Bound'
             partEvent: visualPartChangedEvent
             partMessage: nil
             sourceEvent: nil
             sourceMessage: #upperBound: );
```

We now know enough to implement the final, and probably most important method, `partDictionary`, that creates the `PARTSFormGeneratorItem` instances for the two data items:

`aSourceEvent` Either nil or a symbol. If nil, no link will be created from the creating part to the new part. If a symbol, it names an event of the creating part to be linked to `aPartMessage`.

`aSourceMessage` Either nil or a symbol. If nil, no link will be created from the new part to the creating part. If a symbol, it names the message of the creating part to be linked to `aPartEvent`. 
at: 'Seed'
put: (PARTSFormGeneratorItem
   fromPartFile: visualPartFileName
   name: 'SeedEntryField'
   extent: visualPartExtent
   staticTextLabel: 'Seed'
   partEvent: visualPartChangedEvent
   partMessage: nil
   sourceEvent: nil
   sourceMessage: #seed: );

yourself

When parts are created for these items, links are automatically created between the entry field’s **changed:** event and the random number generator’s **lowerBound:**, **upperBound:**, and **seed:** messages.

### Implementing partEditProperties

To get the form generator to use the property dialog with the Create form parts... button on it, your part needs to reimplement **partEditProperties** as follows:

```plaintext
partEditProperties
   " Open an editor on the Parts properties of the receiver. "
   (Smalltalk at: #PARTSClassName) new
   openOn: self.
```
Glossary

abstract class
A class that is not intended to have instances, but whose 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. See class, instance, encapsulation, subclass, superclass. Compare concrete class.

accelerator key
A sequence of keystrokes that allows you to access menus and menu items without using a mouse. Function keys, and keys combined with Ctrl or Alt can be used as accelerator keys.

action
An object that responds when an application event is triggered. When an application triggers an event, each action is evaluated in order. See event, selector, receiver.

active exception handler
An exception handler defined to take some action when a specific class of exception occurs and its protected block is currently being evaluated. See exception handler, protected block.

active handler's exception environment
The exception environment at the time that an exception handler is executing. That is, the exception environment as it was at the time that the on:do: message was sent. See active exception handler, exception, exception environment.

answer
The object returned as the result of sending a message.

application(s)
A unit of software that one can use to perform a distinct task, which the user perceives as one functional unit—for example, word processing, accounting, operating manufacturing machinery, or simulating atmospheric dynamics. In Visual Smalltalk, an application is a distinct set of classes collaborating to provide this single functional unit.
| **application framework** | A standard set of classes that represent how a user interface can interact with the computational mechanisms underlying an application. The application framework defines abstract superclasses, concrete classes, and sets of messages to support the construction of applications. These classes and messages generalize and abstract patterns by which a user interface collaborates with the application logic. See abstract class, concrete class. |
| **Application Program Interface (API)** | A set of standard, callable functions available to Windows applications, typically to support lower-level tasks and services. |
| **arc** | A portion of the outline of an ellipse. |
| **argument** | An object made accessible to another object (the receiver) for the duration of the execution of a method. |
| **array** | A collection of objects that may be indexed by an integer from one through the size of the array. |
| **attribute** | A property or characteristic that distinguishes one class of objects from another. |
| **background color** | The color in which the less conspicuous parts of a graphics medium appear, such as the background behind text. Compare foreground color. |
| **baseline** | The imaginary line upon which the characters in a line of text appear to stand. Descenders go below the baseline. |
| **behavior** | Normally refers to the external behavior of an object—what it can do and what operations or services it provides, as seen by its clients. That is, the set of messages to which an object can respond. Occasionally refers to the internal behavior of an object, the data and methods it uses to carry out its external behavior (not to be confused with the class Behavior). See message, object. |
| **binary expression** | A binary expression sends a series of binary messages that are evaluated from left to right. The traditional arithmetic operators are implemented in Smalltalk using binary expressions. |
| **binary message** | A message with a single argument whose selector is not a keyword. For example, arithmetic messages, such as \(3 + 4\), are binary messages. |
| **BitBlt** | Bit block transfer. A programming technique used to transfer rectangular blocks of bits. |
| **bitmap** | A two-dimensional array of bits. Bitmaps may be thought of as three dimensional objects containing X/Y coordinates and color bits. |
block
A set of deferred expressions, somewhat like a method, delimited by square brackets. Blocks are evaluated when they receive the message `value`, or some variant. See `method`.

boldface
A darker, heavier set of characters than the characters of the regular font family. See `family, typeface`.

browser
A Visual Smalltalk tool that allows you to visually inspect and edit the contents of Visual Smalltalk classes and methods. See debugger, Class Browser, Class Hierarchy Browser, Method Browser, Selector Browser.

byte
An immediate data type consisting of eight bits. See `immediate data type`.

callback
A Smalltalk method that can be called from, and executed in response to, a procedure outside of Smalltalk, such as an API call.

cascaded message
Cascaded messages are a series of messages to the same receiver. Each message after the first is preceded by a semicolon. A period indicates the end of the cascaded message group.

change log
The file (CHANGE.LOG) that maintains a continuous record of source code changes, image save time stamp messages, and expression evaluations.

character
A member of the alphabet, a number, a punctuation mark, or any other mark used in a writing system.

character constant
A character constant is an instance of class `Character` that is syntactically represented as a dollar sign followed by any printable character.

class
A class should cleanly represent a single idea or concept and characterize a set of objects that have meaningful operations and useful properties. A class is a programming construct analogous to an abstract data type with encapsulated operations, thereby combining data structure and behavior. It defines the behavior of similar objects (instances) by specifying their implementation—the variables they contain and the definition of the methods available for responding to messages sent to them. The class specifies external behavior as well as internal knowledge and methods. Each class is a template for creating objects that model real-world concepts or designed inventions/mechanisms. See `object`. Compare `instance`.

Class Browser
The Class Browser, like the Class Hierarchy Browser, gives you the ability to edit methods but focuses on an individual class within the hierarchy.
<table>
<thead>
<tr>
<th><strong>Glossary</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>class hierarchy</strong></td>
</tr>
<tr>
<td><strong>Class Hierarchy Browser</strong></td>
</tr>
<tr>
<td><strong>class instance variable</strong></td>
</tr>
<tr>
<td><strong>class library</strong></td>
</tr>
<tr>
<td><strong>class method</strong></td>
</tr>
<tr>
<td><strong>class variable</strong></td>
</tr>
<tr>
<td><strong>client</strong></td>
</tr>
<tr>
<td><strong>client area</strong></td>
</tr>
<tr>
<td><strong>clipping region</strong></td>
</tr>
<tr>
<td><strong>combo box</strong></td>
</tr>
<tr>
<td><strong>comment</strong></td>
</tr>
<tr>
<td><strong>common dialogs</strong></td>
</tr>
</tbody>
</table>
### Glossary

**concrete class**  
A class intended to have instances. Concrete classes may inherit functionality from either an abstract or concrete superclass that is common to itself and other concrete classes. A concrete class may also inherit a common interface, but it implements the code necessary to create instances of the unique entity it defines by providing concrete implementations of both inherited and unique behavior. See class, instance, encapsulation, subclass, superclass. Compare abstract class.

**control menu**  
See system menu.

**control structure**  
A programming language construct that controls the flow of processing such as looping and branching. Control structures are implemented in Smalltalk using messages with blocks as arguments or receivers.

**cursor**  
A visual on-screen graphic symbol indicating where the user is currently pointing by using the mouse, keyboard, or other pointing device. Compare insertion point.

**data abstraction**  
Breaking an application into modules based upon the data that it manipulates rather than the functions that it executes. The common characteristics (both instance variables and methods) of a group of similar objects are collected to create a new class.

**data structure**  
An organizational scheme, such as an array, that is applied to data so that it can be interpreted and actions defined for the data can be performed.

**debugger**  
A Visual Smalltalk tool used to debug Smalltalk code. There are two debugger windows: a Walkback window and a Debugger window. A Walkback window pops up automatically when errors are detected. When you need more information than provided in the Walkback, you explicitly request a Debugger window by selecting Debug in the Walkback menu of the Walkback window or by clicking the Debug button located in the Walkback window. See browser, Walkback.

**depth**  
In color systems, the number of bits per pixel in a graphics medium. This determines the number of pixel values, and therefore colors that can be represented. For example, a medium of depth \( n \) can represent \( 2^n \) possible pixel values.

**Device-Independent Bitmap (DIB)**  
A bitmap whose characteristics are modified at runtime for the target graphics output device. The bitmap is “graphics-independent” because it contains a color table that maps pixels to their RGB color values. The DIB format is the preferred format for distributing bitmaps.
**dialog box**
A collection of controls designed to collect additional user information and choices, usually in order to perform a function that requires the information—such as print or save a file, or select or configure a printer.

**dictionary**
An object that contains associations between keys and values.

**Disk Browser**
The Disk Browser provides file management capabilities that are similar to those that the Windows File Manager provides. That is, you can copy, delete, rename, and move files. In addition, you can create and edit files, change the attributes of existing files, and file-in or evaluate Smalltalk code.

**display**
The graphics medium on which windows are displayed. See *screen*.

**displaying**
The operation by which stored pictures or bitmaps are made visible on the display or in a window. See *bitmap*. Compare erasing, filling, stroking.

**do it**
Evaluate the current selection in the active text pane as a Smalltalk expression.

**Double-Byte Character Support (DBCS)**
Most programming languages define the same assignments for the standard 7-bit ASCII characters, but differ in their assignments of characters in the range of [128...255]. For example, in Windows, the ANSI character set uses a set of 256 characters to support most of the Latin-based languages. This character set is referred to as the *single-byte character set*, since every character can be encoded in a single byte. However, certain languages such as Chinese, Japanese, and Korean require more than 256 characters. These languages use a double-byte character set that assigns characters to values in the range [256...65535]. A character with a value of 256 or greater is referred to as a *double-byte character*, since it requires 2 bytes to encode its value.

**drag/drop**
A direct-manipulation technique. After selecting items, the user grabs the items by pressing the left mouse button. Then, the user drags these items to a receiver by holding the mouse button down while moving the mouse cursor to the receiver. When the mouse cursor passes over the receiver, the user drops the items by releasing the mouse button.

**drop-down menu**
A menu that is displayed beneath the name of the menu in a menu bar when the menu name is selected. Visually, the menu appears to drop down like a window shade.

**dynamic binding**
The binding of a method to an object at runtime.
| **Dynamic Data Exchange (DDE)** | A Windows client/server protocol for interprocess communication. The client initiates the conversation with a server that performs the requests. DDE facilitates one-time data transfers and ongoing exchanges in which applications send updates to each other as information becomes available. Although DDE can be initiated by the user, it typically continues without further user interaction. |
| **Dynamic Link Library (DLL)** | A disk file containing a collection of shared, unchanging objects in the standard system. |
| **encapsulation** | The bundling of data (instance variables) with the methods that manipulate it into a single object. |
| **Encyclopedia of Classes** | A printed volume that describes the public classes and methods for a given version of Visual Smalltalk, and provides a method cross-reference. |
| **event** | The mechanism that an object uses to notify an application that something has occurred, to which other objects may need to respond. Issuing the notification is called triggering an event. The most common events result from user actions such as moving the mouse or pressing a key on the keyboard. |
| In Windows, an event is an action or occurrence that causes Windows to generate a WM message. Most commonly, Windows generates WM messages in response to user actions that affect the window object. A method defined for a WM message causes a message to be sent to the window object. |
| **exception** | An unusual or undesired occurrence that can occur during the execution of an application. When an exception occurs, we often want the application to take some special action. See nonresumable exception, resumable exception. |
| **exception environment** | An ordered list of active handlers. Each Smalltalk process has a distinct exception environment. When an exception is signaled, the exception-handling system sends a message to the first entry in the list to determine if it handles an exception of the specific class generated. The first exception handler encountered that can handle the signaled exception does so. See exception, signal. |
| **exception handler** | An object defined to take some action when an exception occurs. It has three parts: the class of exception it is to watch for, the block of code it is to execute when such an exception occurs, and the block of code it is guarding. See exception. |
| **expression** | A message expression is a request to an object (the receiver of the message) to perform a computation and return an object as the answer. There are three kinds of message expressions: unary, binary, keyword. |
file association  
A Microsoft Windows feature that allows you to associate a file with an application. File association allows you to open the file and start the application when you select the file.

file in  
A file-in is Smalltalk source code that you add to the image. You file in Smalltalk code by choosing the File It In menu item after selecting the text in the current window or pane. Smalltalk code can also be added directly to the image by selecting the File/Install... menu item. Either method compiles and saves the Smalltalk code in the image. Contrast file out.

file out  
A Visual Smalltalk procedure that writes the class definition along with all of the instance and class methods of the selected class to a file. The class definition is written out to disk in file-in format and does not affect the class definition in the system. Contrast file in.

filling  
The operation by which figures are colored, using the specified or foreground color, or filled with a pattern. See foreground color. Compare displaying, erasing, stroking.

fixed-width font  
A font whose characters are all the same width, such as a typewriter font. This text is printed in a fixed-width font, so an upper-case “M” is the same width as a lower-case “l.” See character, font. Compare proportional font.

focus pane  
The pane that receives keystrokes or mouse clicks. At any given time, Visual Smalltalk has exactly one focus pane, which is usually the text pane, or the last text pane within whose region the user clicked the mouse. However, a list pane or a graphics pane can also be a focus pane, or you may define another kind of pane to receive focus in your application. If the user clicks the mouse within a pane, that pane will receive the focus if it can.

font  
A set of characters selected from a particular design, called the typeface. Characters in a font have size (such as 10 point), weight (such as bold), and style (such as italic). Font size is normally specified in points. One point is 1/72 inch. Note that Visual Smalltalk supports all fonts provided by the host operating system. These fonts can be divided into two major categories: bitmap fonts and outline fonts. Bitmap fonts can be displayed faster than outline fonts but they cannot be scaled with consistent quality. See character, point size, typeface.

font family  
A typeface, such as Helvetica or Times, composed of many fonts. These fonts share the same basic letterforms, while appearing in a variety of weights, slants, and sizes. See point size, typeface. Compare font.
foreground color  The color in which text or geometric objects are drawn, or other items intended to attract the user’s attention. Compare background color.

framework  A reusable design; a set of classes, usually including abstract classes, that provides the skeleton of an application. Classes in a framework collaborate in ways that are not specific to any particular application, but rather define a structure that a variety of applications could have. See abstract class.

garbage collection  The automatic deallocation of unreferenced objects from memory. See garbage collector.

garbage collector  An internal memory management mechanism that removes any object not referenced by another object in the system. See garbage collection.

global  A class, a pool, or a global variable.

global variable  A variable that is available to all classes within the development environment.

guaranteed execution  Guaranteed execution ensures that specified blocks of code, called protection blocks, are always evaluated if other blocks, called protected blocks, are evaluated. See protected block, protection block.

handle  A 32-bit identifier (token) that represents an object such as a bitmap, memory block, or window.

Hop  In the Debugger window, Hop executes the next expression in the process being debugged.

hue  The sensation derived from the wavelength of light reflected or transmitted by a color; that quality of a color that distinguishes red from blue.

icon  A bitmap that is used as a visual mnemonic for an application or message. Double-clicking on an application icon invokes the application.

image  The Visual Smalltalk development environment and all the objects it contains.

immediate data type  Data types that can be passed as parameters to routines external to Smalltalk. See byte, integer, pointer.

indexed instance variable  Instance variables either have a name or are referred to with an integer index. Indexed instance variables are accessed and changed only through messages (usually using at: and at:put: messages with integer indices). See instance variable. Contrast named instance variable.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>information hiding</td>
<td>The act of hiding some or all of the things that have been encapsulated. Both the data and the methods are hidden. Message passing is the only way to see or affect an object's data. Information hiding protects the internal workings of the object so that, if they must be modified, those changes will not affect any other parts of the program. See object.</td>
</tr>
<tr>
<td>inheritance</td>
<td>The mechanism for both organizing and maintaining a collection of classes that supports reusability by defining new classes in terms of existing ones. The behavior of a class is inherited by its subclasses. A subclass can take advantage of the behaviors of those classes that are above it in the hierarchy, so that the subclass designer needs only to address those behaviors and attributes that differ from those that can be inherited. See class, behavior.</td>
</tr>
<tr>
<td>input event</td>
<td>A message, usually prefaced with WM, sent by the Notifier to a user interface object—generally a window or pane. Compare event, operating system event.</td>
</tr>
<tr>
<td>insertion point</td>
<td>The point where text insertion and deletion occurs in a text pane or dialog box. The mouse cursor at the insertion point is normally represented by an I-beam.</td>
</tr>
<tr>
<td>inspect it</td>
<td>Evaluate the current selection in the active text pane as a Smalltalk expression, and open an Inspector on the result.</td>
</tr>
<tr>
<td>Inspector</td>
<td>Inspectors are debugging tools used to examine and change objects in the system. See debugger.</td>
</tr>
<tr>
<td>install</td>
<td>An operation performed on a disk file to merge the contents of the file with the image. The selected file may be a text file in Smalltalk file-in format, an Object Filer file, or a Smalltalk library.</td>
</tr>
<tr>
<td>instance</td>
<td>Although every object is an instance of a particular class, the term instance is usually used to refer to an object rather than a class. See object, class.</td>
</tr>
<tr>
<td>instance method</td>
<td>Method invoked by sending a message to an instance of a class. Instance methods are so called because they are associated with instances of a class, not with the class itself. The receiver of an instance message is always an object that is an instance of the class or one of its subclasses. See instance, method. Compare class method.</td>
</tr>
</tbody>
</table>

154 Visual Smalltalk Enterprise User's Guide
**instance variable**  
A variable defined to be part of each instance of a class. Instances of the class share the name of the variable, but not its value. Instances of any subclasses also inherit the name of the variable, but not its value. The value of an instance variable is typically initialized when an instance is created, and is different for each instance. Instance methods of the class and its subclasses can refer to an instance variable. Class methods cannot. There are two kinds of instance variables: named and indexed. See named instance variable, indexed instance variable. Compare class variable, class instance variable.

**instantiate**  
To create an instance of a particular class and set its initial data.

**integer**  
An immediate data type consisting of 32 bits. As a Smalltalk object, an instance of a concrete subclass of `Integer`. See immediate data type.

**italic**  
Characters that slant to the right. The characters may also look more like script than the characters of the regular font family. See font family, typeface.

**Jump**  
In the Debugger window, `Jump` executes up to the next breakpoint.

**keyword expression**  
A keyword expression sends a single keyword message with one or more arguments.

**list pane**  
A window pane that allows the user to select items from a list displayed in the control. Automatic scrolling occurs when the user moves the mouse cursor to the top or bottom boundary of the list pane frame.

**literal**  
A value that is expressed as itself rather than as the value of a variable or the result of evaluating an expression. A literal defines an object of class `Number, String, Character, Symbol, or Array`.

**main window**  
The window that is an application’s principal mechanism for displaying output to, and receiving input from, the user. The main window is the ultimate parent of all the panes defined by the application. See pane, parent.

**maximize button**  
An up-arrow button on the right side of the title bar. When pressed, the maximize button increases the size of the active window to fill the screen. Contrast minimize button.

**menu bar**  
The menu bar extends across the top of a window, below the title bar but above the client area. The menu bar displays the titles of menus that can be accessed by using accelerator keys or by clicking on the menu title.
**message**  
The mode of interaction between objects. An object (the sender) sends a message to another object (the receiver) to make a request or notify it of something, possibly passing additional objects as parameters. A message has three parts: the **receiver**, the **selector**, and optional **arguments**. The receiver can ignore the message, reply, or take some action that could include modifying its internal state, sending messages to other objects, etc. What the receiver actually does depends upon the method associated with the message (which is determined by the object’s class). Instances of the class **Message** are used to represent a delayed message-send. See **object**.

**Message Browser**  
See **Selector Browser**.

**message selector**  
The name of an operation. See **message**.

**method**  
A unit of code executed in response to receiving a message.

**method argument**  
Method arguments are temporary variables that are assigned the associated message arguments for the message that caused method invocation. In other programming languages, method arguments are called **parameters**.

**Method Browser**  
The Method Browser lets you browse and edit a list of methods. **Local Senders** and **Local Implementors** items in the **Methods** menu of the Class Hierarchy Browser open Method Browsers with the method list selection limited to the scope of the selected class and its subclasses. See **browser**.

**method index**  
The method index, in the **Encyclopedia of Classes**, is a complete index of all of the methods implemented in Visual Smalltalk.

**method temporary**  
Method temporaries are temporary variables that are visible only within a given method, and are initialized to **nil** upon method invocation.

**minimize button**  
A down-arrow button on the right side of the title bar. When pressed, the minimize button reduces the window to its minimum size, usually an icon, and closes all associated windows. Contrast **maximize button**.

**modal dialog**  
Modal dialog boxes require the user to respond to the dialog box before continuing. **Application modal** dialog boxes require the user to respond before continuing with the current application. However, the user can switch to other applications before responding. In comparison, **system modal** dialog boxes require the user to respond to the dialog box before continuing with the current application or switching to another application. See **dialog box**. Contrast **modeless dialog**.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>modeless dialog</strong></td>
<td>Modeless dialog boxes do not require the user to respond to the dialog box before continuing the current application or switching to another application. See <em>dialog box</em>. Contrast <em>modal dialog</em>.</td>
</tr>
<tr>
<td><strong>named instance variable</strong></td>
<td>Instance variables either have a name or are referred to with an integer index. Named instance variables are accessed by using their name. See <em>instance variable</em>. Contrast <em>indexed instance variable</em>.</td>
</tr>
<tr>
<td><strong>National Language Support (NLS)</strong></td>
<td>NLS mechanisms allow you to develop applications that can be used internationally without any change to the application source code. The application appears as if it is specifically written in the language native to the country in which it is being used, with all presentation text and country-specific formatting of date and time displayed correctly in the native language. See <em>Double-Byte Character Support (DBCS)</em>.</td>
</tr>
<tr>
<td><strong>nonresumable exception</strong></td>
<td>An exception that must return from the message that created the exception handler, because it cannot continue execution. See <em>exception, signal</em>. Compare <em>resumable exception</em>.</td>
</tr>
<tr>
<td><strong>Number</strong></td>
<td>An abstract class used to compare, count, and measure instances of its numeric subclasses—notably <em>Float</em> (numbers expressed in IEEE double-precision floating-point format), <em>Fraction</em> (numbers expressed as one number divided by another), and <em>Integer</em> (whole numbers with no decimal or fraction parts).</td>
</tr>
<tr>
<td><strong>object</strong></td>
<td>The basic building block of a Smalltalk program. It is usually an instance of a real-world concept, modeled by a class, that performs a useful set of related services. Normally an object has one or more instance variables that collectively define the data of the object; behavior defined by a set of methods that can modify those attributes; and an identity that distinguishes it from all other objects. Its data and the methods it uses to implement its behavior are encapsulated within the object, and are kept hidden from other objects. An object can only be affected by sending messages to it. Some objects may be transient, existing temporarily during the execution of a program. Others may be persistent, existing on some form of permanent storage.</td>
</tr>
<tr>
<td><strong>Object Filer</strong></td>
<td>The Object Filer provides a mechanism for persistent object storage and object interchange between images. The Object Filer can be used to dump and load objects in a runtime application, whereas building object libraries requires the development environment. Storing an object with the Object Filer allows objects to be transferred between different versions of Visual Smalltalk by installing cross-loaders that can read the filed object descriptions of a different version.</td>
</tr>
<tr>
<td><strong>Object-Oriented Programming (OOP)</strong></td>
<td>A programming paradigm where modularity is based on kinds (classes) of objects and how they are related to each other. Its central concept is to manage complexity by distributing it to individual objects. The primary concern is identifying what objects must be modeled, and what they must be responsible for doing and knowing. In the OOP paradigm, systems are analyzed, designed, and implemented as collections of objects that interact with each other.</td>
</tr>
<tr>
<td><strong>operating system event</strong></td>
<td>Events such as key presses or mouse button clicks that are generated by the host operating system. Compare <em>event</em>, <em>input event</em>.</td>
</tr>
<tr>
<td><strong>origin</strong></td>
<td>The point in a coordinate system whose ( x ) and ( y ) coordinates are both zero.</td>
</tr>
<tr>
<td><strong>owner-drawn button</strong></td>
<td>Owner-drawn buttons are usually represented by small bitmaps that the application defines.</td>
</tr>
<tr>
<td><strong>pane</strong></td>
<td>A region within a window that organizes the window’s contents. Panes can display text or lists; they can be buttons or labels or any other user interface components. Contrast <em>window</em>.</td>
</tr>
<tr>
<td><strong>parent</strong></td>
<td>The main window that encloses or surrounds a pane in an application. A pane’s position is specified relative to its parent window. See <em>pane</em>, <em>main window</em>.</td>
</tr>
<tr>
<td><strong>pen</strong></td>
<td>A graphics tool object used to draw text and graphics in a window. Pens have attributes such as width and color.</td>
</tr>
<tr>
<td><strong>persistence</strong></td>
<td>Refers to objects whose existence and internal state are maintained on a permanent medium (such as on disk). They are not affected by turning the system off. Normally a program will create many objects when it is run and then destroy those objects when it terminates. Persistent objects outlive the program that creates them, making it possible to share persistent objects among different programs and even different computers. See <em>Object Filer</em>.</td>
</tr>
<tr>
<td><strong>pixel</strong></td>
<td>The smallest element on a display that can be manipulated independently.</td>
</tr>
<tr>
<td><strong>pixel value</strong></td>
<td>The value of the bits specifying a pixel. This value specifies the color of the pixel. See <em>pixel</em>.</td>
</tr>
<tr>
<td><strong>point</strong></td>
<td>A unit of measure in typography that is approximately ( 1/72^{\text{nd}} ) of an inch. Font sizes are typically measured in points. Also the location of a pixel relative to the origin of the window in which the point appears. See <em>font</em>, <em>point size</em>.</td>
</tr>
<tr>
<td><strong>point size</strong></td>
<td>The size of a font, measured in points. It is the distance from the bottom of the lowest descender to the top of the highest ascender. See <em>font</em>, <em>point</em>.</td>
</tr>
</tbody>
</table>
pointer
An immediate data type consisting of the 32-bit address of a data structure. See data structure, immediate data type.

polymorphism
The ability of two or more classes of objects to respond to the same message in different ways. Because the receiving object responds to a message in its own way (according to the behaviors defined or inherited by its class), the same message can be used elsewhere in the system. Polymorphism eliminates the need for case statements (based on the class of object) in a method. Note that dynamic binding (or dynamic typechecking) is required for all objects that use polymorphism.

pool
A group of related variables available to any class whose definition specifies that it accesses the pool. See pool variable.

pool dictionary
A dictionary defined as a global variable that is only available to classes and methods in which the variable is identified as a pool in the class definition. Variables defined in a pool dictionary are only used at compile-time to replace the value of symbolic names in class and method definitions. Pool variables are inherited by subclasses. Compare pool variable.

pool variable
Pool variables are the keys defined in a pool dictionary. For example, WinConstants is a pool dictionary whose keys are pool variables. See pool dictionary.

procedural programming
A programming paradigm where the emphasis is on function rather than data.

property
An optional attribute of a window or pane, such as its initial size or background color. Properties have a name and a value. See pane, window.

proportional font
A font whose characters differ in width, according to the width each naturally requires for its form. This text is printed in a proportional font, so an uppercase “M” is wider than a lowercase “l.” See character, font. Compare fixed-width font.

protected block
A block that performs some action that requires clean-up, or that requires clean-up if it is interrupted before it completes. The protected block is the receiver of ensure: or ifCurtailed: messages, and the protection block is the argument to those messages. See block, guaranteed execution, protection block.

protection block
A block that is guaranteed to execute in order to perform necessary clean-up when a protected block executes. The protected block is the receiver of ensure: or ifCurtailed: messages, and the protection block is the argument to those messages. See block, guaranteed execution, protection block.
**protocol**
The complete description of the messages for any object that is an instance of a given class. The protocol also defines how new instances of the class are created. Protocol definitions for a class always have two parts: class methods and instance methods.

**push button**
A Windows control that invokes an action when pressed by the user. When you click on a push button, the selected state displays, giving the appearance of having pressed the button.

**radio button**
Radio buttons allow the user to exclusively select one of a group of choices. Selecting a radio button from the group deselects the previous selection.

**receiver**
In message syntax, the receiver identifies the object to which the message is sent.

**resources**
The operating system objects that an application creates and accesses to present the user interface. User resources include window and menu handles.

**restore button**
The button in the upper right corner of a window that, when selected, restores the window to its previous size.

**resumable exception**
An exception that can return from the message that signaled it and continue execution. See exception, signal. Compare nonresumable exception.

**RGB**
The color model in which the three primary colors are red, green, and blue.

**sans serif**
Characters lacking serifs. This text is sans serif. See character, font, serif.

**save image**
A Visual Smalltalk operation in which you save the current state of the Smalltalk environment (the image). All objects are restored to their current state when the system is restarted.

**screen**
A physical device that makes a portion of the display visible to the user. See display.

**scroll bar**
A control along the right or bottom edge of a window that allows the user to scroll the contents of the window.

**scrolling**
The process of sequentially moving a document in a window vertically or horizontally so that different parts of the document can be viewed.

**selector**
A message selector specifies the operation to be performed. See message.
Selector Browser  A Selector Browser is a three-pane window that provides a way to identify the complete message for each selector in the method displayed in the browser's text pane. The selectors list pane shows all the selectors of messages sent in the method on which the browser was opened. In addition, a methods list pane provides a convenient way to identify senders and implementors of the selector highlighted in the selectors pane. Also called *Message Browser*. See *browser*.

**self**  The word *self* in a method refers to the receiver of the message. The value of *self* is always the object that received the message.

**self-defined structure**  An interface to a data structure defined in another programming language such as C.

**serif**  The short strokes appearing at the ends of the strokes that make up a character in a font. This text has serifs. See *character*, *font*. Compare *sans serif*.

**server**  In client/server protocols, the server carries out the requests that the client generates.

**shared variable**  Shared variables are variables shared by two or more objects. They exist until explicitly deleted. See *class variable*, *global variable*. Compare *instance variable*, *temporary variable*.

**show it**  A Visual Smalltalk operation in which the current selection in the active text pane is evaluated as a Smalltalk expression. The print string of the result is inserted into the text pane.

**signal**  To create an instance of an exception class to represent an anomalous occurrence. See *exception*.

**Skip**  In the Debugger window, *Skip* executes the next expression in the selected method or up to the next breakpoint, whichever comes first. Note that *Skip* may execute several expressions in lower-level methods.

**Smalltalk**  Language designed specifically to support a pure implementation of object-oriented programming. Developed by Alan Kay and other researchers at XEROX Palo Alto Research Center (PARC) in the early 1970s, Smalltalk became commercial available in the early 1980s.

**Smalltalk application memory**  The sum of the Smalltalk heap and Smalltalk object space. See *Smalltalk object space*, *Smalltalk heap*.

**Smalltalk heap**  That portion of Smalltalk application memory in which data structures other than Smalltalk objects can be created, and to which routines external to Smalltalk have access. See *Smalltalk application memory*. Compare *Smalltalk object space*. 
<table>
<thead>
<tr>
<th>Glossary Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Smalltalk link library</strong></td>
<td>A Smalltalk link library is a file containing Smalltalk objects. They are primarily intended for use as an application delivery mechanism. They support delivery of both class code and data resources.</td>
</tr>
<tr>
<td><strong>Smalltalk object space</strong></td>
<td>That portion of Smalltalk application memory available to create and manipulate Smalltalk objects. The image. See <em>Smalltalk application memory</em>. Compare <em>Smalltalk heap</em>.</td>
</tr>
<tr>
<td><strong>static binding</strong></td>
<td>The binding of a message to an object at compile time.</td>
</tr>
<tr>
<td><strong>status bar</strong></td>
<td>The pane at the bottom of a window that presents status information to the user (for example, information in a status bar might include the current page number and last page number in a document being edited).</td>
</tr>
<tr>
<td><strong>string</strong></td>
<td>A string is an instance of class <strong>String</strong> containing a sequence of instances of class <strong>Character</strong> that can be accessed by indexing. Strings are not necessarily constant. The contents of a string may be changed by sending a message to the string. Compare <strong>symbol</strong>.</td>
</tr>
<tr>
<td><strong>stroking</strong></td>
<td>The operation by which lines are drawn or figures are outlined in the specified foreground color. See <em>foreground color</em>. Compare <em>displaying</em>, <em>erasing</em>, <em>filling</em>.</td>
</tr>
<tr>
<td><strong>subclass</strong></td>
<td>A descendant of a class. See <strong>behavior</strong>, <strong>class</strong>, <strong>inheritance</strong>. Compare <strong>superclass</strong>.</td>
</tr>
<tr>
<td><strong>subclassing</strong></td>
<td>In Visual Smalltalk, subclassing refers to deriving a class through inheritance from another class.</td>
</tr>
<tr>
<td><strong>superclass</strong></td>
<td>The immediate ancestor of a class. See <strong>behavior</strong>, <strong>class</strong>, <strong>inheritance</strong>. Compare <strong>subclass</strong>.</td>
</tr>
<tr>
<td><strong>superclassing</strong></td>
<td>In Visual Smalltalk, superclassing allows you to use a method in a superclass that is redefined in a subclass.</td>
</tr>
<tr>
<td><strong>symbol</strong></td>
<td>An instance of class <strong>Symbol</strong> containing a sequence of instances of class <strong>Character</strong> that can be accessed by indexing. Symbols differ from strings in that their characters may not be changed. Each unique sequence of characters is a single instance of class <strong>Symbol</strong>. Compare <strong>string</strong>.</td>
</tr>
<tr>
<td><strong>symbol constant</strong></td>
<td>A symbol constant identifies the associated symbol object. The form of a symbol constant is a number sign (#) followed by the alphanumeric characters of the symbol.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>System Dictionary</strong></td>
<td>The System Dictionary defines methods for system-oriented functions such as compressing the change log and determining available memory. It also contains all of the names known globally in the system. This includes the names of the classes, global variables, and pool dictionaries. Smalltalk uses the global variable Smalltalk to refer to the System Dictionary.</td>
</tr>
<tr>
<td><strong>temporary variable</strong></td>
<td>A variable defined as part of a method or block, that exists while the method or block executes. See block, method, variable.</td>
</tr>
<tr>
<td><strong>title bar</strong></td>
<td>The caption below the upper border of a window that identifies the associated Windows program. Clicking on the title bar activates (selects) the window. Dragging the title bar moves the window to another location.</td>
</tr>
<tr>
<td><strong>Transcript</strong></td>
<td>The initial workspace that is displayed in the development environment. See workspace, workspace variable.</td>
</tr>
<tr>
<td><strong>typeface</strong></td>
<td>A specific weight or slant of a font family that shares the basic style of that family and includes a variety of sizes. The term face is a shortened form of the term typeface. See boldface, font, italic, family.</td>
</tr>
<tr>
<td><strong>unary expression</strong></td>
<td>A unary expression sends a series of unary messages.</td>
</tr>
<tr>
<td><strong>unary message</strong></td>
<td>A unary message is a message that has no arguments.</td>
</tr>
<tr>
<td><strong>underline</strong></td>
<td>A line appearing at, and extending below, the baseline of a character (or set of characters) on a line of text. See baseline, font, point size.</td>
</tr>
<tr>
<td><strong>variable</strong></td>
<td>The value of a variable is a single object. Variables are either private or shared. Private variables are accessible only to a single object. Shared variables are accessible to multiple objects. The initial letter in a private variable name is lowercase, while the initial letter in a shared variable name is uppercase. See class variable, class instance variable, global variable, instance variable, object, pool variable, temporary variable, workspace variable.</td>
</tr>
<tr>
<td><strong>variable name</strong></td>
<td>A variable name identifies a variable in an object.</td>
</tr>
<tr>
<td><strong>view</strong></td>
<td>The user interface of an application, generally consisting of one or more windows that provide information on the state of the underlying model. See application.</td>
</tr>
<tr>
<td><strong>virtual image</strong></td>
<td>The software representing the classes and methods that are part of the system. The virtual image contains the compiled methods, and provides access to the source code for these methods.</td>
</tr>
<tr>
<td><strong>virtual machine</strong></td>
<td>The underlying non-Smalltalk software that supports the virtual image.</td>
</tr>
</tbody>
</table>
**Walkback**
A Walkback window pops up automatically when errors are detected. When you need more information than provided in the Walkback, you explicitly request a Debugger window by selecting **Debug** in the *Walkback* menu of the Walkback window or by clicking the *Debug* button located in the text pane of the Walkback window. See **debugger**.

**window**
A region on the display that provides a visual representation of an application. A window has a title bar and can be resized. Windows are composite objects consisting of smaller regions called panes. A window is the ultimate parent of all the panes defined within it. See **application**, **pane**, **parent**. Compare **main window**.

**window border**
A line that encloses a window. A window border is often called the *frame*.

**window properties**
In the class **Window**, the properties instance variable holds a dictionary if the window instance has any properties. The keys of this dictionary are symbols (the property "names"), and the values are the property values. An instance of a window object can have an arbitrary number of properties associated with it, such as background color.

**Windows NT**
A version of Windows that supports 32-bit applications, a flat address space, and multiple threads of execution.

**workspace**
A window with a single text pane, within which text can be edited (and optionally loaded from or saved to a file), and from which Smalltalk expressions can be evaluated. The Transcript is the initial workspace window that is displayed in the development environment.

**workspace variable**
A temporary variable associated with a workspace and only accessible within that workspace. Visual Smalltalk automatically creates workspace variables when you assign a value to a name in a workspace. See **variable**.

**wrapper**
An encapsulation mechanism—a way of hiding low-level implementation for a specific purpose. See **encapsulation**.

**wrapper method**
A method that transforms one representation of its arguments into the representation required by the method that it wraps.
The books in this list relate to Visual Smalltalk development or object-oriented programming in general. This is a reference tool, and is not an endorsement for any particular book.


Korieneck, Gene & Tom Wrench. *A Quick Trip to ObjectLand: Object-Oriented Programming with Smalltalk/V*. PTR Prentice-Hall


**Periodicals**

The following publications provide continuing and up-to-date information on object-oriented programming.

*Journal of Object-Oriented Programming*. SIGS Publication, Inc. Subscription requests: JOOP, Subscriber Services, Department OOP, P.O. Box 3000, Denville, New Jersey, 07834.


*The Smalltalk Report: The International Newsletter for Smalltalk Programmers*. SIGS Publications, Inc. Subscription requests: JOOP, Subscriber Services, Department SML, P.O. Box 3000, Denville, New Jersey, 07834.
Index

Symbols
! (exclamation points) 76
- (dash) 51
( ) (parentheses) 43
... (ellipses) 55
@ (at) sign 108

A
abstract classes
  defined 145
accessor methods 21
actions 143
ad hoc initializers 26
Add object... (Developer menu) command
  building catalog parts 127
  placing object instances on workbench 95
Add... (Dictionary menu) command 65
addGroupEditor: message 123
addGroupValidator: message 123
adding
classes
  to Smalltalk library build scripts 31
  to class hierarchy 56–57
Create form parts... button 136
elements to dictionaries 65
events and messages
  to nested part interfaces 90
  to PARTSInterfaceList instances 94
items to walkback lists 69
link labels to parts 40
object instances to workbench 95
parts to catalogs 129
addNames:toPage: message 123
Address, class 91
alias link mechanism
dynamic message selectors and 132
dynamic messages and 131
** all ** category 53
API (Application Program Interface) 146
application framework 86–87
application frameworks
  defined 146, 153
application icons
  creating 102
  setting 102, 109
application image files, building 108
application image files, workbench 110
application layer, coordinator as 86
application objects 84
application parts, launching 111
Application Program Interface (API)
  See API calls
applications
  See also building, applications; debugging
    applications; delivering applications
    as collections of
      objects 15
      packages 26
      defined 145
designing object-oriented 83–87
launching from workbench or EXE files 40
organizing source files
  file types 30
  in Visual Smalltalk Enterprise 30
recorded in clusters 26
releasing memory resources 80
reusability of components 18–19
runtime support files 105–107
saving source code 70–72
architecture
  component 85–86
  four-layered 84–85
arguments
  capturing command-line 102
  defined 146
  method 156
  passed with setValue: message 133
to writeNewFile: message 127
arrays
  defined 146
assigning. See setting
asterisk (*) 49
at (@) sign 108
attributes
  defined 146
B
background colors 146
backups, Visual Smalltalk file 75
binary filed objects 72
binary messages
  defined 146
binary numbers in Byte Array inspectors 65
bind actions
  Module/Build Library... command
    defining 113
    registering event handlers using 100
bind files, creating 108
bind methods
  described 125–126
  in runtime libraries 124
binding
  creating bind file 108
  overview 107
  selecting libraries for static 107
BitBlt operations
  defined 146
bitmap files 101
bitmaps
  defined 146
blocks
  defined 147
BMP files. See bitmap files
BND files. See bind files
bold text 43
borders
  window 164
bound class method 100
Browse -> Implementers of... (Module menu)
  command 49
Browse -> Senders of... (Module menu)
  command 49
Browse Packages (Smalltalk menu)
  command 41
browsers
  defined 147
browsers. See Class Hierarchy Browser;
  Method Browser; Package Browser
bugs. See debugging applications
Build Library... (Package Browser Module
  menu)
  command 113
build scripts
  application Smalltalk libraries 104–105
  bind actions for SLLs in 100
  for catalog parts 127
  Contact Manager class library 112–113
  organizing code using 31
  for runtime and workbench libraries 125
  updating libraries using 71
building
  application image files 108–111
  applications
    application view 88–90
    build scripts and 71
business model 91–94
catalogs, using 36
object interfaces and changes to objects 22
Smalltalk libraries for parts
  bind and unbind methods 125–126
  part file naming conventions 125
  runtime vs. workbench 124–126
Smalltalk libraries, application 104–105,
  112–113
business layer 86
business model
  exposing interface of 93–94
  in four-layer architecture 84
business objects
  conceptual nature of 91
  controller objects and 95
  defined 84
BusinessContact, class 91, 92
Byte Array inspector 65
byte values in indexed instance variables 57
C
callbacks
  defined 147
cancelChanges message 139
capturing command-line arguments 102
cascading messages
  defined 147
catalogs
  described 36
  opening 36
  opening workbenches for parts in 35
  part files
    recreating 130
    Smalltalk, building in 127–130
    workbench, building in 127
category list pane, Package Browser 42
change log. See CHANGE.LOG file
CHANGE.LOG file
  compressing 79
defined 147
  recovering from crashes using 75–77
  selectively recovering changes from 74
  source code changes saved to 73
CHANGE.LOG file, backing up and restoring 75
changed: event
  links from entry field 143
  setValue: message and 134
  triggered by part value changes 133
changing
  color of text in Package Browser 43
  splash screen 101
Index

characters
  defined 147

class browsers 147
class definition editor 49
class definitions
  See also definitions; method definitions
  changes saved to CHANGE.LOG file 73
  contents of 20
  editing 51, 55, 58
  finding in Package Browser 49
  improving as parts 115
  saving 51
  viewing 56

class hierarchy
  adding classes to 56–57
  deleting classes from 58
  viewing 54

Class Hierarchy Browser
  See also Package Browser
  adding classes 56–57
  defined 148
  described 54
  editing classes 58
  examining classes 56
  methods
    adding 59
    deleting 60
  removing classes 58
  window contents 54–55

class initializers 26

class instance variables
  defined 148

class instances
  changes to class definition and 58
  placing on workbenches 37–38
  removing remaining 80–81

class libraries
  defined 148

Class menu
  File Out.../File Out All... commands 72
  Find Class.... command 56
  operations performed using 56
  Remove Class command 59

class methods
  custom property edit dialogs 121–122
  defined 148
  purpose 20

class variables
  defined 148

classes
  See also entries for specific classes; part
  classes

adding
  to build scripts 31
  to libraries 104
  to library files 71
concrete 149
coordinator 87
creating part 116–120
defined 147
defined in packages 24, 43
defining 49–51
deleting methods from 60
finalization and 80
finding in Class Hierarchy Browser 56
for each kind of object 20
initialization by class initializers 26
vs. instances of 15–16
removing from hierarchy 58
value holder icons for 37

client applications
  defined 148

client window pane area
  defined 148
clipping region 148
cloning parts 119–120
close: method 101
CloseStartupDialog() method 101
closing
  Debugger windows 70
  sign-on dialogs 102
  Walkback windows 68
  workbenches 41

.CLFS files 72

collectors
  See also modules; Package Browser;
  packages
  creating 44–45
  defined 25
  organizing packages into 26
CNTCTMGR.BND file 113
CNTCTMGR.EXE file 114
CNTCTMGR.SLL file 111
code, source
  file location of changes to 73
  method
    editing 55
    viewing 61
  module
    in committed revisions 27
    organized into 24
    organized into 43
  preventing loss of 74
  restoring to image after crash 76
saving
  Object Filer, by using 72
  in packages 71
  in Smalltalk link libraries 71, 104–105
  separating runtime and workbench 125
collections, inspecting ordered 65
colors, Package Browser text 43
combo boxes 148
command-line arguments, capturing 102
comments defined 148
Commit... (Package Browser Module menu) command 45
committing packages to repositories 25, 44, 45–46
common system dialogs defined 148
compiling
  expressions in inspectors 64
  methods
    on changes to class definitions 58
    after editing 60
    new 60
complete specifications 25
component architecture 85–86
components
  characteristics of good 85–86
  described 18–19
compressing change and recovery logs 79
concatenateLists: message 118
concatenateListsSeparated: message 118
crash. See image, Smalltalk, recovering from
Create form parts... button 136
createdParts message 139
CreatePart-Button button 136
createParts method 138
creating
  See also adding; defining
  bind files 108
  clusters 44–45
  executable parts 103–104
  forms. See form generation
  ICO files for parts 129
  inspectors on keys 65
  links 38–40
  method categories 51–52
  methods 52–53; 59
  objects in workbench windows 21
  packages 30
  parts in Smalltalk
    building libraries for 124–126
    catalog part file 126–130
    part classes 116–120
    property dialogs 120–124
    random number generator 116
    steps required for 115
  Property Editor part 123–124
  repositories 47–48
  shutdown dialogs 101
cursor, mouse
  defined 149
cursors, workbench 35

dash (-) 51
data abstraction 149
data structures
  defined 149
data structures, inspecting fields of external 66
  data. See data structures, inspecting fields of external; information
DDE (Dynamic Data Exchange)
  defined 151
Debugger menu 69–70
Debugger windows
  defined 149
  described 68–70
  opening 66
devices for debugging applications
  See also errors
tools for
  Debugger windows 68–70
  Walkback windows 66–68
  windows available for 66
  writing transcript messages for 32
decimal numbers in Byte Array Inspectors 65
declaring workspace variables 34
defining
  See also adding; creating
  business objects 91–93
  classes 49–51, 56–57
  constructPropertyManger class method 121
  objects 20
  part classes 116
  partEditProperties instance method 121
  variables 50
Definition (Package Browser Global menu) command 51
  See also class definitions; method definitions
  filing out 72
  finding in Package Browser 49
  in packages 24
deleting
  class instances 80–81
  classes from hierarchy 58
  elements from dictionaries 65
  methods from classes 60
delivering applications
  building delivery files from packages 71
  with components as parts 19
  overview 99
  preparing as stand-alone programs
    shutdown dialogs 101
    sign-on dialogs 101–102
    splash screen 101
    starting application 100
dependency file preparation
  application image file 108–111
  application Smalltalk libraries 104–105
  executable parts, creating 103–104
  runtime support files 105–107
  Smalltalk library bind options 107–108
dependencies, library 129, 130
depth, color 149
Developer menu (Add object... command) 127
development environment
  See also image, Smalltalk contents of workbench 34
  starting vii
team.  See also Visual Smalltalk Enterprise xi
Device-Independent Bitmap (DIB) 149
dialogs
  defined 150
DIB (Device-Independent Bitmap) 149
dictionaries
  defined 150
Dictionary menu
  Dictionary objects, inspecting 65
  Cincom copyright notice 101, 102
Disk Browser 150
displaying
  defined 150
DLLs (Dynamic link libraries)
  defined 151
Do It 150
documentation
  on-line xi
  related ix–x
Double-byte character support
  defined 150
drag/drop
  defined 150
drawn buttons 158
Drop Process (Debugger menu) command 70
dumping objects
  in Byte Array inspectors 65
  Object Filer 72
dumpStringBase: aRadix message 66
dynamic binding 150
dynamic messages 131–133
dynamically bound libraries 107
e
Edit menu (inspectors) 64
editing
  CHANGE.LOG, SOURCES.SML, RECOVER.LOG and warning against 74
class definitions 51, 55, 58
  method source code 55, 60–61
  methods 53, 60
  objects in inspectors 64
  revision numbers 46
  element accessing messages 132
element setting messages 133
  elements, dictionary 65
ellipses (...) 55
encapsulation
defined 151
encapsulation, workbench implementation of 21
ensure blocks
execution
after closing Walkbacks 68
after editing methods 68
preventing execution of 70
error: message 67
errors
See also debugging applications
during recovery from image crash 78
on removing classes 59
run-time, common 66
evaluating
expressions
Dolt 150
event handlers
overview 17
registering for startupApplication event 100, 112
EventManager, class
nonvisual part subclasses of 116
objects that should be subclasses of 91
events
See also entries for specific events
adding to nested part interfaces 90
to be supported on workbench 117
defined 151
described 17–18
exposing public interface and 117
holders and 96
input 151
list of
all an object can trigger 117
to be exposed 94
operating system 158
in public interface of objects 85
triggered by
business objects 92
parts 22–23
in user-driven applications 87
exception environment 145, 151
exception handlers
active 145
defined 151
exceptions
defined 151
exclamation points (!) 76
executable parts, creating 103–104
execution
controlling in Debugger windows 68–70
guaranteed 153
resuming after Walkbacks 68
of source in revisions 27
stopping
without executing ensure blocks 70
executing ensure blocks after 68, 70
opening Walkback windows 67
exited image on: comment 78
expressions
ad hoc initializers as 26
defined 151
evaluated in inspectors 64
in restored code 77
unary 163
ExternalBuffer, class 66
F
Field Inspector 66
fieldEventsList class method 92
fields
inspecting external data structure 66
specifying pages for property edit dialog 123
fields message 66
file association 152
File It In (Smalltalk menu) command 77
File It In (Smalltalk menu) item 152
File menus
Insert Part File... command 129
Open... command 76
Save command 45, 60, 64
Save Image... command 70, 74
Workbench
Save as... command 40, 103
Save command 40
file names, saving parts to new and current 40
File Out... (Method menu) command 72
File Out/File Out All... (Class menu) commands 72
file-based repositories 27
file-ins 152
fileName method 137
file-outs 152
files
backups of Visual Smalltalk 75
creating bind 108
in delivered applications 99
packaging run-time 114
random number generator 116
runtime support 105–107
storing repository information in data 26
workspace contents as text 34
filing out 72
filling 152
finalization 80
Find Class (or Global)... command 49
Find Class... (Class Hierarchy Browser Classes menu) command 56
Find Module... (Package Browser Module menu) command 49
fixed-width fonts 152
focus panes 152
fonts
defined 152
foreground colors 153
form generation
customizing 139–140
partEditProperties, implementing 143
specifying generated part characteristics 141–143
undoing 139
Form Item Layout dialog 140
format: message 122
forms. See form generation
frames. See also borders 164
fromPartFile: message 141–142
FVIEW.PAR part 90

G

garbage collection
defined 153
garbage collection, forcing 81
generatePartsFor: message 137
generatePartsFor:fromDialog: message 138
generating forms. See form generation
get: message 122
getCommandLine method 102
Global menu (Package Browser)
Definition command 51
Find Class (or Global)... command 49
New -> Class command 49
global variables
defined 153
globals 153
graphical user interface. See GUI (graphical user interface)
green text 45
guaranteed execution 153
GUI (graphical user interface)
building application view 88–90
objects serving roles in 84
visual parts and 22

H

halt message
opening Walkback windows using 67
resuming execution after
in Debugger windows 69
in Walkback windows 68
handles 153
help, on-line xi
hexadecimal numbers in Byte Array Inspectors 65
hierarchical method categories 51
hierarchy
defined 148
hint line, enabling/disabling 35
holders, dynamic object 96
Hop button 69
described 153

I

ICO files
creating for parts 129
setting minimize icons and 103
icon: message 103
iconFromFile: message 129
icons
application 102–103, 109
catalog part vs. classes 37
defined 153
loading from part files to image 129
minimize 102–103
representing classes in Workbench catalog 16
workbench cursor and part 35
image files, application 108–111
image, Smalltalk
See also development environment
defined 153
recovering from crash of
maximizing chances of 75
in Visual Smalltalk 75–77
in Visual Smalltalk Enterprise 77–78
saving 70, 74
to V.EXE file 73
when binding libraries 109
implementors, viewing all 61
indexed instance variables
defined 153
in inspectors instance variable lists 63
specifying
as object or byte indexable 50
instance variables as 57
indices, viewing collection element 65
information
business objects as modeling 84
collectors and flow of 95
coordinator and flow of 86, 87
information-hiding 21–22
information-hiding
defined 154
inheritance  defined 154
initializeContactEvents method 96
initializeParts method 138
initializers
class vs. ad hoc 26
Contact Manager bound 113
controlling running of 47
registering startUpApplication event
handler 100
setting object values using 31
InitiatorInformation, class 92
Insert Part File... (File menu) command 129
insertion point 154
Inspect It (Smalltalk menu) item
defined 154
Inspect menu 64
inspectorFields message 66
Inspectors
defined 154
inspectors
described 63
editing objects in 64
opening 64
installing
defined 154
installing Visual Smalltalk viii
installPartRuntimeLibrary class library 125–126
instance messages 20–21
instance methods
defined 154
implementing instance messages 20–21
viewing 55
instance variables
Contact and BusinessContact classes 92
defined 155
specifying as indexable 50, 57
storage of property values in 20
viewing
all for object 63
current value of 64
instances, class
defined 154
instances, classes vs. 15–16
integers
defined 155
interfaces
See also GUI (graphical user interface)
component 19
dynamically changing 131
encapsulation and hiding of objects
behind 21–22
exposing
business model 93–94
view 89–90
on workbench 117–120
parts with multiple values 135
interfaces dialog 89–90
italic text 43
J
Jump button 69
defined 155
K
keyboard accelerator commands
defined 145
keyMessages method 132
keys, creating inspectors on 65
keyword expressions 155
keyword messages 62
L
labels, link 40
Launch (Workbench Test menu) command 40
launching
See also opening
application parts 111
applications in delivery preparation 100
CNTCTMGR.EXE file 114
development environment viii
parts 40
libraries, Smalltalk
bind options for 107–108
building for parts 124–126
delivering components as 19
organizing code and 31
registering event handlers with bind
actions for 100
saving source code in 71
library dependencies 129, 130
License Agreement 99
link labels 40
links
automatic creation of, with forms 135
between parts
events to messages 22–23
Nested Part and underlying parts 90
connecting view and model objects 95
creating 38–40
PARTSArrayHolderPart 132
list panes
defined 155
literals
defined 155
Load... (Package Browser Module menu)
command 46
loading
  icons from part files 129
  objects in Object Filer 72
packages 46
  revisions without opening 27
locking types, repository 48
logic
  application 84
  business 84
lookup:arguments: method 132
loose methods
  capturing command-line arguments 102
  defined 24
  registering startUpApplication event handler 100
lowerBound: message 143

M
  map files 106
menu bars
  defined 155
menus. See entries for specific menus
Message Browser 61–62
message selectors
  as method names 17
  Class Hierarchy Browser methods pane 59
  defined 156, 160
  dynamic 132
  Message Browser 61–62
  Method Browser 61
  method walkback windows 67
messages
  See also methods; entries for specific messages
  adding to nested part interfaces 90
  as bind actions for SLLs 100
  to be supported on workbench 117
  defined 17, 156
  defining business model interface 93
  dynamic 131–133
  element accessing 132
  exposing public interface and 117
  list of
    all defined for an object 117
    to be exposed 94
  parts responses to 22
  in public interface of objects 85
  sent as results of events 18
  transcript display of system and custom 32
  in user-driven applications 87
  viewing all sent in a method 61
Messages (Method menu) command 62
message-sends
  in Debugger windows 68
  in Walkback windows 67
method arguments 156
Method Browser 60
Method Browsers
  defined 156
method categories
  creating 51–52
  defining methods in 52
method definition template 53
method definitions. See class definitions;
definitions; method definition template
method editor window 53
Method menus
  File Out... command 72
  Messages command 62
  New command 52
  New Method command 59
  opening Method Browser from 60
  Remove command 60
method temporaries
  defined 156
method walkbacks 67
methods
  See also class methods; messages; entries for specific methods
  adding
    to library files 71
    to Smalltalk library build scripts 31
bind 124
class definitions 20
  compiling
    on changes to class definitions 58
    after editing 60
    new 60
  coordinator class 87
  creating 59
  defined 16–17
  defining 52–53
  editing 60
    in Method Browser 60
    in Package Browser 53
  exposing object interfaces in Workbench 93
  filing out 72
  finding definitions for 49
loose
  capturing command-line arguments 102
  defined 24
  registering startUpApplication event handler 100
private 93
viewing
all messages sent in 61
source code for 55
wrapper 164
minimize icons 102–103
modal dialogs
defined 156
model
See also business layer
connecting to view 95
coordinating with view 87, 94–97
Smalltalk code and building 86
modeless dialogs 157
module definition editor 44
module list pane 41
Module menu (Package Browser)
Build Library... command 113
Commit... command 45
Find Module... command 49
Load... command 46
New command 44
modules
always present in development environment 25
building application Smalltalk libraries in 104–105
controlling revisions of 27
as described in specifications 25
packages and clusters as 23
More levels (Debugger menu) command 69
.MTH files 72

N
named instance variables
defined 157
naming
applications 109
part files 125
Nestled Part 89
nested parts 103
New (Package Browser Method menu) command 52
New (Package Browser Module menu) command 44
New -> Class (Package Browser Global menu) command 49
New Method (Class Hierarchy Browser Method menu) command 59
New Workspace (Smalltalk menu) command 34
NLS (National Language Support)
defined 157
nonresumable exceptions
defined 157
nonvisual parts
See also parts; visual parts
default property edit dialog for 121
defined 22
defining part classes for 116
Number, class
defined 157

O
Object Filer 72
defined 157
object pointers
defined 159
Object, class 54
object-oriented programming
designing applications using 83–87
overview 15
workbench and 21–23
object-oriented programming (OOP)
defined 158
objects
constructing complex from simple 21
declaring what events can be triggered by 92
defined 157
defining 20
dynamic, representing 96
examining behavior of 63–66
improving as workbench behavior of 115
Object Filer operations on 72
overview 13
persistence of 158
placing instances of, on workbench 95
properties of 14
roles in applications 83–85
setting values of 31
as triggering events 17–18
octal numbers in Byte Array Inspectors 65
on-line help and documentation xi
OOP (object-oriented programming)
defined 158
Open (Smalltalk menu) command 47
Open -> Recovery Log (Smalltalk menu) command 78
Open Catalog... (Workbench View menu) command 37
open event 40
open message
as default interface item 90
launching parts and 40
sending icon: message using 103
Open... (File menus) command 76
opening
See also launching
catalogs 36
close log 76
Class Hierarchy Browser 54
Debugger windows 66, 68
inspectors 64
Method Browser 60
new dialog part files 137
Package Browser 41
Recovery Log Workspace 78
revisions without loading 27
sign-on dialogs 102
Walkback windows 67
workspace windows 34
openOn: method 66, 103
OpenStartupDialog() method 101
operating systems
events 158
OrderedCollection, class
converting inherited list of messages to
inspecting instances of 65
origin coordinate
defined 158
OS/2, IBM
application runtime support files for 105
building application image file 108
Contact Manager bind file 113
custom property edit dialogs for 121
files to delete from application directory
114
owner-drawn buttons. See drawn buttons

P
Package Browser
See also Class Hierarchy Browser; Method Browser
defining classes 49–51
editing class definitions 51
finding definitions in 49
methods
creating 52–53
editing 51–52, 53
opening 41
packages
committing 45–46
creating 44–45
loading 46
repositories
connecting to 48
creating 47–48
text display conventions 43
window contents 41–43
Package Commit Dialog 45
packages
See also clusters; modules; Package Browser
building application Smalltalk libraries in
104–105
committing to repositories 25, 45–46
creating 30, 44–45
defined 24
loading 46
saving source code in 71
packaging run-time files 99, 114
pages, property edit dialog 123
pages, window
defined 158
PAR files. See part files
parent pages 158
parentheses ( ) 43
Part Accessor parts 104
part classes
creating
enabling cloning 119–120
exposing object interfaces 117–120
runtime libraries 124
part files
catalog
recreating 130
Smalltalk, building in 127–130
workbench, building in 127
naming conventions 125
opening workbenches for existing 34
property editor dialogs in, saving 124
saving 40
dictionary method 142–143
partEditProperties instance method 121
partEvents method
exposing object interfaces 93, 117
holders and 96
listing all events 117
partFormGeneratorItemsFor: method 139
partFormGeneratorNames method 139
partLinkClassFor: method 131, 132
partMessages method
dynamic messages and 131
exposing object interfaces 93, 117
listing all messages 117
partMirrorCopy: message 119, 120
partMirrorCopyReal: message 119, 120
partRealSelectorFor: method 131, 132
parts
See also nonvisual parts; visual parts
converting to executable 103–104

Visual Smalltalk Enterprise User’s Guide 177
Index

creating in Smalltalk
  building Smalltalk libraries for 124–126
catalog part file 126–130
property dialogs 120–124
random number generator 116
steps required for 115
delivering components as 19
events, messages, and 22
exposing interface of nested 89
generating forms for
  adding Create form parts... button 136
customizing form generation 139–140
  partEditProperties, implementing 143
  specifying generated part characteristics 141–143
launching 40
linking between events and messages 38–40
placing on workbenches 37
validating and setting values for 133–135
visual vs. nonvisual 22
PARTS Wrappers 107
PARTSAliasLink, class 132
PARTSApplication class method 129
PARTSArrayHolderPart part 132–133
PARTSF ile, class 129
PARTSFormGenerator, class
  creating instances of 138
  generatePartsFor: message and 137
  overview 136
PARTSFormGeneratorItem, class
  fromPartFile: message and 141
  mapping item names to instances of 140
  overview 136
PARTSIcon, class 129
PARTSInterfaceList, class 94, 117
PARTSRandomPropertiesDialog2, class
  items to add to 137–139
PARTSValueHolderPart, class 97
PAX files 103, 104
pens 158
persistence 158
PersonalContact, class 91
PhoneInformation, class 91
pixels
  defined 158
pointers
  from compiled methods to code in change log 73
  indexed instance variables and 57
See also object pointers
points
  defined 158
polymorphism
  defined 159
pool dictionaries
  defined 159
pool variables 51
  defined 159
pools 159
PPNNVPRT.PAR file 120
PPPRANDOM.PAR file 116
PPVSLPRT.PAR file 120
Preference Group editor 43
presentation layer
  differentiating from business layer 86
  four-layer architecture 84
presentation objects 84
private methods 93
procedures. See methods
programming tools 29
properties
  See also property edit dialogs
  adding parts automatically to applications 135
catalog part file 127
changes requiring recreating part files 130
  defined 159
objects as having 14
  specification 25
window 164
property edit dialogs
  adding Create form parts... button to 136
  creating
    custom 121–124
    from default dialogs 120
Property Editor part 123–124
property values
  set by accessor methods 21
  specifying format of 122
  storage in instance variables 20, 50
PropertyAccess, class 122
PropertyManager, class 122
protected blocks of code
  defined 159
protocols
  defined 160
public interface. See also interfaces 85
pushbuttons
  defined 160
PVCS repositories 27
PWCE33.SLL file 111
PWONPR1n.SLL file 109
PWWNPR1n.SLL file 109
Q
Query object 95
Query, class 93

R
radio buttons
  defined 160
Radix (Inspect menu) command 65
random number generator part
  constructPropertyManager method and 122
  creating using PARTSAplication method 129
  files included 116
  list messages for 118
RANDOM.ICO file 116
RANDOM.PAR file 116
RANDOMR.SLL file 116
receivers
  defined 160
receivers, messages and 17
RECOVERLOG file
  backing up and restoring 75
  purpose 74
  Visual Smalltalk Enterprise crashes and 77–78
Recover/Reset Log menu command 79
recovering from image crash
  maximizing chances of 75
  in Visual Smalltalk 75–77
  in Visual Smalltalk Enterprise 77–78
Recovery Log Workspace 78
red italic text 43
registering
  createParts method 138
  dynamic message selectors 132
  event handlers for startUpApplication 100, 112
Remove (Dictionary menu) command 65
Remove (Method menu) command 60
Remove Class (Class menu) command 59
removePartRuntimeLibrary class method 126
removing. See deleting
reports of binary filed objects 72
repositories
  committing
    module revisions to 27
    packages to 25, 45–46
  connecting to 47, 48
  creating 47–48
  described 26–27
  PVCS vs. file-based 27
  saving modules in 25

Repositories Browser 47
Reset Log (Recover menu) command 79
resources. See system resources
Restart (Debugger menu) command 70
Restore (inspector Edit menu) command 64
resumable exceptions
  defined 160
Resume (Debugger menu) command 69
reusability
  component 18
  component architecture and 85
revision numbers
  creation of 27
  editing 46
  incrementing 46
revisions, repository storage of 26
RGB (Red, Green, Blue) color model
  defined 160
run-time errors 66
run-time files
  binding 107–108
  packaging 99, 114
  preparing
    application image file 108–111
    application Smalltalk libraries 104–105
  creating executable parts 103–104
  overview 99
runtime libraries
  bind and unbind methods 125–126
  described 124

S
Save as... (Workbench File menu) command 40, 103
Save command
  Class Hierarchy Browser File menu 60
  inspector File menu 64
  Package Browser File menu 45
  Workbench File menu 40
Save Image As dialog box 109
Save Image... (File menu) command 70, 74
Save imager dialog 74
saved image on: comment 78
saving
  application icons 103
  edited class definitions 51
  image
    before risking crash of 75
    to V.EXE file 73
    when binding libraries 109
    when editing objects in inspectors 64
  method definitions 53
  module definitions 45
module revisions 27
part files 40
property editor dialogs 124
Smalltalk image 70
    defined 160
source code
    by filing out 72
    in packages 71
    Object Filer, by using 72
Smalltalk link libraries 71
workspace contents 34
scripts. See build scripts
scroll bars
    defined 160
scrolling panes
    defined 160
seed: message 143
select: message 139
selecting
    classes to browse 56
    code to restore after crashes 76
    libraries to statically bind 107, 109
    repositories for committing packages to 46
selectOffAll message 139
Selector Browsers
    defined 161
SelectorPropertyAccessor, class 122
self object
    in Byte Array inspector 65
    in inspector instance variables list 63
self variable
    defined 161
self-defined structures 161
SelfDefinedStructure, class 66
senders
    messages and 17
    viewing all for a message 61
Senders (Class Hierarchy Browser Method menu) command 61
server applications
    defined 161
server objects 84
Service Manager 32
services browser 32
SessionModel, class
    getCommandLine method 102
    startUpApplication method 100
set: message 122
setElement: message 133
setting
    application and minimize icons 102, 103
    object values using ad hoc initializers 26, 31
part values 133–135
    property values using accessor methods 21
setValue: message 133–135
shared variables 161
Show hints (Workbench View menu)
    command 35
Show It (Smalltalk menu) item 161
shutdown dialogs 101
signaling
    defined 161
sign-on dialogs 101–102
Skip button 69
    defined 161
SLL files. See libraries, Smalltalk
Smalltalk
    See also Visual Smalltalk
    building business model in 91–94
    coordinating view and model objects in 87
    creating parts in
catalog part file 127–130
part classes 116–120
property dialogs 120–124
random number generator 116
Smalltalk application memory 161
Smalltalk heap 161
Smalltalk language 161
Smalltalk language. See also Visual Smalltalk
Smalltalk link libraries. See libraries, Smalltalk
Smalltalk menu
    Browse -> Implementors of... command 49
    Browse -> Senders of... command 49
    Browse Packages command 41
    File It In command 77
    New Workspace command 34
    Open command
        Recovery Log 78
        Repository Browser 47
Smalltalk menu (Visual Smalltalk Enterprise) 79
Smalltalk object space 162
source code. See code, source
source files
    organizing
        file types 30
        in Visual Smalltalk 31
        in Visual Smalltalk Enterprise 30
        SOURCES.SML file and compressed 73
        SOURCES.SML file
            backing up and restoring 75
            cleaning up change log using 79
            creating and updating 73
specifications 25
splash screen, changing 101
stand-alone programs, preparing
  changing splash screen 101
  creating
    shutdown dialogs 101
    sign-on dialog 101–102
  overview 99
starting application 100
starting. See launching; opening
startUpApplication event 100
startUpApplication method 102, 112
static binding 162
  statically-bound libraries
    advantages and disadvantages 107
    selecting libraries to be 107, 109
status bars
  defined 162
strings
  defined 162
subclasses
  adding to class hierarchy 56–57
  defined 162
  as displayed in class hierarchy list 54
subclasseing
  defined 162
superclasses
  defined 162
superclasseing 162
symbol constants 162
symbols
  defined 162
System Dictionary 163
system requirements viii
system resources
  See also resource files
  runtime support files for 106

T
Team -> Compress Changes (Visual Smalltalk Enterprise Smalltalk menu) command 79
team members, sharing work among 25–26
temporary variables
  defined 163
Terminate (Debugger menu) command 70
test menu 40
testing
  application outputs in transcript 32
text files
  bind files as 108
  writing source code to 72
title bars
  defined 163
transcript 32

T
Transcript window
  defined 163
  triggerFieldEvents message 92

U
  unary expressions 163
  unary messages
    defined 163
  unbind methods 125–126
  underlined text 43
  undoing form generation 139
  "unpackaged" package
    contents of 25
    new classes and methods in 54
  update event 92
  updating libraries 71
  upperBound: message 143
user interface
  building application view 88–90
  objects serving roles in 84
  visual parts and 22

V
V.EXE file
  image saves to 73
  launching applications from 100
  Visual Smalltalk icon in 102
V.MAP file 106
validateValues: method 123
validating part values 133–135
value holder icons 37
value message 133–135
values
  of holders for dynamic objects 96
  setting 26
  validating and setting part 133–135
  viewing instance variable current 64
variable names
  defined 163
variable names, specifying 50
variables
  defined 163
  defining 50
  in class definitions 20
  shared 161
  viewing class 55
  workspace 34
View menu (Workbench)
  Open Catalog... command 37
  Show hints command 35
viewing
  byte array radices 65
class definitions 56
class hierarchy 54
Index

external data structure fields 66
hidden subclasses in class hierarchy list 55
link labels 39
message selectors for methods 61–62
message-sends leading to errors 67, 68
methods
  all 53
  source code for 61
ordered collection element index values 65
views
  See also presentation layer
  building application 88–90
  connecting to model 95
  coordinating with model 87, 94–97
  exposing interface for 89–90
  PARTS Workbench and building 86
  specifying icons for 103
views, application
  defined 165
virtual image 163
virtual machine
  defined 165
visual parts
  See also nonvisual parts; parts
  default property edit dialog for 121
  defining classes for 116
  forms as collections of 135
Visual Smalltalk
  See also Smalltalk; Smalltalk language
  application framework 86–87
  class representations in 16
  compressing change log in 79
  failure to start after crash 76
  installing viii
  organizing source code files in 31
  overview vii
Visual Smalltalk Enterprise
  building application Smalltalk libraries in
    104–105
  class hierarchy browser and 54
  compressing change log in 79
  crashes in, recovering from 77–78
  described 23–24
  dividing work among team members 25
  as extension of Visual Smalltalk
  environment 19
  initializers 26
  organizing source code files in 30
  overview vii
  repositories 26–27
  revision control 27
  saving Smalltalk source code in 71
  separating runtime and workbench code
    125
Visual Smalltalk Enterprise package 25
VNOGULMAP file 106
VWSIGNON.C file 101

W
Walkback windows
  defined 164
  walkback windows
    as debugging tools 66–68
    removing classes with active instances or subclasses 59
  wild cards 49
  window properties 164
Window, class 116
windows
  See also system resources; walkback windows; workbench windows
  in application view 88
  Class Hierarchy Browser 54–55
  Debugger 68–70
  defined 164
  main 155
  workspace 34
Windows, Microsoft
  application runtime support files for 105–106
  building application image file 108
  Contact Manager bind file 113
  custom property edit dialogs for 121
  files to delete from application directory 114
workbench libraries
  bind and unbind methods 125–126
  described 124
workbench windows
  closing 41
  constructing complex objects in 21
  cursor shapes 35
  parts of 35
Workbench, PARTS
  See also workbench windows
  building catalog parts in 127
  business model interface exposure in 93
  catalogs 36
  connecting events to messages in 18
  coordinating view and model objects in
    87, 94–97
  creating links on 38–40
  described 21–23
  launching parts 40
  overview 34–35

182 Visual Smalltalk Enterprise User's Guide
Index

placing parts onto 37–38
saving part files 40
workspace variables
  defined 164
workspace windows
  described 34
  Recovery Log 78
workspaces
  defined 164
wrapper methods 164
wrappers
  defined 164
wrapping external resources 84
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.

CINCOM
The Smart Choice®