

VisualWorks

Tutorial

Copyright © 1995 by ParcPlace-Digitalk, Inc. All rights reserved.

Part Number: DS20002003

Revision 2.1, December 1995 (Software Release 2.5)

This document is subject to change without notice.

RESTRICTED RIGHTS LEGEND:

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

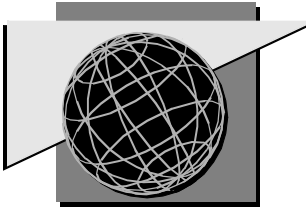
Trademark acknowledgments:

ObjectKit, ObjectWorks, ParcBench, ParcPlace, and VisualWorks are trademarks of ParcPlace Systems, Inc., its subsidiaries, or successors and are registered in the United States and other countries. DataForms, MethodWorks, ObjectLens, ObjectSupport, ParcPlace Smalltalk, Visual Data Modeler, VisualWorks Advanced Tools, VisualWorks Business Graphics, VisualWorks Database Connect, VisualWorks DLL and C Connect, and VisualWorks ReportWriter are trademarks of ParcPlace Systems, Inc., its subsidiaries, or successors. ENVY is a registered trademark of Object Technology International, Inc. All other products or services mentioned herein are trademarks of their respective companies. Specifications subject to change without notice.

The following copyright notices apply to software that accompanies this documentation:

VisualWorks is furnished under a license and may not be used, copied, disclosed, and/or distributed except in accordance with the terms of said license. No class names, hierarchies, or protocols may be copied for implementation in other systems.

This manual set and online system documentation copyright © 1995 by ParcPlace-Digitalk, Inc. All rights reserved. No part of it may be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from ParcPlace-Digitalk.



Contents

About This Book

ix

| | |
|-------------------------------------|------|
| Audience | ix |
| Organization | ix |
| Conventions | x |
| Typographic Conventions | xi |
| Special Symbols | xi |
| Screen Conventions | xii |
| Mouse Buttons | xii |
| Three-Button Mouse | xiii |
| Two-Button Mouse | xiii |
| One-Button Mouse | xiii |
| Mouse Operations | xiv |
| Additional Sources of Information | xiv |
| Printed Documentation | xiv |
| Online Documentation | xv |
| Obtaining Technical Support | xvi |
| Before Contacting Technical Support | xvi |
| How to Contact Technical Support | xvi |
| Electronic Mail | xvii |
| Electronic Bulletin Boards | xvii |
| World Wide Web | xvii |
| Telephone and Fax | xvii |

Chapter 1 Getting Started

1

| | |
|--|---|
| What Is VisualWorks? | 1 |
| VisualWorks as a Smalltalk Environment | 2 |
| Starting VisualWorks | 2 |
| Macintosh Platforms | 3 |
| OS/2 Platforms | 3 |
| UNIX Platforms | 3 |
| Windows Platforms | 3 |

- A First Look at VisualWorks 4
 - VisualWorks Main Window 4
 - Main-Window Menus 5
 - Main-Window Tool Bar 6
 - System Transcript 6
 - Workspace Window 6
- Interacting with VisualWorks 7
 - Mouse-Button Functions 7
 - Getting Some Practice 8
 - Managing VisualWorks Windows 11
- Saving Your VisualWorks Image 13
 - Creating Your Own Working Image 13
 - If You Created an Image in a New Location 14
 - Taking Snapshots 15
- Exiting VisualWorks 15
- What's Next: The VisualWorks Environment 16

Chapter 2 The VisualWorks Environment 17

- Starting Your Working Image 17
- Sending Messages to Smalltalk Objects 18
- Browsing the Smalltalk Class Library 19
 - Using a System Browser 19
 - Finding a Class by Name 23
 - Adding a Category 24
 - Browsing the Class Hierarchy 24
 - Using a Hierarchy Browser 24
- Storing and Retrieving Information in Files 26
 - Writing to Disk Files 26
 - Retrieving Information from Disk Files 26
- Running an Application 28
- Browsing Online Documentation 30
 - Exploring the Cookbook's Sample Applications 33
- Customizing Your Working Image 35
- Viewing Changes Since the Last Save 36
- What's Next: Creating Applications 37

Chapter 3 Introduction to VisualWorks Application Building 39

- Application Requirements 39
- VisualWorks Approach to Application Design 40
 - Layered Structure 40
 - Domain Models 42
 - Application Models 42

| | | |
|--|--|-----------|
| Why Layering? | 43 | |
| UI-Based Structure | 44 | |
| Why UI-Based Structure? | 44 | |
| Building Blocks in the Framework | 45 | |
| Framework for Database Applications | 46 | |
| Designing the Sample Application | 46 | |
| Designing the User Interface | 47 | |
| Designing the Models | 47 | |
| Designing Domain Models | 47 | |
| Designing Application Models | 48 | |
| What's Next: Constructing the Sample Application | 49 | |
| | | |
| Chapter 4 | Creating a Graphical User Interface | 51 |
| Designing the Checkbook Main Window | 51 | |
| Design Alternatives | 52 | |
| Creating the Main Window | 53 | |
| Opening a Blank Canvas | 53 | |
| Painting the Canvas | 55 | |
| Sizing the Canvas | 55 | |
| Painting a Widget | 55 | |
| Selecting and Deselecting a Widget | 56 | |
| Positioning a Widget | 57 | |
| Resizing a Widget | 57 | |
| Copying and Pasting a Widget | 57 | |
| Painting Multiple Copies of a Widget | 58 | |
| Deleting a Widget | 58 | |
| Setting Properties | 59 | |
| Displaying a Widget's Properties | 59 | |
| Applying a Changed Property | 60 | |
| Moving the Selection to the Next Widget | 61 | |
| Inspecting the List Properties | 61 | |
| Setting the Input Field Properties | 62 | |
| Setting the Window Properties | 63 | |
| Installing the Canvas | 64 | |
| Finding an Installed Canvas | 66 | |
| Editing a Menu Bar | 67 | |
| Opening the Interface | 70 | |
| Behind the Scenes | 70 | |
| Inspecting the Prototype Window | 71 | |
| Revising the Main Window | 72 | |
| Adding More Widgets | 72 | |
| Refining Widget Arrangement | 74 | |

- Selecting Multiple Widgets 74
- Equalizing Widget Sizes 75
- Aligning Widgets 76
- Spacing by Pixels 77
- Grouping Widgets 77
- Adjusting Window Layout 78
- Creating the Check Window 79
 - Painting and Setting Properties 79
- Previewing a Window for Another Platform 82
- What's Next: Programming in Smalltalk 82

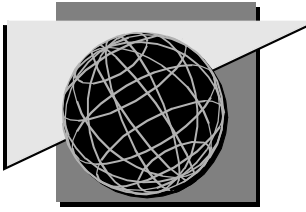
Chapter 5 Developing the Domain Models 83

- What You Should Read 83
 - If You Are New to Smalltalk 83
 - If You Already Know Smalltalk 84
- Creating the Check Class 85
 - Locating the Application's Category 86
 - Defining the Data Structure for the Check Class 87
 - Analysis: The Check Class Definition 88
 - Creating a Check Instance 89
 - Analysis: Message Expressions 90
 - Analysis: Messages for Creating Instances 90
 - Documenting the Check Class 91
 - Analysis: The Check Class Comment 91
 - Providing for Access to Check Data 92
 - Analysis: Message Protocol 94
 - Analysis: Method Definitions 95
 - Analysis: Naming Conventions 96
 - Analysis: Method Compilation 96
 - Setting Check Information 97
 - Analysis: More about Message Expressions 97
 - Providing for Character-Based Display 99
 - Analysis: Constructing a String 100
 - Analysis: Streams 100
 - Displaying a Check Instance's Description 101
 - Analysis: The **do it**, **print it**, and **inspect** Commands 102
 - Analysis: Method Lookup 102
 - Creating the Checkbook Class 104
 - Defining and Documenting the Checkbook Class 105
 - Analysis: Subclasses of Model 106

| | |
|--|-----|
| Creating a Checkbook Instance | 107 |
| Providing for Checkbook Initialization | 108 |
| Analysis: Initial Data Types | 109 |
| Analysis: Class and Instance Methods | 109 |
| Creating an Initialized Checkbook Instance | 110 |
| Analysis: More about Method Lookup | 110 |
| Providing for Access to Checkbook Data | 112 |
| Analysis: Limited Access to Variables | 113 |
| Analysis: Change Notification | 113 |
| Providing for Checkbook Transactions | 115 |
| Analysis: More about Complex Expressions | 117 |
| Analysis: Alternative Implementation | 119 |
| Testing the Checkbook Transactions | 120 |
| Analysis: Transcript Messages | 122 |
| Analysis: Syntax Errors | 123 |
| What's Next: Programming the Interface | 125 |

| | | |
|------------------|--|------------|
| Chapter 6 | Programming the Interface | 127 |
| | VisualWorks Approach to Interface Programming | 127 |
| | Specifying Basic Appearance and Behavior | 128 |
| | Programming Application-Specific Behavior | 128 |
| | Action Widgets | 129 |
| | Data Widgets | 130 |
| | Another Look at Application Structure | 131 |
| | Programming the Application Model | 132 |
| | Setting Up Your Work | 133 |
| | A Few Reminders | 133 |
| | Browsing the Application Model | 134 |
| | Providing the Checkbook Behind the Interface | 135 |
| | Analysis: Initializing an Application Model | 135 |
| | Programming the Amount to Deposit : Field | 137 |
| | Analysis: Aspect Property | 139 |
| | Analysis: The Definer | 139 |
| | Analysis: Lazy Initialization, Booleans, Blocks | 140 |
| | Analysis: Value Holders | 142 |
| | Programming the Deposit Button | 143 |
| | Analysis: Action Property | 144 |
| | Analysis: <code>makeDeposit</code> Logic | 144 |
| | Analysis: Warning Dialog | 146 |
| | Testing the Deposit Widgets | 147 |

| | |
|---|------------|
| Analysis: Behind the Scenes During Setup | 148 |
| Analysis: Behind the Scenes During Operation | 149 |
| Analysis: Widgets as Dependents | 150 |
| Analysis: Modifying a Running Application | 150 |
| Programming the Balance : Field | 151 |
| Analysis: The Definer Revisited | 153 |
| Analysis: Aspect Adaptors | 153 |
| Testing the Balance : Field | 155 |
| Analysis: Setup of the Aspect Adaptor | 155 |
| Analysis: Operation of the Aspect Adaptor | 156 |
| Programming the Check Register List | 158 |
| Analysis: Setup of the List | 160 |
| Analysis: SelectionInList Instances | 161 |
| Analysis: When the Collection Changes | 162 |
| Programming the Menu Bar | 164 |
| Setting Up for the Remaining Work | 166 |
| Providing for Writing New Checks | 167 |
| Setting Up the Check Dialog Box's Basic Behavior | 168 |
| Analysis: Actions for OK and Cancel Buttons | 170 |
| Analysis: Setup of the Dialog Box | 170 |
| Programming the Input Fields in the Check Dialog Box | 172 |
| Analysis: Aspect Paths | 175 |
| Analysis: Setup of an Aspect Path | 176 |
| Analysis: Subject Channels | 177 |
| Analysis: Advantages of Aspect Paths | 178 |
| Analysis: Limitations of Aspect Paths | 179 |
| Finishing the <code>writeNewCheck</code> Method | 180 |
| Providing for Check Cancellation | 181 |
| What's Next? | 182 |
| Appendix A Glossary | 183 |
| Appendix B Widget Quick Reference | 199 |
| Index | 207 |



About This Book

This tutorial introduces VisualWorks®, a fully object-oriented environment for constructing applications using the ParcPlace Smalltalk™ programming language.

This tutorial presents steps for constructing a sample application with VisualWorks. In the process, this tutorial introduces the VisualWorks tools, class library, and approach to application design. It also introduces basic object-oriented concepts and the Smalltalk language.

Audience

This tutorial is intended for anyone who is new to VisualWorks. This tutorial does *not* assume that you know object-oriented concepts, Smalltalk, or graphical user-interface application architecture.

If you are new to applications with graphical user interfaces, you may want to consult your platform's documentation for general information about using a mouse to interact with an application.

Organization

This tutorial falls into three parts.

Chapters 1 and 2 introduce VisualWorks:

- n Chapter 1, “Getting Started,” shows you how to start and exit VisualWorks, find your way around the VisualWorks main window, interact with VisualWorks’ graphical user interface, and save your work.
- n Chapter 2, “The VisualWorks Environment,” introduces the basic tools for exploring and configuring the VisualWorks environment. Some of these tools help you find and manipulate Smalltalk objects; others provide information about your VisualWorks image. Chapter 2 also

introduces basic Smalltalk concepts such as object, message, class, instance variable, and method.

Chapters 3 through 6 walk you through building a sample application:

- n Chapter 3, “Introduction to VisualWorks Application Building,” introduces the sample application that you will build. It describes the general design of a VisualWorks application and outlines the design that you will use for the sample application.
- n Chapter 4, “Creating a Graphical User Interface,” describes how to create the visual portion of a graphical user interface.
- n Chapter 5, “Developing the Domain Models,” explains how to create the Smalltalk classes that provide the basic processing for the application.
- n Chapter 6, “Programming the Interface,” explains how to integrate the graphical user interface with the Smalltalk classes created in Chapter 5.

Two appendixes explain terms used in this tutorial and in the VisualWorks interface:

- n Appendix A, “Glossary,” defines terms that are particular to VisualWorks applications and the Smalltalk language.
- n Appendix B, “Widget Quick Reference,” describes the various widgets available to you in the VisualWorks Palette.

Conventions

This section describes the notational conventions used to identify technical terms, computer-language constructs, mouse buttons, and mouse and keyboard operations.

Typographic Conventions

This book uses the following fonts to designate special terms:

| Example | Description |
|---------------------|--|
| <i>template</i> | Indicates new terms where they are defined, emphasized words, book titles, and words as words. |
| cover.doc | Indicates filenames, pathnames, commands, and other C++, UNIX, or DOS constructs to be entered outside VisualWorks (for example, at a command line). |
| <i>filename.xwd</i> | Indicates a variable element for which you must substitute a value. |
| windowSpec | Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks graphical user interface. |
| Edit menu | Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples. |

Special Symbols

This book uses the following symbols to designate certain items or relationships:

| Examples | Description |
|---|--|
| File?New command | Indicates the name of an item on a menu. |
| <Return> key <Select> button <Operate> menu | Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name. |
| <Control>-<g> | Indicates two keys that must be pressed simultaneously. |
| <Escape> <c> | Indicates two keys that must be pressed sequentially. |
| Integer>>asCharacter | Indicates an instance method defined in a class. |



| Examples | Description |
|-----------------|---|
| Float class>>pi | Indicates a class method defined in a class. |
| <i>Caution:</i> | Indicates information that, if ignored, could cause loss of data. |
| <i>Warning:</i> | Indicates information that, if ignored, could damage the system. |

Screen Conventions

This tutorial contains a number of sample screens that illustrate the results of various tasks. The windows in these sample screens are shown in the default Smalltalk look, rather than the look of any particular platform. Consequently, the windows on your screen will differ slightly from those in the sample screens.

Mouse Buttons

Many hardware configurations supported by VisualWorks have a three-button mouse, but a one-button mouse is the standard for Macintosh users, and a two-button mouse is common for OS/2 and Windows users. To avoid the confusion that would result from referring to <Left>, <Middle>, and <Right> mouse buttons, this book instead employs the logical names <Select>, <Operate>, and <Window>.

The mouse buttons perform the following interactions:

| | |
|------------------|--|
| <Select> button | <i>Select</i> (or choose) a window location or a menu item, position the text cursor, or highlight text. |
| <Operate> button | Bring up a menu of <i>operations</i> that are appropriate for the current view or selection. The menu that is displayed is referred to as the <i><Operate> menu</i> . |
| <Window> button | Bring up the menu of actions that can be performed on any VisualWorks <i>window</i> (except dialogs), such as move and close . The menu that is displayed is referred to as the <i><Window> menu</i> . |

Three-Button Mouse

VisualWorks uses the three-button mouse as the default:

- n The left button is the <Select> button.
- n The middle button is the <Operate> button.
- n The right button is the <Window> button.

Two-Button Mouse

On a two-button mouse:

- n The left button is the <Select> button.
- n The right button is the <Operate> button.
- n To access the <Window> menu, you press the <Control> key and the <Operate> button together.

One-Button Mouse

On a one-button mouse:

- n The unmodified button is the <Select> button.
- n To access the <Operate> menu, you press the <Option> key and the <Select> button together.
- n To access the <Window> menu, you press the <Command> key and the <Select> button together.

Mouse Operations

The following table explains the terminology used to describe actions that you perform with mouse buttons.

| When you see: | Do this: |
|----------------------|---|
| click | Press and release the <Select> mouse button. |
| double-click | Press and release the <Select> mouse button twice without moving the pointer. |

| When you see: | Do this: |
|----------------------|--|
| <Shift>-click | While holding down the <Shift> key, press and release the <Select> mouse button. |
| <Control>-click | While holding down the <Control> key, press and release the <Select> mouse button. |
| <Meta>-click | While holding down the <Meta> or <Alt> key, press and release the <Select> mouse button. |

Additional Sources of Information

Printed Documentation

In addition to this tutorial, the core VisualWorks documentation includes the following documents:

- n *Installation Guide*: Provides instructions for the installation and testing of VisualWorks on your combination of hardware and operating system.
- n *Release Notes*: Describes the new features of the current release of VisualWorks.
- n *Cookbook*: Provides step-by-step instructions for performing hundreds of common VisualWorks tasks.
- n *User's Guide*: Provides an overview of object-oriented programming, a description of the Smalltalk programming language, a VisualWorks tools reference, and a description of various reusable software modules that are available in VisualWorks.
- n *International User's Guide*: Describes the VisualWorks facilities that support the creation of nonEnglish and cross-cultural applications.
- n *Object Reference*: Provides detailed information about the VisualWorks class library.

The documentation for the VisualWorks database tools consists of the following documents:

- n *VisualWorks' Database Tools Tutorial and Cookbook*: Introduces the process and tools for creating applications that access relational data-

bases. The “Cookbook” chapter describes how to programmatically customize various aspects of a database application.

- n *Database Connect User’s Guide*: Provides information about the external database interface. Versions of it exist for Oracle7, SYBASE, and DB2 databases.

Online Documentation

To display the online documentation browser, open the **Help** pull-down menu from the VisualWorks main menu bar and select **Open Online Documentation**. Your choice of online books includes:

- n *Database Cookbook*: Online version of the “Cookbook” part of the *VisualWorks’ Database Tools Tutorial and Cookbook* described above.
- n *Database Quick Start Guides*: Describes how to build database applications. It covers such topics as data models, single- and multiwindow applications, and reusable data forms.
- n *International User’s Guide*: Online version of the *International User’s Guide* described above.
- n *VisualWorks Cookbook*: Online version of the *Cookbook* described above.

Obtaining Technical Support

If, after reading the documentation, you find that you need additional help, you can contact ParcPlace-Digitalk Technical Support. ParcPlace-Digitalk provides all customers with help on product installation. ParcPlace-Digitalk provides additional technical support to customers who have purchased the ObjectSupport package. VisualWorks distributors often provide similar services.

Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- n The *version id*, which indicates the version of the product you are using. Choose **Help?About VisualWorks** in the VisualWorks main window. The version number can be found in the resulting dialog under **Version Id**:

- n Any modifications (*patch files*) distributed by ParcPlace-Digitalk that you have imported into the standard image. Choose **Help?About VisualWorks** in the VisualWorks main window. All installed patches can be found in the resulting dialog under **Patches**:
- n The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select **copy stack** in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.

How to Contact Technical Support

ParcPlace-Digitalk Technical Support provides assistance by:

- n Electronic mail
- n Electronic bulletin boards
- n World Wide Web
- n Telephone and fax

Electronic Mail

To get technical assistance on the VisualWorks line of products, send electronic mail to **support-vw@parcplace.com**.

Electronic Bulletin Boards

Information is available at any time through the electronic bulletin board CompuServe. If you have a CompuServe account, enter the ParcPlace-Digitalk forum by typing **go ppdforum** at the prompt.

World Wide Web

In addition to product and company information, technical support information is available via the World Wide Web:

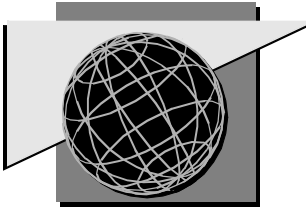
1. In your Web browser, open this location (URL):
`http://www.parcplace.com`
2. Click the link labeled “Tech Support.”

Telephone and Fax

Within North America, you can:

- n Call ParcPlace-Digitalk Technical Support at 408-773-7474 or 800-727-2555.
- n Send questions and information via fax at 408-481-9096.
Operating hours are Monday through Thursday from 6:00 a.m. to 5:00 p.m., and Friday from 6:00 a.m. to 2:00 p.m., Pacific time.

Outside North America, you must contact the local authorized reseller of ParcPlace-Digitalk products to find out the telephone numbers and hours for technical support.



Chapter 1

Getting Started

This chapter provides an introductory description of VisualWorks and shows you how to:

- n Start VisualWorks
- n Find your way around the VisualWorks main window
- n Interact with the VisualWorks graphical user interface
- n Save your work
- n Exit VisualWorks

What Is VisualWorks?

VisualWorks is a fully object-oriented environment for constructing applications, using the ParcPlace Smalltalk programming language. With VisualWorks, you can rapidly build graphical user interfaces for new and existing applications. In addition, you can use VisualWorks to link your application to various relational databases.

With the following VisualWorks features, you can build applications quickly and easily:

- n Point-and-click tools for incorporating controls (or *widgets*) into your application's graphical user interface
- n A predefined application framework that you can adapt for your own application
- n Mechanisms for reusing applications and interfaces
- n Links to relational databases such as Oracle7, SYBASE, and DB2.
- n Instant portability of your application over UNIX, Microsoft Windows, OS/2, and Macintosh platforms

VisualWorks as a Smalltalk Environment

VisualWorks capabilities are implemented in the ParcPlace Smalltalk programming language and therefore are part of the *ParcPlace Smalltalk system*. This system consists of interacting *objects*—software units that contain collections of related data plus operations for manipulating that data. These objects collaborate to perform a wide variety of functions.

Some of the objects in the ParcPlace Smalltalk system exist so that you can incorporate them into your own programs. In fact, part of your work in VisualWorks will consist of familiarizing yourself with the objects in the system library, adapting these objects to suit your needs, and adding new objects to the system.

Other objects (sometimes called *system objects*) provide functions that are usually associated with a software development system: an editor, a compiler, a debugger, print utilities, a window system, and so on. ParcPlace Smalltalk is more than a programming language; it provides VisualWorks with a complete programming environment.

Note: Throughout this tutorial, the term Smalltalk refers to ParcPlace Smalltalk.

Starting VisualWorks

This tutorial assumes that VisualWorks is already installed on your machine or system. Starting VisualWorks means running an *object engine* with an *image*, where:

- n An *image* is a file or document that contains the entire VisualWorks environment; it stores compiled versions of the objects in the Smalltalk system.
- n The *object engine* is an executable program that runs Smalltalk on your platform; it essentially “sets in motion” the system objects in an image.

The first time you start VisualWorks, you use the *standard image*—that is, the image that was delivered with VisualWorks. Thereafter, you normally do your work in your own *working image*, returning to the standard image only when you want to create a new working image from it.

Now start the standard VisualWorks image, following the directions for your platform. (Start this image even if you already created a custom working image during installation.)

Macintosh Platforms

To start the standard image on a Macintosh computer:

1. Open the **image** folder in the VisualWorks installation folder (typically called **visual**).
2. Double-click the **visual.im** document.

OS/2 Platforms

To start the standard image on an OS/2 platform:

- % Double-click the **VisualWorks** program object in the **VisualWorks 2.5** folder.

UNIX Platforms

To start the standard image on a UNIX platform:

1. Verify that your window manager is operating.
2. Enter a command of the following form at the UNIX prompt:

```
% visualworks image-path
```

where **image-path** is the pathname of the standard image (for example, **/usr/visual/image/visual.im**).

Windows Platforms

To start the standard image on a Windows 95 platform:

- % Double-click the **visual** document in the **Image** folder in the VisualWorks installation folder (typically called **visual**).

To start the standard image on other Windows platforms:

- % Double-click on the **VisualWorks** program-item icon in the **VisualWorks 2.5** program group in the Program Manager.

A First Look at VisualWorks

On all platforms, starting VisualWorks from the standard image opens the VisualWorks main window and a Workspace, as shown in Figure 1-1. These windows are described in the following sections.

Note: In the sample screens that follow, the windows are shown in the default Smalltalk look, rather than the look of any particular platform. Consequently, the windows on your screen will differ slightly from those in the sample screens.

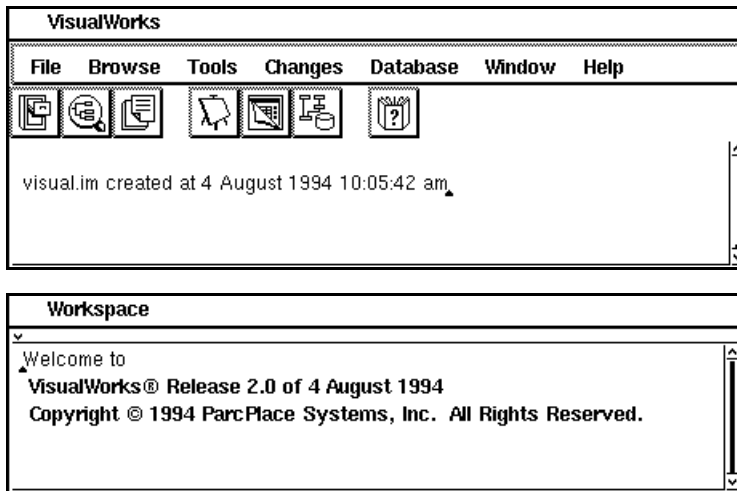


Figure 1-1 The windows displayed in the standard image

VisualWorks Main Window

The VisualWorks main window is the starting point for your work. It remains on your screen as long as VisualWorks is running.

The VisualWorks main window is identified by the title **VisualWorks** in the window's title bar, as shown in Figure 1-2. It also contains a menu bar, a tool bar, and a message area known as the *System Transcript*.

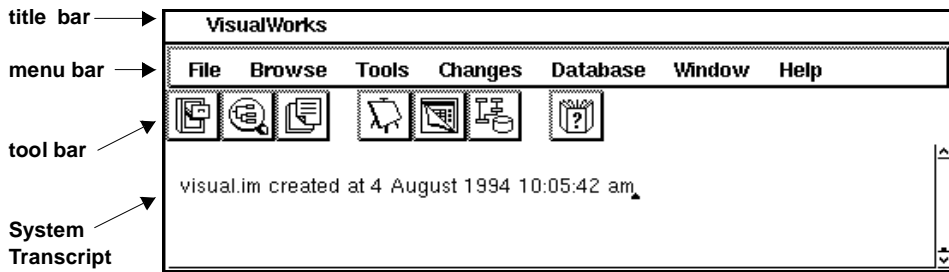


Figure 1-2 The VisualWorks main window

Main-Window Menus








The menus on the VisualWorks main window provide access to all VisualWorks tools and capabilities. These menus are summarized in the following table:

| Menu | Contains commands for: |
|------------------|---|
| File | Saving, customizing, and exiting the VisualWorks image |
| Browse | Browsing the Smalltalk programs in the image |
| Tools | Opening a variety of development tools, including file editors and tools for creating graphical user interfaces |
| Changes | Opening tools for viewing changes made to the image and for organizing your work into separate projects |
| Data-base | Invoking VisualWorks' database tools, which help you create applications that access relational databases |
| Window | Managing the open VisualWorks windows |
| Help | Opening the various online documents and guides |

***Note to Macintosh users:** For all normal VisualWorks operations, you use the menus that are displayed in the VisualWorks main window (or in other individual VisualWorks windows). You use the menus that VisualWorks places on the Macintosh menu bar only if VisualWorks fails to respond to the normal menus. See the VisualWorks Installation Guide (Macintosh) for information about the VisualWorks menus on the Macintosh menu bar.*

Main-Window Tool Bar

The tool bar on the VisualWorks main window provides access to the more frequently used tools. Each button on the tool bar is a shortcut for a particular menu command. These buttons are briefly described in the following table:

| Button | Invokes this tool: |
|---|--|
|  | File List, for creating, editing, and viewing files in the platform's file system |
|  | System Browser, for browsing and creating Smalltalk programs |
|  | Workspace, a Smalltalk scratch pad |
|  | Canvas, for creating graphical user interfaces for applications |
|  | Resource Finder, for locating and running applications |
|  | Data Modeler (see the <i>VisualWorks' Database Tools Tutorial and Cookbook</i>) |
|  | Online Documentation Browser, for reading online task reference and quick start guides |

System Transcript

The System Transcript is the area below the tool bar in the VisualWorks main window. The System Transcript displays system messages such as reporting when the current image was created. In Chapter 5, you will use the System Transcript to display output from testing code. You can also use the System Transcript as you would a Workspace (see the next section).

Workspace Window

In addition to the VisualWorks main window, the standard image displays a Workspace. Workspaces are windows that you can use as scratch pads. In addition to entering text (such as notes to yourself), you can enter and evaluate fragments of Smalltalk code. This makes Workspaces especially useful for:

- n Prototyping new code before making it a permanent part of the system
- n Testing code that has no specific graphical user interface

When you first start the standard image, the Workspace that is opened displays copyright information. You can have more than one Workspace open at a time.

Interacting with VisualWorks

You interact with VisualWorks through its graphical user interface. That is, you use a pointing device such as a mouse to manipulate visual controls such as menus and buttons in the VisualWorks windows. If you are new to using applications with graphical user interfaces, consult your platform's documentation for general information about interacting with such applications.

Mouse-Button Functions

One feature that makes VisualWorks different from other applications is that you interact with it using three mouse-button functions, called <Select>, <Operate>, and <Window>. You can invoke these functions from a three-button, a two-button, or a one-button mouse. To find out how these buttons correspond to the buttons on your mouse, see page xii.

In general, you use:

- n The <Select> button for making selections in a window—choosing menu items, clicking action buttons, highlighting text, selecting the location of keyboard input, and so on.
- n The <Operate> button for displaying and making choices from a pop-up menu called the <Operate> menu. The contents of this menu depend on the location of the mouse pointer.
- n The <Window> button for displaying and making choices from a pop-up menu called the <Window> menu. The commands on this menu perform window-management operations such as closing or resizing a window.

Getting Some Practice

Now try some basic operations to get acquainted with the VisualWorks interface and its logical mouse buttons:

1. Move the mouse until the pointer is over some text in the Workspace window. (The Workspace should still contain the copyright statement.)
2. Click the <Select> button. This moves the text cursor to the position indicated by the pointer. The text cursor (equivalent to an insertion point

in other applications) is a small, solid triangle between two characters at the base of the line of text.

3. Type some characters. They are inserted to the left of the text cursor.

Note: You must keep the mouse pointer within the window that is to receive keyboard input.

4. Undo your typing by choosing the **undo** command from the Workspace's <Operate> menu:
 - n With the pointer still in the Workspace, press and hold the <Operate> button. Move the pointer over the **undo** command and release the button.

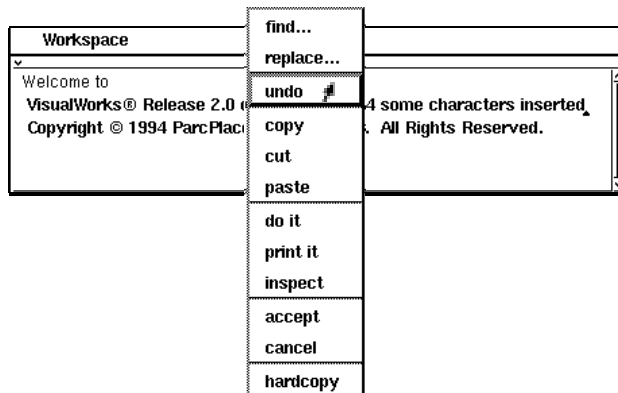


Figure 1-3 The undo command on the <Operate> menu

5. Open an additional Workspace by choosing **Tools?Workspace** from the VisualWorks main window:
 - n Put the pointer on **Tools** in the menu bar. Press and hold the <Select> button. Move the pointer over the **Workspace** menu item and release the button.

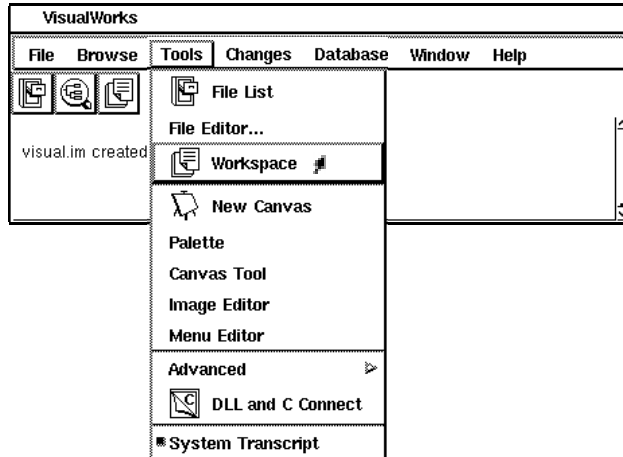


Figure 1-4 The Workspace command on the Tools menu

A window outline is displayed.



Figure 1-5 Window outline

6. Move the mouse pointer to position the window outline and click to display the Workspace.

Note: Unless otherwise specified, words like “click” and “double-click” always refer to the <Select> button.

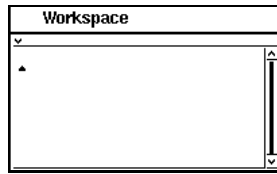
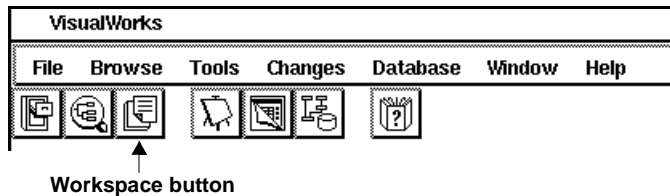


Figure 1-6 A new Workspace

7. Open a third Workspace, resizing it as you open it:
 - a. Click the Workspace button in the tool bar of the VisualWorks main window.



- b. Position the window outline as before.
 - c. Press and hold the <Select> button. The mouse pointer appears at the lower-right corner of the outline.
 - d. Move the mouse pointer to resize the outline as desired.
 - e. Release the <Select> button.
8. Select some text in the first Workspace, which should still contain the copyright statement:
 - n To select an arbitrary amount of text, hold the <Select> button down and drag the mouse pointer over the desired text; then release the pointer.
 - n To deselect text, click anywhere in the window.
 - n To select all text, double-click at the beginning of the first line or at the end of the last line.
 - n To select a single line of text, double-click at the beginning or end of the line.
 - n To select a single word, double-click within the word.
 - n To extend a selection, <Shift>-click where you want the selection to begin or end.

9. With some text selected, choose **copy** from the <Operate> menu. This copies the text to your platform's clipboard.
10. Move the pointer to another Workspace and choose **paste** from the <Operate> menu. This inserts the contents of the clipboard into the Workspace.
11. With the pasted text still selected, choose **cut** from the <Operate> menu. This deletes the text from the Workspace and puts it in the clipboard so you can paste it again.

Note: On many platforms, you can delete unselected characters using a <Delete> or <Backspace> key; however, such characters are not put into the clipboard.

12. In a Workspace that contains text:
 - a. Choose **accept** from the <Operate> menu. This causes VisualWorks to remember the current contents of the window.
 - b. Make some changes (add, delete, or copy any text).
 - c. Choose **cancel** from the <Operate> menu. This causes the window to revert to its accepted state.
13. Practice finding and replacing text (use **find** and **replace** on the <Operate> menu).
14. If your platform is set up with a default printer, print the contents of a Workspace by choosing **hardcopy** from the <Operate> menu.

Managing VisualWorks Windows

You can manage VisualWorks windows using your platform's window manager. Alternatively, you can use equivalent VisualWorks operations.

Try some window-management operations on VisualWorks windows:

1. Close one of the Workspaces by choosing the **close** command from the <Window> menu:
 - n With the pointer in the Workspace, press the <Window> button. Move the pointer over the **close** command and release the button.

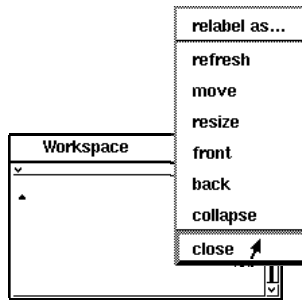


Figure 1-7 The **close** command on the <Window> menu

If a notifier is displayed, click **yes** to discard the text in the Workspace and close its window. Note that no notifier is displayed if you had accepted the text in this Workspace in the previous section.

2. Resize a Workspace window:
 - a. Choose **resize** from the <Window> menu. A window outline is displayed.
 - b. Move the pointer to resize the outline as desired and then click. The window is displayed in the new size.
3. Collapse a VisualWorks window by choosing **collapse** from the <Window> menu. Collapsing a window is a way of reducing clutter on your screen while keeping the window available when you need it.
 - n On Windows, UNIX, and OS/2 platforms, collapsing a window is equivalent to minimizing or iconifying it.
 - n On a Macintosh platform, collapsing a window reduces it to its title bar.
4. Restore the collapsed window to its original size using your platform's window-management operations:
 - n On Windows, UNIX, and OS/2 platforms, you can usually click or double-click on the collapsed window.
 - n On a Macintosh platform, click the collapsed window's zoom box.

Note to Macintosh users: Many Macintosh applications have a grow box in the lower-right corner for resizing windows. In VisualWorks windows, this grow box is invisible. To use it, you position the mouse pointer in the window's lower-right corner, hold down the <Option> key, press and hold the mouse button, and move the pointer to stretch the window size.

Saving Your VisualWorks Image

Whenever you interact with VisualWorks, you affect the state of the image you are running. Consequently, you save an image whenever you want to preserve a snapshot of its state.

Creating Your Own Working Image

You normally save your work in a user- or project-specific *working image* rather than the standard image. This preserves the standard image so that you can use it for reference or to create new images for new application developers or new projects.

Because this tutorial will guide you through the steps of creating an application, it is recommended that you create a working image for that purpose. To create the new image, you save the currently running image under a new name:

1. Choose **File?Save As...** in the VisualWorks main window. A dialog box prompts you with the name of the current image.
2. In the dialog box, edit (or replace) the current name to specify the filename for the new image. You can:
 - n Replace the name **visual** with a name such as **tutorial**. This creates the new image in the current working directory; on many platforms, this is the location that contains the standard image.
 - n Enter a fully qualified pathname to specify the new image's name and location in the file system. For example:
C: \MYWORK\TUTOR on a Windows or OS/2 platform
/usr/sue/mywork/tutorial on a UNIX platform
HD:MyWork:tutorial on a Macintosh

Note: Do not include the **.im** file extension in the filename. VisualWorks will add that for you. Furthermore, you must specify a location that already exists; VisualWorks will not create a directory for you.

Note to Macintosh users: You construct a pathname using a colon (:) to separate the volume name, one or more folder names, and the document name.

3. Click **OK**. As a result:
 - n A new file (for example, **tutorial.im**) is created on your disk. A message in the System Transcript reports this.

- n The VisualWorks windows on the screen now belong to the new image.
4. If you created the new image in the same location as standard image, skip to “Taking Snapshots,” below. Otherwise, continue with the following section.

If You Created an Image in a New Location

Every image consults several additional files for adjunct information:

- n A *sources file* provides the source text of the image’s compiled Smalltalk objects.
- n A *help file* provides the text of the online documentation.

If you created your new working image in a location different from that of the standard image, the new image may not be able to find these files. To ensure that these files can be found:

1. Choose **File?Settings** in the VisualWorks main window. This displays the Settings Tool, which you will learn more about on page 35.
2. Edit the **Sources:** field to specify the fully qualified name of the installed sources file, typically:
 - n `/usr/visual/image/visual.sou` on a UNIX platform
 - n `C:\VISUAL\IMAGE\VISUAL.SOU` on a Windows or OS/2 platform
 - n `HD:visual:image:visual.sou` on a Macintosh computer
3. Click the **Accept** button.
4. Click the **Help** tab (*not* the **Help** button). It is located to the right, between tabs labeled **Window** and **Icon**.
5. In the **Documentation Directory:** field, specify the fully qualified name of the installed online documentation directory, typically:
 - n `/usr/visual/online` on a UNIX platform
 - n `C:\VISUAL\ONLINE` on a Windows or OS/2 platform
 - n `HD:visual:online` on a Macintosh
6. Click the **Accept** button and close the Settings Tool.

Taking Snapshots

Now that you are running your own working image, you can save all subsequent work without affecting the standard image. As with most file-based applications, it is a good idea to save your image (or “take a snapshot”) periodically, especially after:

- n Changing environment settings (as described above)
- n Arranging VisualWorks windows in a useful way
- n Adding new Smalltalk objects to the system

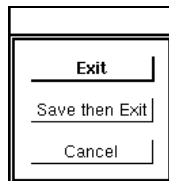
Save your image now to preserve the current display (and, if applicable, the changed sources file setting). To do this:

1. Choose **File?Save As...** in the VisualWorks main window. A dialog box prompts you with the name of the current image.
2. Click **OK**. The System Transcript reports the save.

Exiting VisualWorks

To exit VisualWorks:

1. Choose **File?Exit VisualWorks...** from the VisualWorks main window. The following dialog box appears:



In this dialog box:

- n **Exit** terminates VisualWorks without saving the image.
 - n **Save then Exit** saves the image and then exits VisualWorks.
 - n **Cancel** leaves VisualWorks running.
2. Because you have already saved your image, click **Exit**.

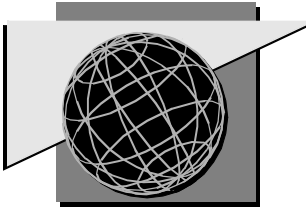
If you cannot exit using the **File?Exit VisualWorks...** command, see “Emergency Exit” in Chapter 17 of the *VisualWorks User’s Guide*.

Note for Windows users: You must exit VisualWorks before you can shut down Windows.

Note for Macintosh users: When exiting VisualWorks normally, do not use **File?Quit** from the Macintosh menu bar. This command should be used only for an emergency exit.

What’s Next: The VisualWorks Environment

So far, you’ve learned about what VisualWorks is, how to start it, how to exit it, and how to perform some of the most basic user operations. In Chapter 2, you will take a brief look at the basic tools in the VisualWorks environment.



Chapter 2

The VisualWorks Environment

This chapter introduces you to the basic tools for exploring and configuring the VisualWorks environment. Some of these tools help you find and manipulate Smalltalk objects; others provide information about your VisualWorks image. In the following sections, you will restart your working image and:

- n Send messages to Smalltalk objects
- n Browse the Smalltalk class library
- n Store and retrieve information in files
- n Run an application
- n Browse online documentation
- n Customize your working image
- n View a list of changes made to the image

Starting Your Working Image

If you exited VisualWorks at the end of the last chapter, restart it now. To do this:

- %o Follow the directions on page 3, using the icon or filename for your working image instead of specifying the standard image. (Your working image is the image you created on page 13.)

When VisualWorks starts, its windows are arranged as they were when you last saved the image.

Sending Messages to Smalltalk Objects

In Chapter 1, you used a Workspace for simple text editing. In this section, you will use a Workspace to send *messages* to Smalltalk objects. Recall from page 2 that an object consists of some data plus a set of operations that manipulate the data. A message is a request for an object to carry out one of its operations.

For example, you can cause the System Transcript to display some text by sending a message to it:

1. Open a Workspace, if necessary.
2. In the Workspace, type the following lines:

```
Transcript cr.  
Transcript show: 'Hello, world!'
```

Be sure to include the period, the colon, and single quotation marks.

3. Select (highlight) these lines.
4. Choose the **do it** command from the Workspace's <Operate> menu.

The text **Hello, world!** appears on its own line in the System Transcript of the VisualWorks main window.

You just entered and evaluated two Smalltalk *message expressions*. Each message expression describes a message (**cr** and **show:**) to a *receiver* (**Transcript**). When you choose **do it**, each message expression is evaluated, causing the receiver to carry out the operation requested by the message. Chapter 5 will give you more experience in creating and evaluating message expressions.

Because VisualWorks is a Smalltalk system, all VisualWorks operations are accomplished by objects sending messages to other objects. That is, whenever you choose a menu item or click a button in a VisualWorks tool, you start a chain of message-sends that perform the tool's action.

Browsing the Smalltalk Class Library

Every object in the Smalltalk system is an *instance* of a *class*. A class is a template for defining the data and operations for a particular type of object. That is, a class defines:

- n The *instance variables* in which instances store data
- n The *instance methods* (procedures) that describe how instances carry out operations

All instances of a given class have the same form and behavior, but they contain different data in their variables.

VisualWorks comes with a large *library* of predefined classes from which you can create objects as part of the applications you build. You can also add your own classes to this library to create more specialized objects.

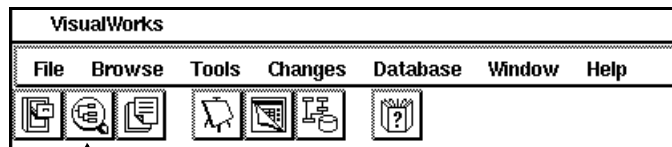
Using a System Browser

You can use a System Browser to explore the classes in the VisualWorks class library. (In Chapter 5, you will use the System Browser to create classes as well.) You can open multiple System Browsers to see different parts of the class library at a time. For example, opening two System Browsers is useful for comparing two classes or methods side by side.

Follow these steps to explore a portion of the VisualWorks class library:

1. Open a System Browser by choosing **Browse?All Classes** in the VisualWorks main window.

Shortcut: Click the System Browser button in the tool bar.



System Browser button

The System Browser you opened looks something like this:

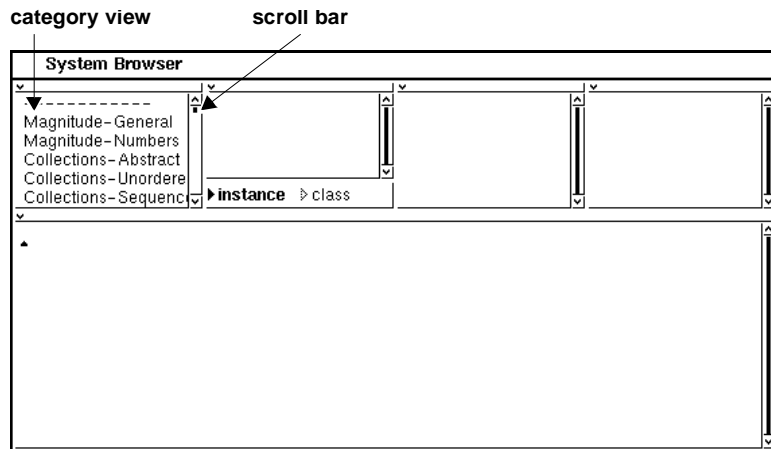


Figure 2-1 A newly opened System Browser

A System Browser, like many VisualWorks tools, has regions called *views* for displaying different kinds of information. The upper-left view lists the *categories* in the system. Categories are the way Smalltalk organizes groups of similar classes. Every class in the system belongs to exactly one category.

2. Scroll through the list of categories in the category view:
 - n Put the pointer on the scroll bar to the right of the list, press and hold the <Select> button, and drag the scroll bar down to show the remainder of the list.
3. Select the first category in the list (**Magnitude-General**) to see which classes belong to it. (Scroll back to it if necessary.) Note that:
 - n These classes are listed in the *class view* next to the category view.
 - n A template for creating a new class is displayed in the window's *code view*.

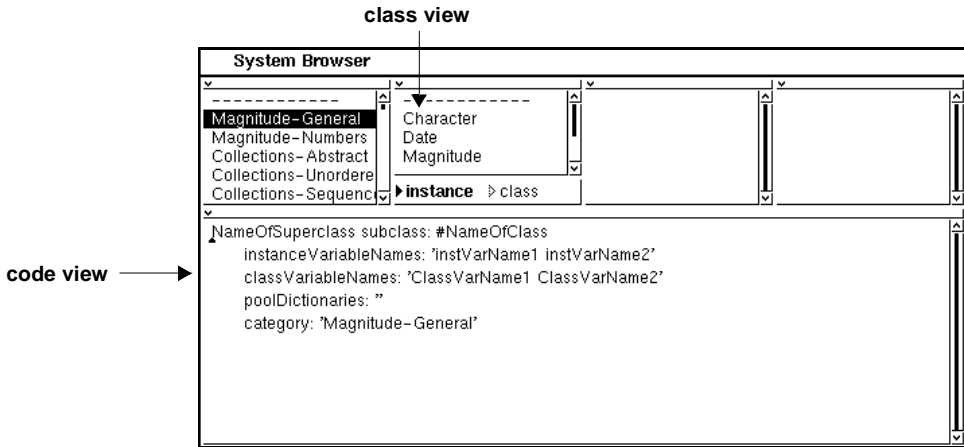


Figure 2-2 A System Browser with a category selected

4. In the class view, select the `Date` class to see its instance variables and methods. Instances of `Date` represent individual days of a year:
 - n The instance variables (`day`, `year`) appear as part of the *class definition* in the code view.
 - n The instance methods are grouped in *protocols*, which are listed in the protocol view. Protocols are categories for organizing methods.

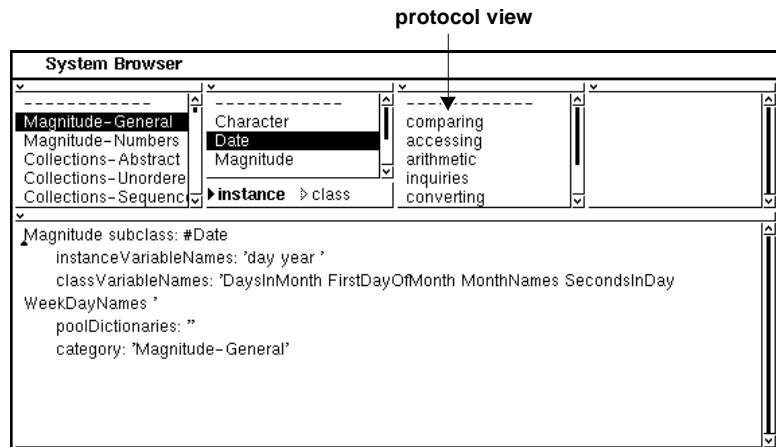


Figure 2-3 A System Browser with a class selected

5. In the protocol view, select the protocol **accessing** to see the instance methods it contains:
 - n These methods are listed in the *method view*.
 - n A template for creating a new method is displayed in the code view.

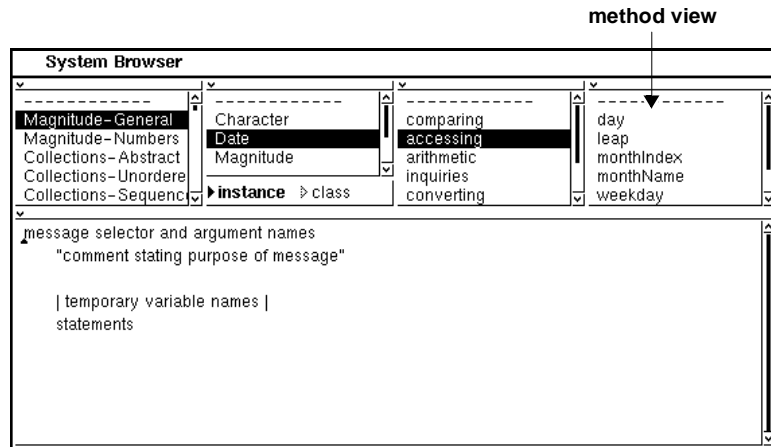


Figure 2-4 A System Browser with a protocol selected

6. In the method view, select the instance method **weekday**. Its definition appears in the code view. The method's comment indicates that this operation calculates the day of the week on which a given date falls.

Note: If a notifier appears saying your sources file is invalid, you probably need to perform the steps in the section "If You Created an Image in a New Location," on page 14.

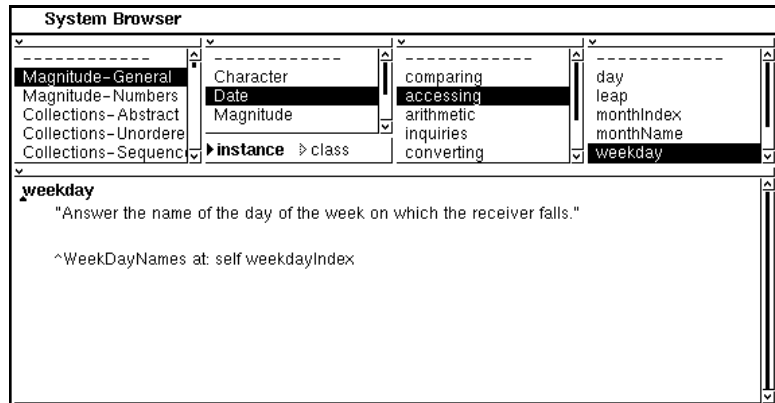


Figure 2-5 A System Browser with a method selected

To see the `weekday` method in action, you can send an appropriate message to an instance of `Date`. For example:

1. In a Workspace, type the following message expression:

```
Date today weekday
```

2. Select this expression and choose **print it** from the Workspace's <Operate> menu. This evaluates the message expression and displays its result in the Workspace. The expression reports the name of the current day of the week.

Finding a Class by Name

Browsing categories is a good way to explore the class library. However, if you already know the name of the class you want to view, you can find it directly. For example, assume you want to find the class called `Point`, whose instances represent locations on the screen defined by `x` and `y` coordinates. To do this:

1. Put the pointer in the category view and choose **find class...** from the <Operate> menu.
2. Type `Point` in the input field of the dialog box and click **OK**. The System Browser displays the definition of `Point`, which belongs to the `Graphics-Geometry` category.

Note: You can use the asterisk (*) as a wildcard character in the string you type in the dialog box.

Adding a Category

You can add categories to the class library for organizing your own code. For example, later in this tutorial, you will build a sample application. You can prepare for this by creating a category to put it in. To do this, you:

1. Click the **Graphics-Geometry** category to deselect it, if necessary.
2. Choose **add...** from the category view's <Operate> menu.
3. Type **Examples-VWTutorial** in the input field of the dialog box and click **OK**. The category view automatically scrolls to the end, showing the added category.

By default, new categories are created at the end of the list. You can create a new category in a particular location by selecting an existing category first; the new category is inserted above the selection.

Browsing the Class Hierarchy

Every class (except one) in the Smalltalk class library is a *subclass* of some other class (its *superclass*). A subclass is a *specialization* of its superclass—its instances have the same kind of data and behavior as instances of the superclass, plus some of their own. That is, as a subclass, every class has more variables and methods than appear in its definition—it also has the variables and methods it *inherits* from its superclass. Inheritance captures the similarities among related kinds of objects, while allowing for their differences.

The inheritance relationships among classes form a hierarchy rooted in the class **Object**. That is, all classes are directly or indirectly subclasses of the class **Object**, which defines the state and behavior common to all objects in the system. **Object** does not inherit from any other class.

Using a Hierarchy Browser

You can use a Hierarchy Browser to browse the branch of the inheritance hierarchy to which a given class belongs. This is useful for getting a complete picture of a class—what is defined in it and what it inherits.

For example, assume that you want to know how Smalltalk implements numbers. You can explore the inheritance hierarchy that contains the class **Number**:

1. Choose **Browse?Class Named...** in the VisualWorks main window.

2. Type **Number** in the input field of the dialog box. This displays a Hierarchy Browser on the **Number** class.
3. Resize the Hierarchy Browser window as shown in Figure 2-6 so that you can read the contents of the class view easily. Subclasses are indented under superclasses in this view.

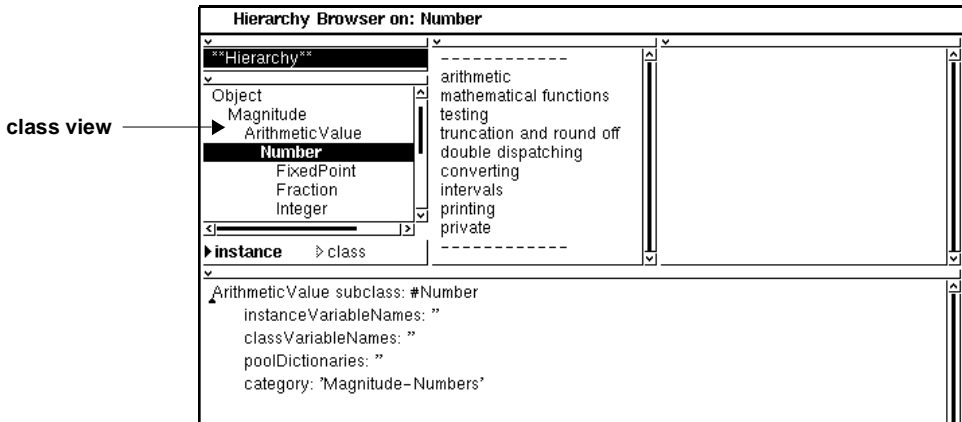


Figure 2-6 A Hierarchy Browser on the Number class

4. Scroll up the class view to see all of **Number**'s superclasses, their superclasses, and so on up to **Object**. You can select any of these classes to see variables and methods that are inherited by **Number**.
5. Scroll down the class view to see all of **Number**'s subclasses, their subclasses, and so on. You can select any of these classes to see the various kinds of numbers in the system.
6. Close the Hierarchy Browser.

If you are looking at a class in a System Browser and you want to browse its inheritance hierarchy, you can select the class and choose **spawn hierarchy** from the class view's <Operate> menu. This opens a separate window containing a Hierarchy Browser.

Storing and Retrieving Information in Files

A VisualWorks image stores an entire Smalltalk system in a single disk file. Sometimes it is useful to save subsets of the system in separate disk files. For example, as you develop new classes for an application, you can write the category that contains them to a disk file. This is called *filing out* the category. The resulting disk file can serve as a backup, an archive, or a way of preserving intermediate stages of your work.

Just as you can file out code from an image to a disk file, you can also *file in* code—that is, you can retrieve filed-out code by reading it back into your image. This serves as a means of sharing work across images. For example, if another user creates an application that you can use, that user can file it out of his or her image so that you can file it into yours.

Writing to Disk Files

You can file out categories, classes, and individual methods from any browser. For example, you can:

1. Select a class in the System Browser or the Hierarchy Browser.
2. Choose **file out as...** from the class view's <Operate> menu. A dialog box displays the default name for the new file (the class's name with the **.st** file extension).
3. If desired, enter a different name; then click **OK**.
4. Use your platform's file-management facilities to verify that the file was created in the current working directory.

Retrieving Information from Disk Files

You use a File List to locate and select files in your file system and then read them into your image. For example, you can file in the provided sample calculator application (assuming that you installed all of the VisualWorks files and directories):

1. Choose **Tools?File List** in the VisualWorks main window.
Shortcut: Click the File List button in the tool bar.

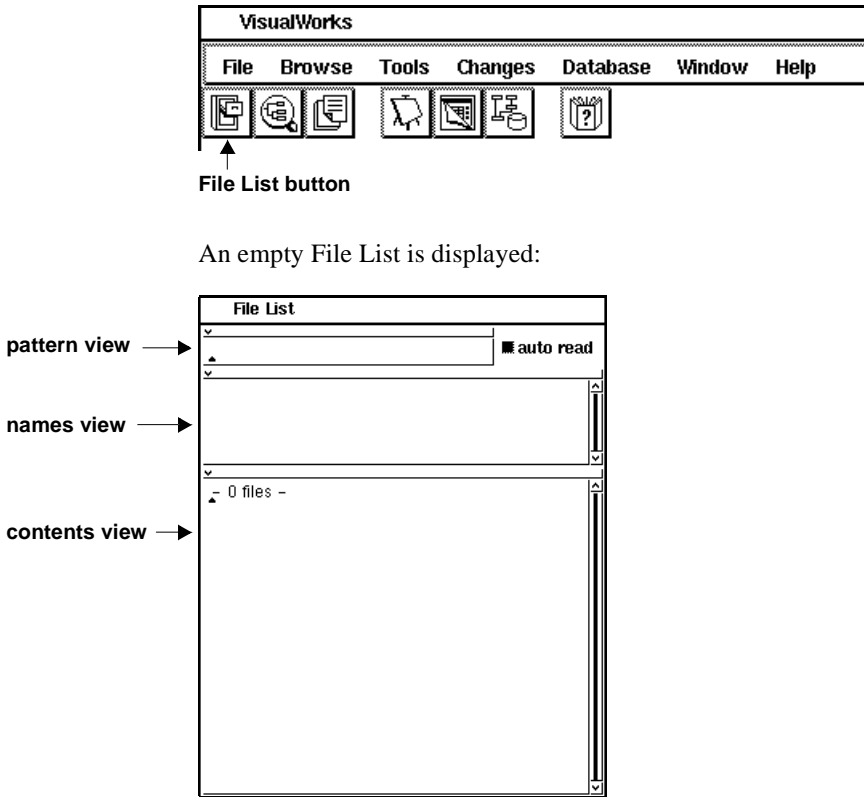


Figure 2-7 An empty File List

2. In the *pattern view* at the top of the File List, type a pathname pattern that matches the contents of the **tutorial** subdirectory of your VisualWorks installation directory, typically:
 - n `/usr/visual/tutorial/*` on a UNIX platform
 - n `C:\VISUAL\TUTORIAL*` on a Windows or OS/2 platform
 - n `hd:visual:tutorial:*` on a Macintosh
 In the pattern, the asterisk (*) is a wildcard character.
3. Press <Return>. The *names view* lists the files and directories that match the name in the pattern view. In this case, the names view displays two directory names.
4. In the names view, select the pathname for the **basic** directory. The *contents view* displays the selected directory's contents (two filenames).

5. With the pointer in the names view, choose **new pattern** from the <Operate> menu. This changes the pattern view so it specifies the contents of the **basic** directory and adjusts the names view accordingly.
6. In the names view, select the pathname for **calc.st**. The contents view displays the source code for the sample application stored in the file.
7. With the pointer in the names view, choose **file in** from the <Operate> menu. This reads in the source code from the selected file and compiles it into the image. Notice the progress messages displayed in the System Transcript.
8. Close the File List.
9. Verify that the file-in was successful:
 - a. In a System Browser, scroll to the bottom of the category view to locate the category **UIExamples-General**. If necessary, refresh the view by choosing **update** from the category view's <Operate> menu.
 - b. Select the **UIExamples-General** category. Notice that it contains two classes, **Calculator** and **CalculatorExample**.

File Lists can also be used as general-purpose browsers for your file system. Through a File List, you can list the contents of any directory or file, edit a file, and create new files.

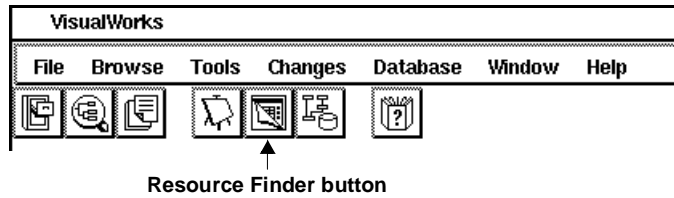
Running an Application

Applications built with VisualWorks normally include at least one class that contains *resources*. Resources are specific kinds of information that are required for assembling an application's graphical user interface. Resources include:

- n Specifications for constructing windows
- n Specifications for constructing menus
- n Graphical images (such as icons) for use in windows
- n Queries for retrieving data for display from a relational database

You use a Resource Finder to locate the resources in the system and the classes that define them. Consequently, you can use the Resource Finder as a convenient way to locate and start applications. For example, to locate and start the calculator application you just filed in, you can:

1. Choose **Browse?Resources** from the VisualWorks main window.
Shortcut: Click the Resource Finder button in the tool bar.



This brings up the Resource Finder, whose class view lists all the classes that contain resources:

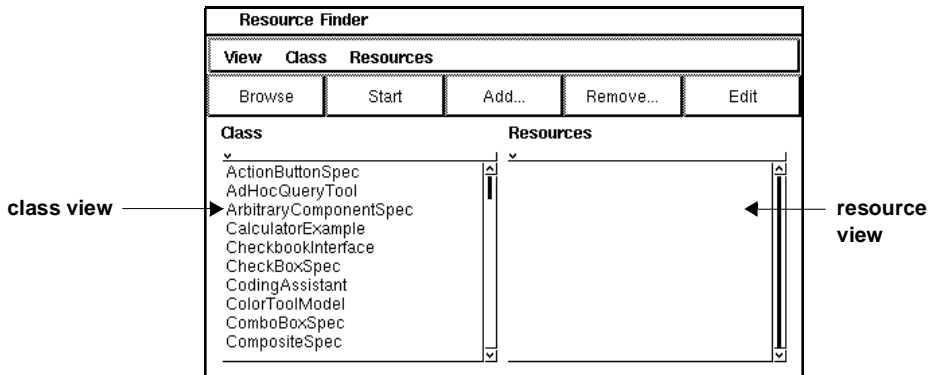


Figure 2-8 A newly opened Resource Finder

2. Locate the `CalculatorExample` class in the class view.

Hint: Because you created this class when you filed it in, you can choose **View?User Classes** to filter out the system classes.

3. Select the class `CalculatorExample`. This lists its resource (`windowSpec`) in the resource view:

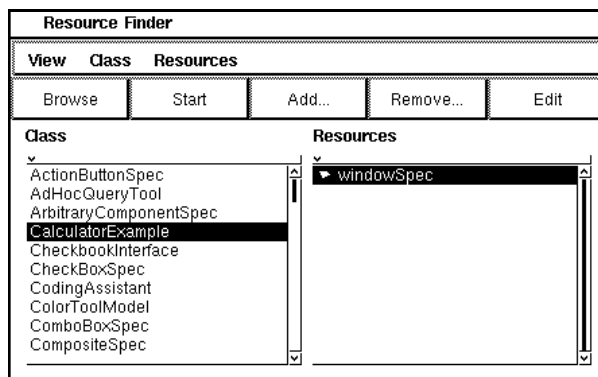


Figure 2-9 A Resource Finder with a class selected

4. Click the Resource Finder's **Start** button. This starts the application. Try out the calculator; when you are finished, close its window.
5. Exit the Resource Finder by closing its window or by selecting **Exit** from its **View** menu.

Browsing Online Documentation

The VisualWorks online document library contains:

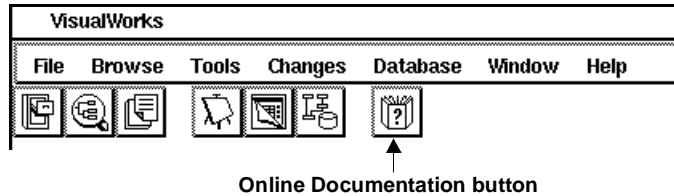
- n The *VisualWorks Cookbook*: A collection of “how-to” topics that explain Smalltalk basics and provide steps for common application-building tasks
- n The *Database Cookbook*: A collection of “how-to” topics that pertain specifically to using VisualWorks’ database tools
- n The *Database Quick Start Guides*: An overview of steps for building database applications with VisualWorks
- n The *International User’s Guide*, which describes the VisualWorks facilities for creating nonEnglish and cross-cultural applications

The online cookbooks also exist as printed books in the VisualWorks documentation set.

You access the online document library through the Online Documentation Browser. For example, to find online information about Smalltalk message expressions, you:

1. Choose **Help?Open Online Documentation** from the VisualWorks main window.

Shortcut: Click the Online Documentation button in the tool bar.



This brings up the Online Documentation Browser, which displays a list of online books:

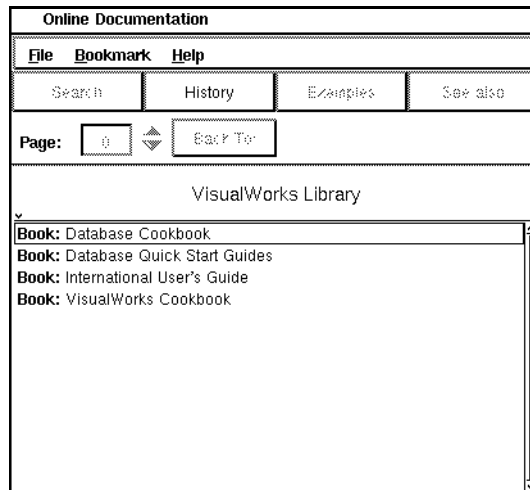


Figure 2-10 A newly opened Online Documentation Browser

2. In the Online Documentation Browser, select **Book: VisualWorks Cookbook**. This lists the Cookbook's chapters.

Note: If a notifier informs you that the online book's file is missing, you probably need to perform the steps in the section "If You Created an Image in a New Location," on page 14.

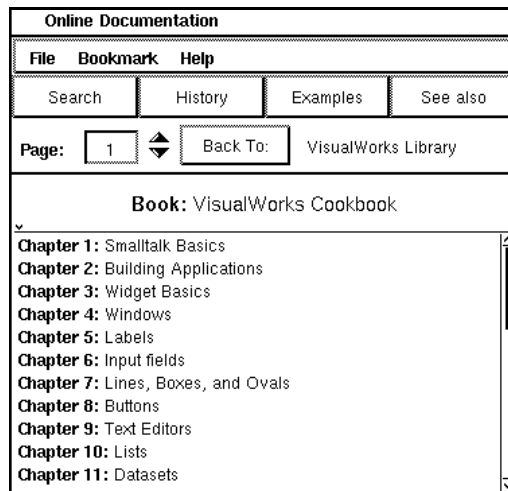


Figure 2-11 Browsing the VisualWorks Cookbook chapters

3. Select **Chapter 1: Smalltalk Basics**. This lists the chapter's topics.
4. Select the topic **Constructing a message**. Cookbook topics normally contain the following sections:
 - n **STRATEGY** (concepts for understanding a task and choosing among alternative tasks)
 - n **BASIC STEPS** (steps for performing that task)
 - n **VARIANTS** (steps for performing similar tasks)
5. Read the first two sections of **Constructing a message**. The basic steps give directions for constructing a sample message expression.
6. Click the **Examples** button in the Online Documentation Browser. This brings up an Examples window that displays the sample code to which the steps refer:

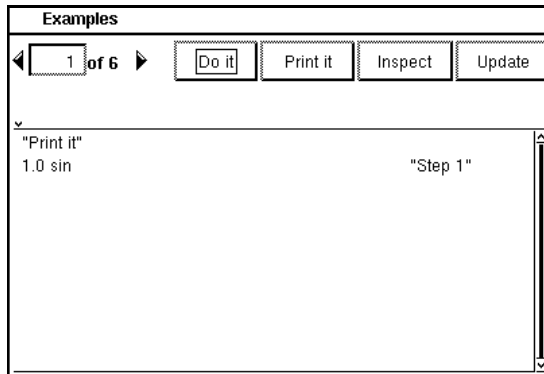


Figure 2-12 An Examples window for online documentation

7. Notice that the comment in the Examples window says "Print it". This means you can click the **Print it** button to evaluate the expression and display the result. (Other examples may tell you to "Do it" or "Inspect it".)
8. Display the next example for this topic by clicking the right arrow at the top of the Examples window. (This example is described in the **VARIANTS** section of the topic.) You can scroll back and forth through a topic's examples using the arrows.
9. Close the Examples window (but leave the Online Documentation Browser open). In the resulting notifier, click **yes** to discard the results of step 7.

Exploring the Cookbook's Sample Applications

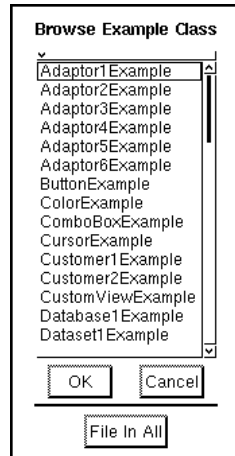
In many topics, the code fragments shown in the Examples window are part of an entire sample application. As you learn more about building applications, you will want to see the example code in context. The following steps show you how to find sample applications that are used in the Cookbook:

1. In the Online Documentation Browser, click the **Back To:** button several times to return to the list of VisualWorks Cookbook chapters.
2. Select **Chapter 17: Notebooks** and then select the topic **Adding a notebook**.
3. Scroll to the **BASIC STEPS** section and notice the line:

Online example: **Notebook1Example**

This indicates that the examples in the basic steps are part of a sample application called **Notebook1Example**.

4. Choose **File?Browse Example Class** in the Online Documentation Browser. This displays a list of the sample applications that support the online documentation:



5. Locate and select **Notebook1Example** in this list; then click **OK**.
6. In the resulting notifier, click the **File It In** button to request that **Notebook1Example** be filed into your image. Notice the progress messages that appear in the System Transcript. When filing in is complete, a window outline appears.
7. Position the window outline and click to display the Hierarchy Browser on the **Notebook1Example** class. This is where you can examine the example code in context.
8. Run the sample application by opening a Resource Finder (see page 28), selecting **Notebook1Example**, and clicking **Start**. This application illustrates the use of a notebook widget to list all the Smalltalk classes in alphabetical order.
9. Close the **Notebook1Example** window, the Resource Finder, the Hierarchy Browser, and the Online Documentation Browser. (Leave the VisualWorks main window open.)

Customizing Your Working Image

You can customize a number of aspects of your image through the Settings Tool. In general, you can:

- n Control the default size, look, and behavior for a number of VisualWorks tools.
- n Specify where VisualWorks can find various files and directories. (You may have already done this on page 14.)

To display the Settings Tool:

1. Choose **File?Settings** from the VisualWorks main window. As shown in Figure 2-13, the Settings Tool is arranged as a *notebook*, with one page per customizable feature. Each page is indicated by a labeled *tab*.

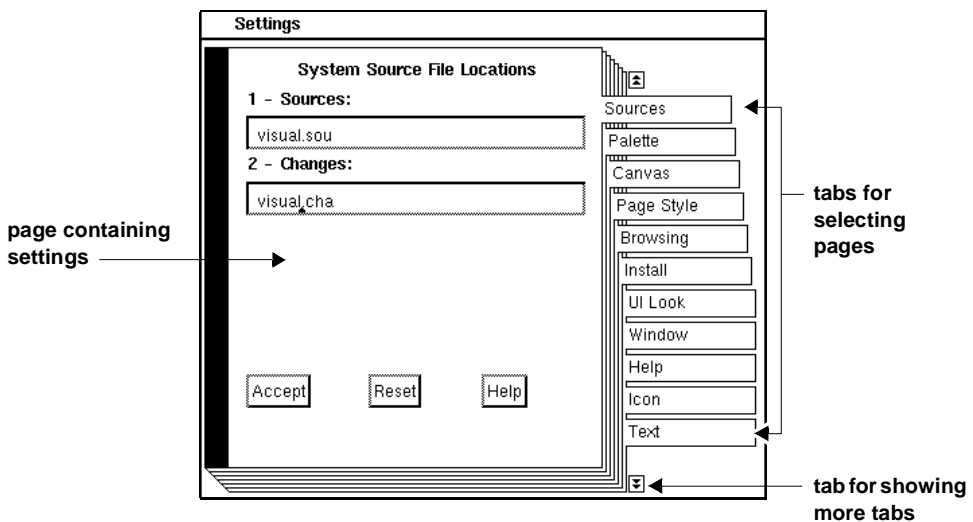


Figure 2-13 Settings Tool

2. Click on the tab labeled **UI Look**. The settings on this page control the look of VisualWorks windows.
3. Click the **Help** button on the **UI Look** page and read the description. Notice that the default look selection is **Auto Select**, which means that VisualWorks selects a look that is compatible with your platform.
4. Choose a different look selection (but leave **Basic Tools Adopt Look** selected):

- n With the pointer on the **Look Selection** menu button, press and hold the <Select> button. Move the cursor to the desired menu item and then release the button.
5. Click **Accept**. Notice the effect on any open windows such as the VisualWorks main window.
6. Change the look back to **Auto Select** and click **Accept**.
7. Close the Settings Tool.

Viewing Changes Since the Last Save

VisualWorks records the changes that you make to the classes and methods in the Smalltalk system in your image. These changes are listed in a *changes file*, which is located in the same directory as the image file. The changes file has the same name as the image file, except that its file extension is **.cha**.

The changes file records:

- n Changes to class and method definitions that result from editing or filing in code
- n Actions that create objects and send messages to them

VisualWorks records these changes so that they can be replayed (reloaded) in your image. This is useful for recovery after power outages or system failures, because it allows you to reconstruct any unsaved changes in your image. The more frequently you save your image, the fewer changes you need to replay when recovering it.

To see the changes that you made since the last time you saved your image:

1. Choose **Changes?Open Change List** from the VisualWorks main window. This opens an empty Change List.
2. Put the pointer in the *changes view* in the upper-left corner of the Change List and choose **file in/out?recover last changes** from the <Operate> menu.

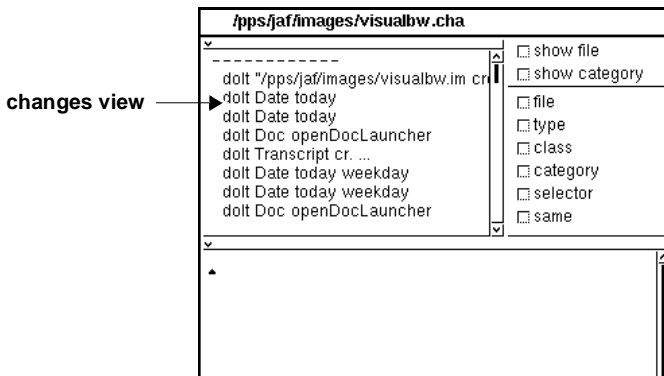


Figure 2-14 Change List

Your changes are listed in the change view. You typically need to filter this list before replaying any changes. For information about recovering an image, see Chapter 17, “Troubleshooting,” in the *VisualWorks User’s Guide*.

3. Close the Change List.

What’s Next: Creating Applications

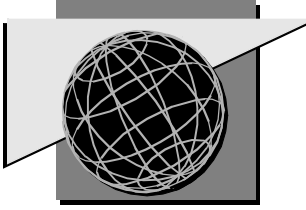
So far, you’ve been introduced to:

- n Basic Smalltalk concepts (objects, messages, classes, instance variables, and methods)
- n Basic VisualWorks tools (Workspace, System Browser, File List, Resource Finder, Online Documentation Browser, Settings Tool, Change List)

In the following four chapters, you will build on what you’ve learned to create a sample application. Chapter 3 introduces this application, and Chapters 4 through 6 show you how to build it.

Note: *If you are specifically interested in creating database applications, you should read at least Chapters 3 and 4 before consulting the VisualWorks’ Database Tools Tutorial and Cookbook.*

Before you go on, you should save your image to keep the category you added and the code you filed in.



Chapter 3

Introduction to VisualWorks Application Building

This chapter:

- n Introduces the requirements for the sample application that you will develop over the next few chapters
- n Describes the general characteristics of VisualWorks application design
- n Outlines the design you will use for the sample application

Application Requirements

Your task is to use VisualWorks to write a simple online checkbook that records basic checking-account transactions. This application must:

- n Enable users to make deposits into a checking account
- n Enable users to write checks against the account
- n Enable users to cancel written checks
- n Provide a list of the written checks
- n Provide the account balance

In addition to these functional requirements, there is a design requirement that the application have a *graphical user interface*—one or more windows and/or dialog boxes that give the user appropriate controls for viewing information, entering information, and invoking operations.

For the sake of simplicity, the Checkbook application does not handle persistent data—that is, it does not connect to a database. Instead, the checkbook (and the account it represents) is created when the application is started and destroyed when the application is closed.

VisualWorks Approach to Application Design

This tutorial presents steps for constructing the sample application with VisualWorks. The end product of this process is a running Smalltalk application—a set of interrelated objects that interact by sending messages to each other. Thus, the process of creating an application involves:

- n Deciding what objects are required by the application
- n Adapting or creating classes that define the data and behavior of these objects

Although it is possible to design and create an application from fundamental objects, you can accelerate the process by using the VisualWorks *application framework*. The classes in this framework provide a core structure that you augment to build complete applications:

- n For some parts of the application, you use point-and-click operations to specify the relevant portions of the framework.
- n For other parts of the application, you create subclasses from the framework classes and add code as appropriate.

The classes in the VisualWorks framework fit together in specific ways. These classes embody a particular approach to structuring an application, which you need to understand in order to use classes from the framework and integrate them with classes of your own.

The following sections describe two main characteristics of VisualWorks application structure—namely, that it is layered, and that one of these layers captures and expresses the organization of the user interface.

Layered Structure

A VisualWorks application is conceptually divided into two parts:

- n The *information model*, which handles data storage and processing
- n The *user interface*, which handles input and output

This separation of concerns results in corresponding layers of objects:

- n *Model objects* (or simply *models*), which define and manipulate data structures.
- n *User-interface (UI) objects*, which present data from the models and enable users to interact with this data. UI objects are the objects that make up a display screen; they include windows and *widgets* (controls such as input fields, action buttons, scrollable lists, and the like).

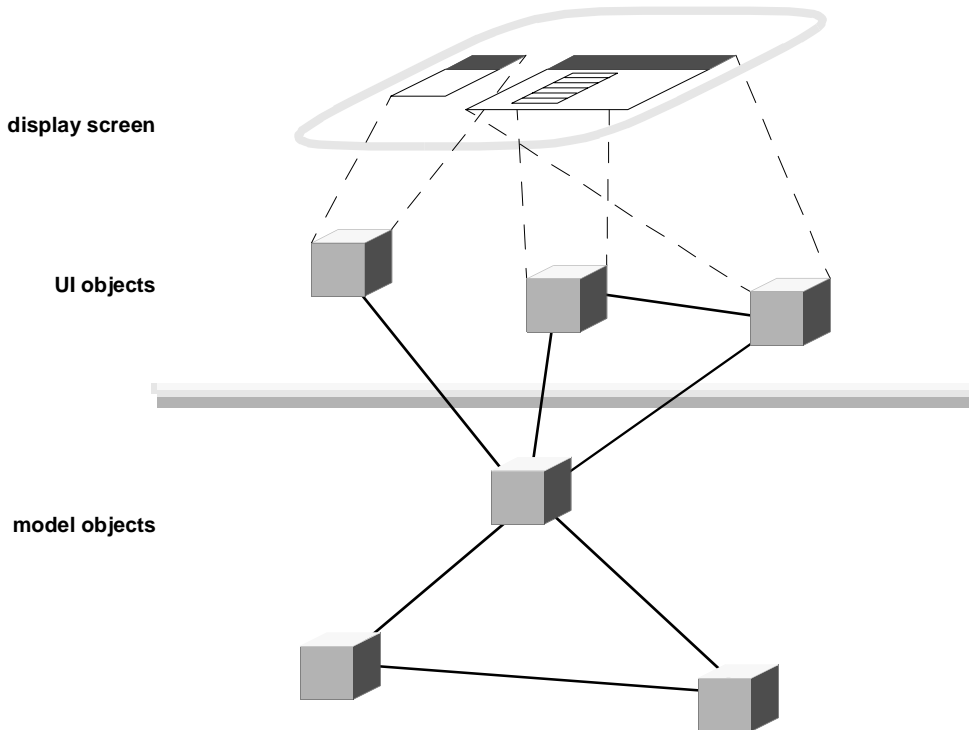


Figure 3-1 *Layers of the user interface and information model*

Though distinct, UI objects and models are highly interconnected. For example, every widget that displays information depends on some model for that information. By itself, each UI object simply provides visual characteristics such as shape and color, as well as any visual response to keyboard and mouse input (for example, movement or color change).

A typical application is further layered to distinguish different kinds of models. Among these are domain models and application models. (Other kinds of models—namely, value models—are described in Chapter 6.)

Domain Models

Domain models simulate the state and behavior of real-world objects in the application's *domain*, which is the area of endeavor that the application helps to automate (for example, accounting, inventory control, payroll, and the like).

Domain models define the data that is relevant to the domain and perform the operations that process the data. For example, an accounting application might include domain models such as customers, debtors, and creditors.

Application Models

Application models provide a layer of information and services between UI objects and domain models. Among other things, an application model defines the application-specific behavior of individual widgets in the interface—for example, by:

- n Establishing the connections between the widgets and the data they present
- n Controlling how the widgets interact with each other

An application model may also provide additional data definition and processing that are required by the application but are not part of the core domain. For example, the application model in an accounts-receivable application may provide transactions that are not part of any accounting domain models.

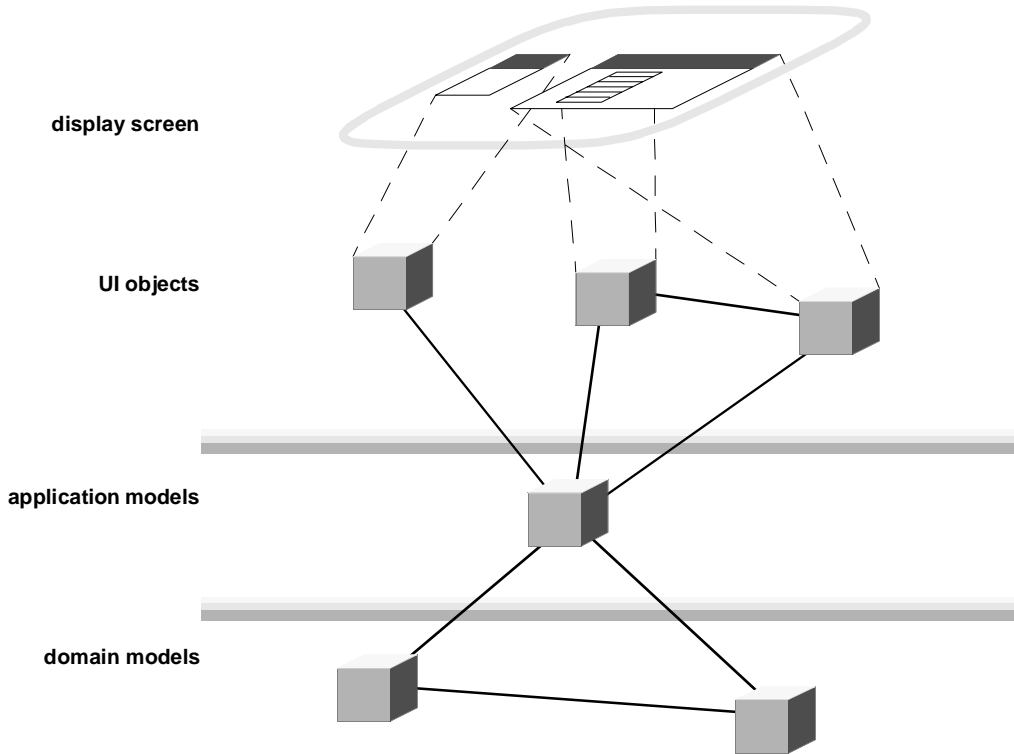


Figure 3-2 Layers within the information model

Why Layering?

The layering of models and UI objects is the foundation for developing multiple applications in the same domain. That is, families of related applications can reuse existing domain models and have different application models to support their different user interfaces.

Similarly, layering facilitates maintenance. In general, an application's domain models tend to be relatively stable, whereas its user interface may require considerable revision from one release to the next. This means editing or replacing application models, while making only minor changes to domain models.

UI-Based Structure

As a rule, each application model in a VisualWorks application supports one window in the interface. That is, each application model defines the application-specific behavior for a particular window and the widgets in it. Consequently, a multiwindow application typically has several interconnected application models.

More generally, the application-model layer mirrors the organization of the user interface. A user interface identifies chunks of related information and operations, and it presents these chunks in one or more interrelated windows, where:

- n Some windows are primary—persistent windows in which users perform the bulk of their work.
- n Other windows are secondary—dialog boxes and “satellite” windows that are opened only as adjuncts to some primary window.

Each application model corresponds to some portion of the window organization, supporting an individual window or a group of related windows (for example, a primary window and its dialogs). In fact, if a window contains relatively independent subregions, a separate application model may exist for any of those subregions.

Why UI-Based Structure?

Like layering, UI-based structuring promotes reuse. Each application model that supports a meaningful chunk of user interface can be combined with other application models to form new interfaces. In fact, you can think of each application model as defining an independently runnable “application unit” from which larger applications can be constructed.

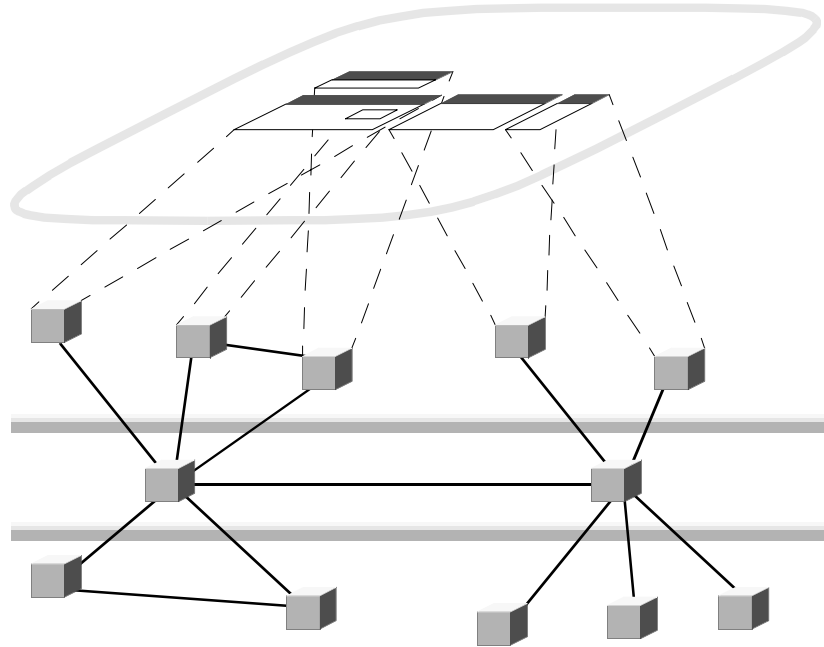


Figure 3-3 Composing larger applications from smaller ones

Building Blocks in the Framework

The VisualWorks framework provides classes for specific parts of the application structure described above:

- n Predefined classes for UI objects (windows and widgets)
- n Superclasses for application models

You create your application models as subclasses of the appropriate superclasses in the framework. Each such application model inherits the ability to create UI objects from framework classes. That is, you simply specify the layout and contents of the windows in the interface and then store these specifications in the appropriate application models. When the application runs, each application model builds its window(s) according to your specification(s), creating suitable window and widget objects from the classes in the framework.

Domain models are not part of the application framework, because their structure and contents vary widely from domain to domain. Instead, you build domain models from standard Smalltalk classes. In fact, separate domain models may be unnecessary in very small applications or where reuse is not a consideration; in these cases, all of the application's processing resides in application models.

Framework for Database Applications

VisualWorks applications that access relational databases have the same basic structure outlined here, except that they have specialized kinds of application models (for example, *data forms* and *database applications*). Furthermore, instead of “hand-built” domain models, the database application uses *entity classes* generated automatically from database tables. See the *VisualWorks' Database Tools Tutorial and Cookbook* for more information.

Designing the Sample Application

At this point, you are ready to design the sample application, considering its requirements in light of the VisualWorks design approach described above.

Designing an application involves making choices and trade-offs; there is no one right structure. The following sections guide you through the choices made for the Checkbook application. Note that some of these choices serve to simplify the application so that the tutorial can focus on essential aspects of the development process.

The Checkbook application is a very small application with straightforward requirements. When analyzing and designing more complex applications, you may want to use a more formal methodology. ParcPlace-Digital offers training and consulting for a methodology called Object Behavior Analysis and Design (OBA/D).

Designing the User Interface

When you design the Checkbook application's user interface, you decide:

- n How many windows?
- n What kind of windows?
- n What information and operations belong in each window?

The application's requirements suggest a user interface consisting of two windows:

- n A persistent main window that:
 - q Provides controls for initiating the required operations (writing a check, canceling a check, making a deposit)
 - q Displays the written checks and the current balance
- n A dialog box that gathers the input for writing a new check

Designing the Models

When you design the Checkbook application's models, you consider:

- n What real-world information and processes should the application represent?
- n Which portions, if any, are reusable in other applications (or elsewhere in the same application)?
- n Which portions are likely to change frequently, and which are likely to remain stable?

Designing Domain Models

The Checkbook application simulates a user's interactions with a checking account. A checking account is essentially a quantity of money that you spend by writing checks. Each check is identified by a unique number and authorizes the transfer of a specific amount of money to some payee. You keep a record of the checks you write by listing them in the register of a checkbook. You track the available amount of money in the account by keeping the running total, or balance, of the checks listed in the register.

From this description, at least two domain models suggest themselves:

- n A **Check**, which contains a unique number, a date, an amount, and a payee
- n A **Checkbook**, which contains a register and a running balance

Note that you could define an additional domain model (a **Deposit**) to handle deposits similar to checks. However, to keep the application simple, deposits are treated as numbers added to the balance.

Designing Application Models

The Checkbook application needs at least one application model to establish the connections between the widgets in the interface and the information in the **Check** and **Checkbook** objects.

You could choose to have two application models, one for each window, or you could choose to support both windows with one application model. The decomposition of application models is a matter of judgment about what information makes sense together and what is separately reusable. Assuming that neither window will be reused separately, you decide on a single application model, called **CheckbookInterface**.

You could choose to keep domain models separate or to incorporate their state and behavior into the application model. Assuming that the user interface will undergo constant revision but the checkbook-related information and processing will not, you decide to keep **Check** and **Checkbook** as separate classes.

What's Next: Constructing the Sample Application

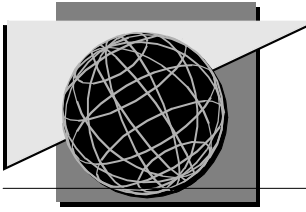
The next three chapters of this tutorial guide you through the following general steps in constructing the sample application:

1. Specify the layout and contents of the main window and the dialog box (Chapter 4).
2. Create and program the `Checkbook` and `Check` classes (Chapter 5).
3. Program `CheckbookInterface` to connect the specified widgets to appropriate information and actions (Chapter 6).

Note that when you build your own applications, you may perform similar steps but in a different order:

- n You may start with existing domain models to which you add a user interface.
- n You may complete the user interface and application model and then decide to split off several separate domain models.
- n You may go back and forth between the interface and the models, adding individual features incrementally.

The work you will do in the next three chapters is cumulative, so you should *save your image periodically*, especially before taking a break or exiting VisualWorks.



Chapter 4

Creating a Graphical User Interface

This chapter describes how to create the visual portion of a graphical user interface. For the Checkbook application, this means specifying the contents and layout of two windows:

- n The Checkbook main window
- n The Check dialog box

Later, in Chapter 6, you will learn how to program the application-specific behavior for the various elements you specified.

Designing the Checkbook Main Window

In the previous chapter, you established a preliminary design for the Checkbook application's main window. Now you refine the preliminary design into a more detailed design. This means deciding which controls, or *widgets*, you want in the window and how you want them to be positioned relative to each other. (See Appendix B for descriptions of the available widgets.)

As a start, assume that you design the window as shown in Figure 4-1. This design includes:

- n Widgets that display information—in particular, a *list* for displaying the collection of written checks and a read-only *input field* for displaying the balance
- n A widget that gathers input—in particular, an *input field* for entering deposit amounts
- n Widgets that invoke operations—in particular, a *menu bar* with *menus* for closing the application and for writing and canceling checks
- n Widgets that organize the window—in particular, *labels* that identify the purpose of various other widgets

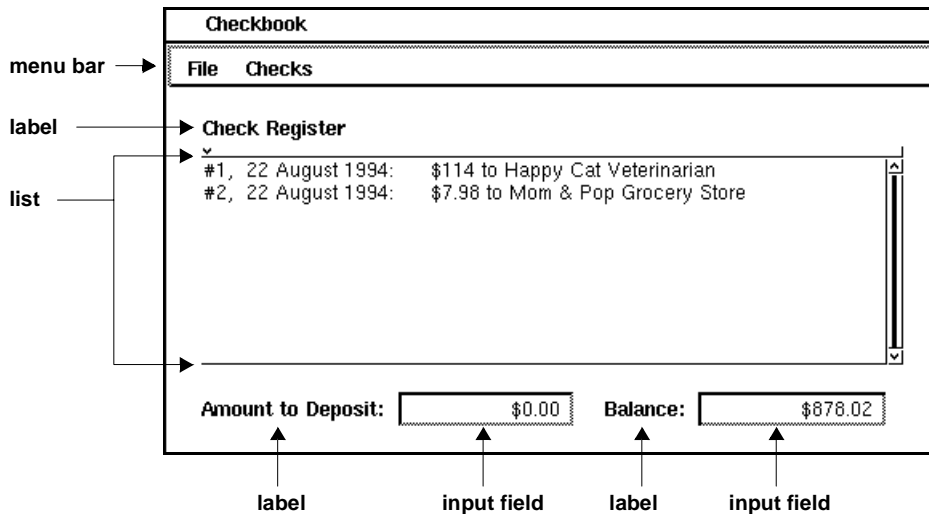


Figure 4-1 The Checkbook application's main window

Design Alternatives

Even an initial, rough-cut design makes choices among alternatives for presenting information and operations. For example, in the proposed design:

- n Each written check is represented as a line of text in a list. Alternatively, check information could be formatted in columns in a *table* or a *dataset*.
- n The deposit operation is invoked by entering an amount in an input field and pressing <Return>. Alternatively, the operation could be triggered more explicitly by an *action button* or an item on a menu.

The proposed design is a sufficient starting point, because it is the simplest design that meets the application requirements. In the next sections of this chapter, you will create a prototype of the main window with this design and then refine the prototype to improve its usability.

Creating the Main Window

You create the Checkbook main window by creating a visual specification of its contents and layout. This process includes the following steps, which are described in detail in the rest of this section:

1. Opening a blank *canvas*

2. *Painting* the canvas with widgets chosen from a *Palette*
3. Setting *properties* for each widget and *applying* them to the canvas
4. Editing the contents of any menus on the canvas
5. *Installing* the canvas in an application model

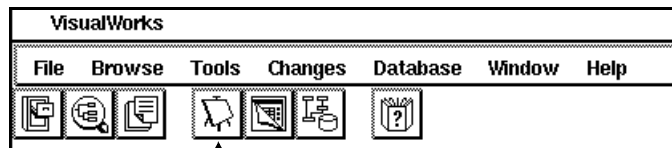
Opening a Blank Canvas

A VisualWorks canvas is a “window under construction”—a work area that you configure until it looks like the window you want in your application. In general, you create a separate canvas for each window in an application.

To open a blank canvas for the Checkbook main window, you:

1. Start VisualWorks, if necessary.
2. Choose **Tools?New Canvas** from the VisualWorks main window.

Shortcut: Click on the New Canvas button in the tool bar.



New Canvas button

3. Use the mouse pointer to position the rectangular window outline on the screen, and then click the <Select> button.

VisualWorks opens a window containing an unlabeled canvas, plus two additional tools, as shown in Figure 4-2:

- n A Palette of the standard widgets supplied by VisualWorks (these are described in Appendix B). You choose widgets from this Palette to paint them on the canvas.
- n A Canvas Tool that you use to fine-tune the appearance of the canvas and to invoke additional canvas-preparation tools.

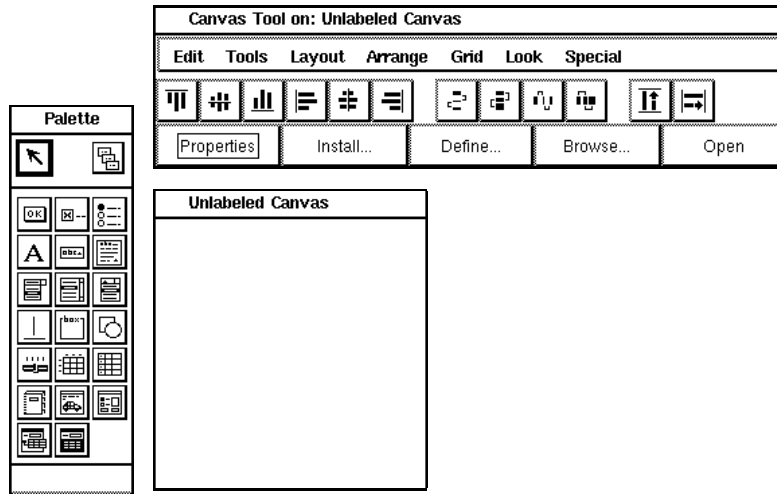


Figure 4-2 A blank canvas, its Palette, and its Canvas Tool

The displayed Palette and Canvas Tool are associated with this particular canvas; operations you invoke from them affect only this canvas. Every canvas you open has its own Palette and Canvas Tool associated with it.

Minimizing or closing a canvas automatically closes its Palette and Canvas Tool. However, you can move, minimize, or close these tools independently of the canvas. If you close a canvas's Palette or Canvas Tool, you can bring it back by positioning the mouse pointer in the canvas and choosing **tools?palette** or **tools?canvas tool** from the <Operate> menu.

Painting the Canvas

You paint the blank canvas with the main features of the Checkbook main window—a list, three input fields, and three labels. (You will add the menu bar in a later section.)

Sizing the Canvas

Before you start painting widgets, you resize the canvas to establish the preferred size of the Checkbook main window:

1. Use your window manager to enlarge the canvas window (use Figure 4-1 as guide).
2. Choose **Layout?Window?Preferred Size** from the Canvas Tool.

Setting a canvas's preferred size determines the initial size of the window when the running application opens it.

You can resize the canvas at any time while you paint it. However, resizing the canvas does not change its preferred size; to do this, you must choose **Layout?Window?Preferred Size** again.

Painting a Widget

To paint the required list onto the canvas:

1. Verify that the *single-selection button* on the Palette is active (it has a heavy, dark outline; see Figure 4-3). If it is not active, select it by positioning the mouse pointer over it and clicking the <Select> mouse button.

When active, the single-selection button allows you to paint a single copy of a widget on the canvas.

2. Select the list widget on the Palette. The *indicator field* at the bottom of the Palette displays the name of the selected widget (**List**).

If you select the wrong widget, select other widgets until the indicator field displays **List**.

3. Paint the list by moving the mouse pointer to the canvas and clicking the <Select> button. Figure 4-3 shows the painted list.

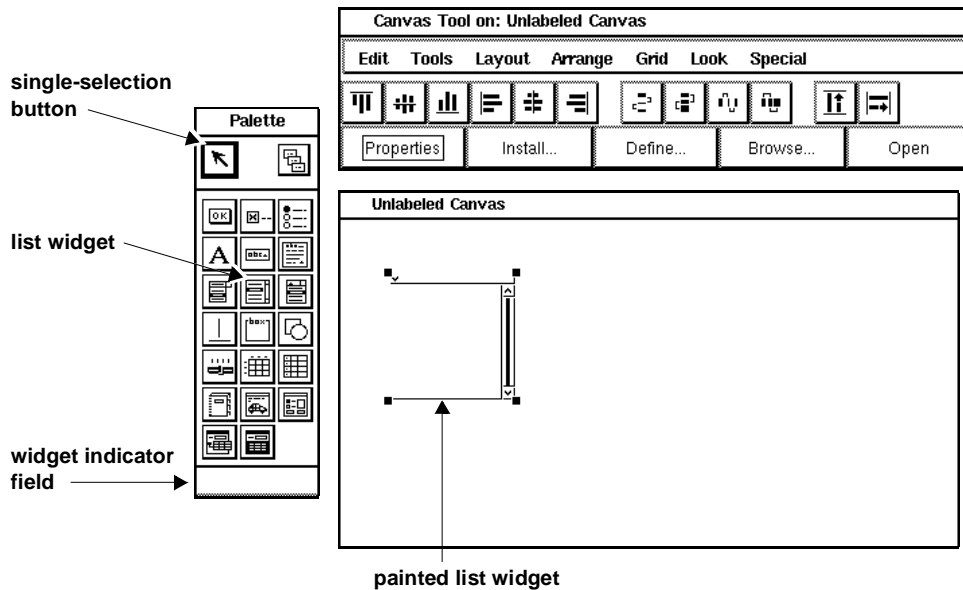


Figure 4-3 The canvas with a list widget painted on it

Selecting and Deselecting a Widget

Notice the *selection handles* (black squares) on the four corners of the list you just painted. They indicate that the widget is selected in the canvas. Practice deselecting and reselecting the list:

1. To deselect the list, either:
 - n Click the <Select> button anywhere in the canvas outside the list's selection handles.
 - n <Shift>-click inside the selection handles.
2. To reselect the list, click the <Select> button anywhere inside the list or on its borders.

Positioning a Widget

To position the list within the canvas:

1. Select the list, if necessary, and position the mouse pointer within the selection handles.
2. Press and hold down the <Select> button; then move the pointer. The list moves, too.
3. Drag the list to the desired position (use Figure 4-1 as guide), and then release the <Select> button. The list remains selected.

Resizing a Widget

To change the list's shape and size:

1. Select the list, if necessary.
2. Position the mouse pointer over one of the selection handles.
3. Press and hold down the <Select> button; then move the pointer. The corner of the list moves, too.
4. Drag the corner until the list is the desired shape (use Figure 4-1 as guide), and then release the <Select> button. The list remains selected. You may want to reposition it to accommodate its new size.

Copying and Pasting a Widget

Now you need to paint two input fields onto the canvas. You can paint each field individually, just as you painted the list, or you can use copy and paste as a shortcut.

To paint two input fields, using copy and paste:

1. Select the input field widget from the Palette and paint it on the canvas below the list.
2. With the field still selected, select **edit?copy** from the <Operate> menu.
3. Select **edit?paste** from the <Operate> menu. This makes a second copy of the field directly on top of the original one.
4. Drag the copy to the appropriate location.

Painting Multiple Copies of a Widget

Another shortcut for painting multiple copies of a widget is *repeat-painting*. To repeat-paint the three required labels:

1. Click the *repeat-selection button* on the palette (see Figure 4-4).

2. Select the label widget from the Palette.
3. Click on the canvas where each label is to appear (above the list and to the left of each input field).
4. Turn off repeat-painting by clicking the single-selection button.

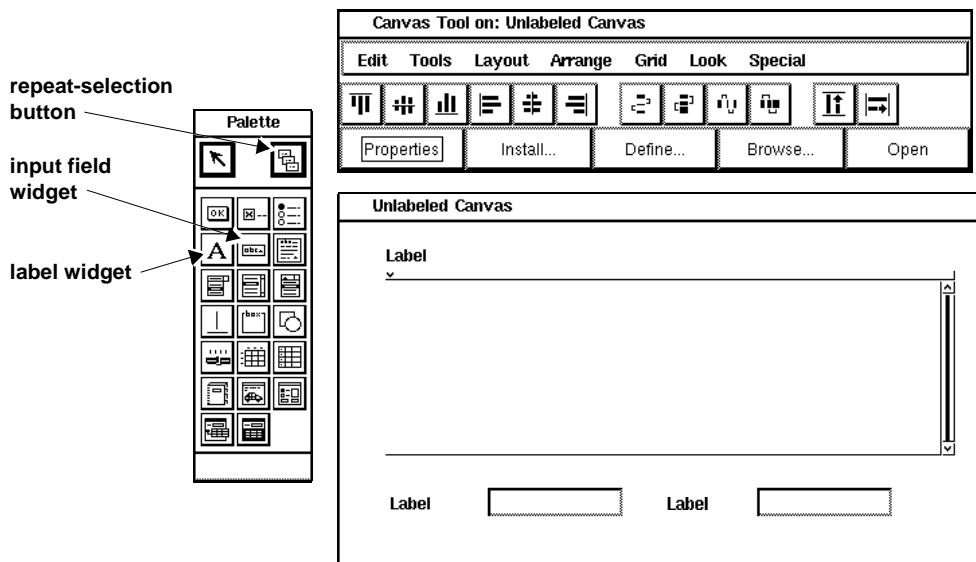


Figure 4-4 The canvas with a list, two fields, and three labels

Deleting a Widget

Sometimes you accidentally paint the wrong widget or too many copies of a widget. To delete a widget:

1. Select the widget to be deleted.
2. Select **edit?cut** from the <Operate> menu. This saves the widget to the canvas clipboard so you can paste it back in. On some platforms you can use the <Delete> or <Backspace> keys to delete a selected widget.

Setting Properties

Now that you have painted the basic elements of the Checkbook main window, you set properties for each widget and for the window itself. Properties define a variety of visual attributes, such as font, color, borders, and so on. For some widgets, such as input fields, properties also indicate the nature

of the data to be displayed and how that data is to be referenced by the application.

In this section, you will specify just the visual properties for the labels, list, and input fields. Later, in Chapter 6, you will specify the *action* and *aspect* properties when you program the Checkbook application's graphical user interface.

Displaying a Widget's Properties

To display the properties for a widget:

1. Select the widget in the canvas. In this case, select the label above the list.
2. Click the **Properties** button on the Canvas Tool.

VisualWorks opens the Properties Tool. You use the Properties Tool to examine and change the properties for the selected widget.

As shown in Figure 4-5, the Properties Tool displays the properties that apply to the current selection in the canvas. These properties are arranged in a notebook containing pages of related properties. You display other pages by clicking the tab of the page you want to see.

Note that the Properties Tool does not belong to a particular canvas the way the Canvas Tool and Palette do. Thus, a single open Properties Tool can be used for working on multiple canvases.

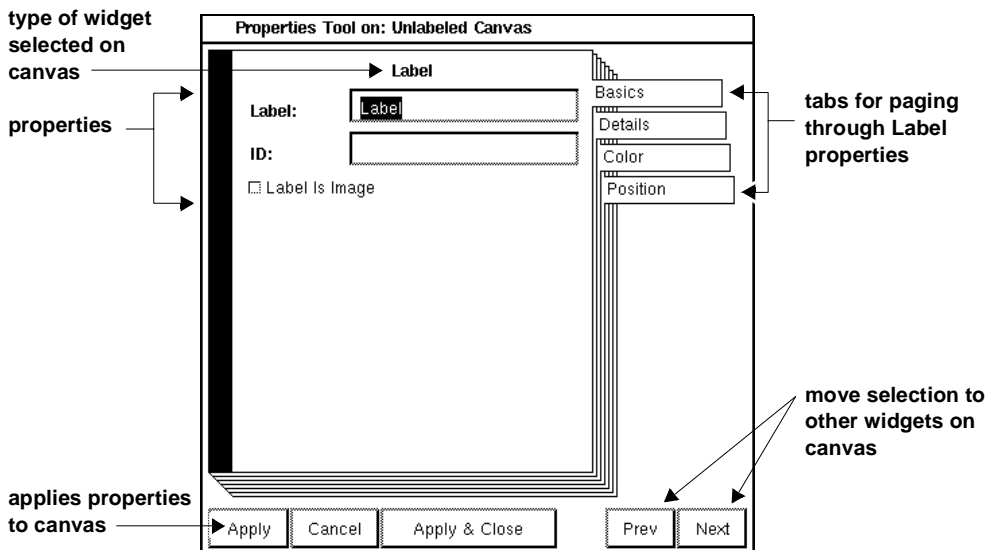


Figure 4-5 The Properties Tool, showing basic properties for a label

Applying a Changed Property

Like all of the labels on the canvas, the label you selected displays the default text **Label**. You change this text to suit the application by changing and *applying* a property. To do this:

1. Verify that the label above the list is still selected and that the **Basics** page is displayed in the Properties Tool.
2. In the Properties Tool, enter **Check Register** as the value of the **Label:** property.
3. Click **Apply**. The new label text appears on the canvas.

Applying properties adds information to the canvas, enriching the specification of the selected widget. For many properties, such as **Label**, the added information is also visible on the canvas.

Where appropriate, you can change multiple properties on a single page and apply them all at once. However, in this example, you need to change only one property for each label; the default settings are sufficient for the remaining properties.

Moving the Selection to the Next Widget

Now you need to display the properties for the remaining labels so you can set their text. One way to do this is to move the mouse pointer back to the canvas and select another label. Alternatively, you can use the **Next** and **Prev** buttons to move the selection on the canvas without moving the mouse pointer out of the Properties Tool.

To set the text for the remaining labels:

1. Verify that the **Check Register** label is still selected in the canvas.
2. Click **Next** on the Properties Tool to move the selection to the next label (in the lower-left corner of the canvas). The Properties Tool now displays properties for this label.

*Note: If necessary, click **Prev** or keep clicking **Next** until the appropriate label is selected.*

3. Change the **Label** property for the selected label by entering **Amount to Deposit:** and clicking **Apply**.
4. Click **Next** to move the selection to the last label.

*Note: If **Next** is disabled, click **Apply**. You must either apply or cancel changed properties before moving on to the next widget.*

5. Change the Label property for the selected label by entering **Balance:** and clicking **Apply**.

With their new, longer text, the labels may now overlap the input fields. If necessary, reposition the labels and fields to correct this.

Inspecting the List Properties

This example uses just the default property settings for the list widget. To see what these default settings are:

1. Move the selection to the list widget. This causes the Properties Tool to display the list's properties.
Notice that properties for a list differ from those for a label. You will return to the list properties shown on the **Basics** page in Chapter 6.
2. Click the tab for the **Details** page. Verify that the following properties are selected:
 - n **Vertical**, which provides the vertical scroll bar on the right edge of the list.
 - n **Bordered**, which provides the border surrounding the list.

- n **Can Tab**, which causes the list to be part of the tab chain; that is, when the application runs, the user will be able to move focus to the list by tabbing.

Setting the Input Field Properties

The input fields in this application will display amounts of money expressed in U.S. dollars. You set properties to control how the information appears in these fields. For each field:

1. Display the field's properties by selecting it in the canvas.
2. Select the nondefault property settings shown in the following table. (Look for the properties on different pages.)
3. Apply each page of changed settings before going on to the next page (or to the next widget).

| Widget | Page | Property | Setting |
|--------------------------|----------------|-------------------|--------------------------------|
| Input field (Deposit) | Basics | Type: | Number |
| | Basics | Format: | \$\$,##0.00;[Red](\$\$,##0.00) |
| | Details | Align: | Right |
| Input field (Balance) | Basics | Type: | Number |
| | Basics | Format: | \$\$,##0.00;[Red](\$\$,##0.00) |
| | Details | Align: | Right |
| | Details | Can Tab: | Off |
| | Details | Read Only: | On |

The **Format:** property controls the *output formatting* of each field. Thus, when the second field displays the current balance, the selected setting causes this number to be displayed with:

- n A preceding dollar sign
- n A comma separating the thousands and hundreds columns
- n Two decimal places for cents
- n Parentheses around negative numbers, which are displayed in red

Setting the Window Properties

You use window properties to provide the Checkbook main window with a title and a menu bar. To do this:

1. Deselect all the widgets in the canvas. You can either:

- n <Shift>-click on a selected widget
- n Click anywhere on the canvas other than in a widget

This causes the Properties Tool to display properties that apply to the window.

2. Set the window title by entering **Checkbook** for the **Label** property.
3. Create an empty menu bar by selecting **Enable** and entering **menuBar** for the **Menu** property. (The **Menu** property will be explained when you edit the menu bar.)
4. Apply these settings. The title of the canvas changes from **Unlabeled Canvas** to **Checkbook**, and an empty menu bar appears on the canvas.
5. If the empty menu bar displaces any other widgets, resize the canvas or reposition the other widgets as necessary.

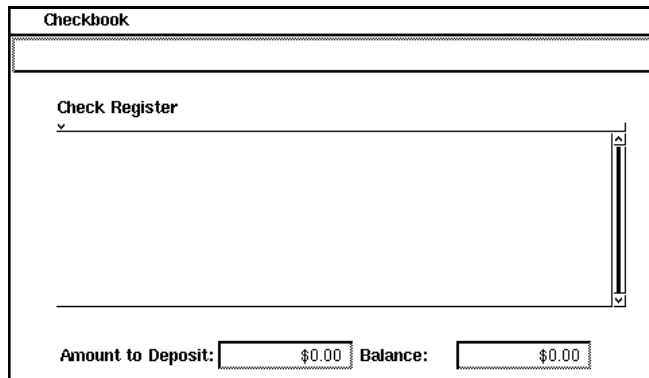


Figure 4-6 The Checkbook canvas with a menu bar

You have finished setting properties for the moment; if you like, you can close the Properties Tool. Be sure to save your image, especially if you plan to take a break. However, do not close the canvas until you have completed the next section.

Installing the Canvas

At any time in the painting process, you can save the canvas by *installing* it in an application model. Installing a canvas creates an *interface specification*, which serves as the application's blueprint for building an operational window. Each installed interface specification is stored in (and returned by) a unique method in the application model.

You can think of a canvas as the VisualWorks graphical user interface for creating and editing an interface specification. Whereas a canvas is a graphical depiction of the window's contents and layout, an interface specification is a symbolic representation that an application model can interpret.

To install the canvas for the Checkbook main window:

1. Click **Install...** in the Canvas Tool. This brings up a dialog box for specifying the application model and the class method in which to install the canvas.
2. In the **INSTALL on Class:** field of the dialog box, enter **CheckbookInterface**. (This is the name you chose on page 48 for the application model.)
3. Verify that the **enter new Selector:** field at the bottom of the dialog box contains the method name **windowSpec**.

By convention, **windowSpec** is the default name for a method that stores an interface specification for a main window (a window that is to be opened automatically when an application starts).

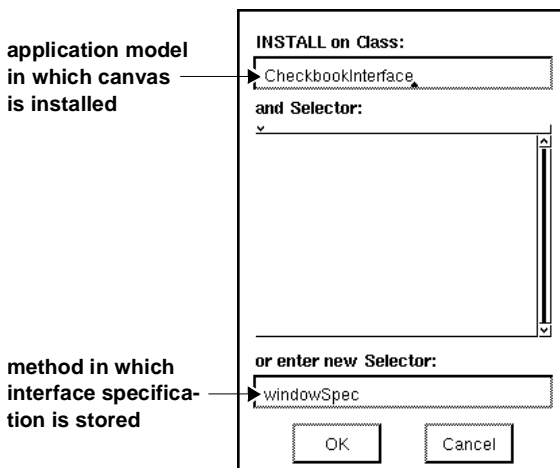


Figure 4-7 Installing the canvas

4. When the dialog box looks like Figure 4-7, click **OK**.
Because the **CheckbookInterface** class does not yet exist, an additional dialog box prompts you to create it.
5. In the **CREATE New Class** dialog box:

- a. Leave the **Name:** field as is (it should contain the name `CheckbookInterface`).
- b. Enter `Examples-VWTutorial` in the **Category** field to specify the category that is to contain the new class. (If you didn't create this category in Chapter 2, it is created for you in this step.)
- c. Click the **Application** radio button to specify the type of application model you want the new class to be. You choose **Application** because `CheckbookInterface` is to support a persistent window in a nondatabase application.

Note that this choice causes `CheckbookInterface` to be created as a subclass of the `ApplicationModel` class, which is part of the VisualWorks application framework.

- d. When the **CREATE New Class** dialog box looks like Figure 4-8, click **OK**.

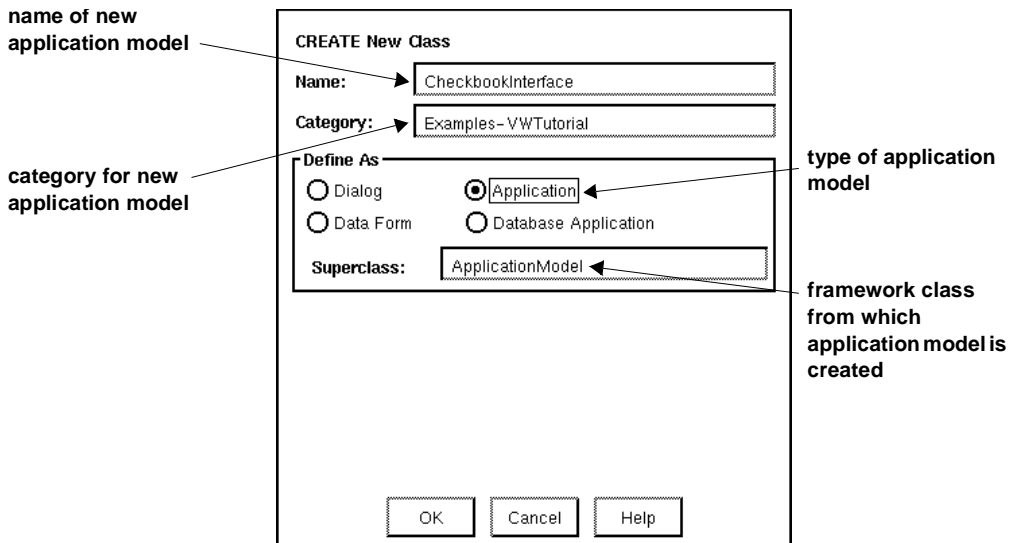


Figure 4-8 Creating the CheckbookInterface application model

6. Click **OK** again in the **INSTALL on Class** dialog.
7. Save your image to preserve the newly created application model.

As you will see in later sections, you can do a number of things with an installed canvas—you can start the application to see a running prototype of the window, and you can file out the application model so that you can file it

into another image. For now, though, continue on to the next section, which describes how to close and reopen the canvas.

Finding an Installed Canvas

Installing a canvas makes it possible to close the canvas and then open it again through the Resource Finder. To do this:

1. Close the window containing the canvas. The Palette and Canvas Tool close automatically.
2. Open the Resource Finder—for example, by clicking its icon in the VisualWorks main window.
3. In the Resource Finder, locate and select the `CheckbookInterface` class. Notice that `windowSpec` is listed as a resource of this class.

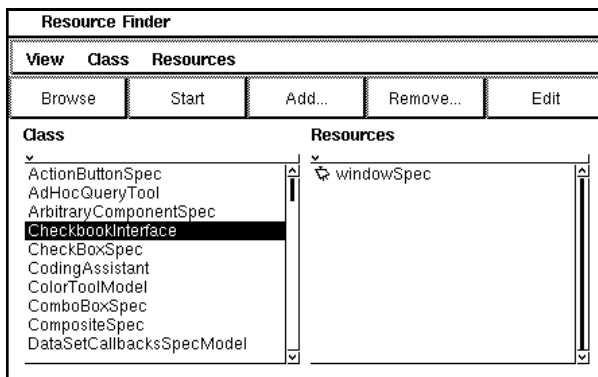


Figure 4-9 The Resource Finder with CheckbookInterface selected

4. With `windowSpec` selected, click **Edit** in the Resource Finder. This brings up the canvas whose interface specification is stored in `windowSpec`—that is, the canvas for the Checkbook main window.

Editing a Menu Bar

Your initial prototype of the Checkbook main window is almost complete; you still need to put menus on the empty menu bar. Assume that you have decided on two menus:

- n **File**, which contains a **Close** command for closing the application

- n **Checks**, which contains a **Write...** command for writing new checks and a **Cancel** command for canceling a selected check

You use the Menu Editor to create the menus that appear on the menu bar. More specifically, you create textual entries for the desired menu items, and the Menu Editor uses these entries to generate code for building an appropriate menu object. This code is then installed in a method in the application model, similar to the way an installed interface specification is stored. (You can use the Menu Editor to create a menu for any widget that provides a menu, such as a menu button.)

To create the menus for the Checkbook menu bar:

1. Bring up the canvas for the Checkbook main window, if necessary, and verify that you have completed steps 3 and 4 in “Setting the Window Properties” on page 62.
2. In the Canvas Tool, choose **Tools?Menu Editor** to open the Menu Editor for this canvas.
3. In the text area of the Menu Editor, type the menu titles (**File** and **Checks**) on separate lines.
4. Using Figure 4-10 as a guide, type a one-line entry for each menu item under the relevant menu title. Each entry must contain the following elements, from left to right:
 - n A leading <Tab> character.
 - n The text of the menu item’s label.
 - n One or more <Tab> characters.
 - n The name of the method that will perform the menu item’s action. Because such method names will not be established until Chapter 6, simply enter nil for now.

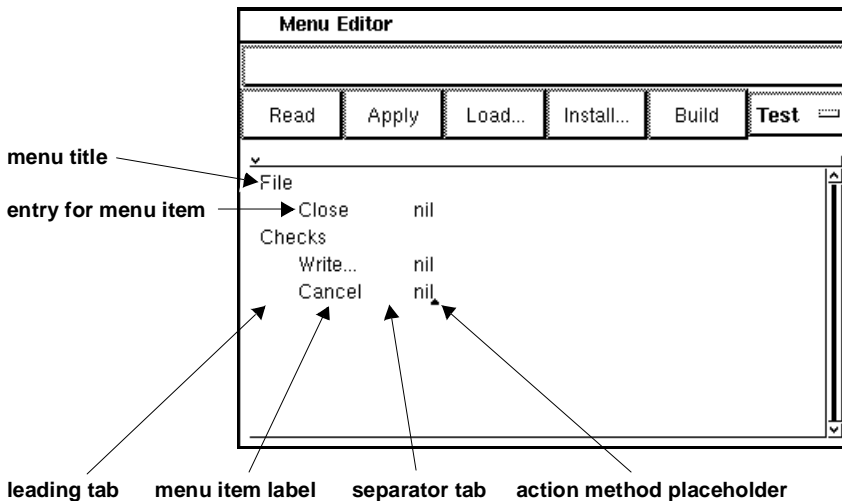


Figure 4-10 The Menu Editor with the menu bar contents

5. Click **Build** to generate code for building a menu object. A test version of the menu bar you just specified appears at the top of the Menu Editor.
6. Click on each menu title in the Menu Editor's test bar to verify that the menus contain the right items. If not, make corrections by repeating steps 4 and 5.

Note that the Menu Editor also provides a **Test** button. This is useful for testing menus created for menu buttons.

7. Click **Apply** to apply the tested menu bar to the canvas.

A dialog box first prompts you to install the menu code in a method called `menuBar` in `CheckbookInterface`. (Recall that `menuBar` is the name you entered for the **Menu** property on page 63.)

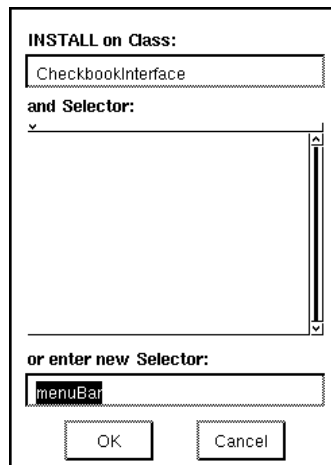


Figure 4-11 Installing the menu bar

8. Click **OK** to install the menu code. After the code is installed, the menus appear on the canvas.
9. Click **Install...** on the Canvas Tool to reinstall the canvas (you changed the canvas when you applied the menu bar to it). The **INSTALL on Class** dialog box appears with the method name `windowSpec` highlighted; click **OK**.
10. Notice that the Resource Finder now lists two resources for `CheckbookInterface`:
 - n `windowSpec`, which stores the interface specification for the Checkbook main window
 - n `menuBar`, which stores the menu code for the main window's menu bar
11. Close the Menu Editor and save your image!

Opening the Interface

Congratulations! You have completed the initial version of the Checkbook main window. By installing the window's canvas in an application model, you created a minimal application that can be started.

To start the minimal Checkbook application, you can:

- % Click **Open** in the Canvas Tool.

Alternatively, you can start the application from the Resource Finder:

1. Select `CheckbookInterface` in the Resource Finder.
2. Click **Start**.

Starting the Checkbook application opens the Checkbook main window, which looks something like Figure 4-12:

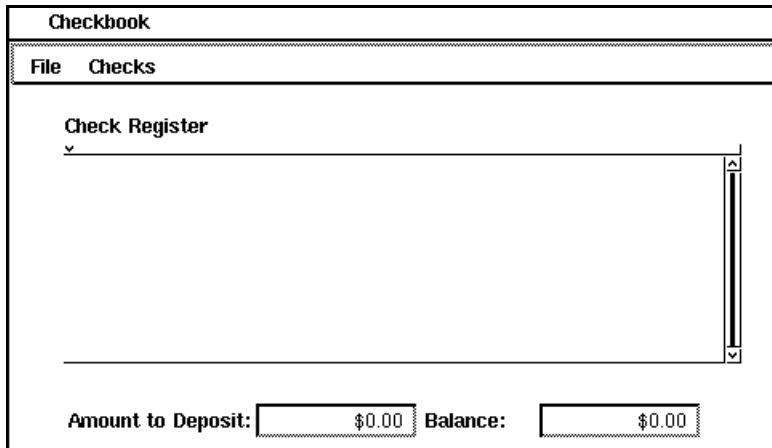


Figure 4-12 The Checkbook main window

Behind the Scenes

Regardless of how you start the application, the same thing happens from a Smalltalk point of view:

1. The Canvas Tool (or Resource Finder) sends an `open` message to the `CheckbookInterface` class.
2. The `CheckbookInterface` class understands this message (because it is an application model) and responds by creating an instance of itself.
3. This instance, in turn, creates a *builder*, which is an instance of a class in the VisualWorks framework called `UIBuilder`.
4. The application model's builder proceeds to build an operating window from the interface specification in the `windowSpec` method. That is, for each widget in the specification, the builder:
 - a. Identifies an appropriate widget class in the VisualWorks framework
 - b. Creates an instance of the identified class

The builder then assembles these instances, along with various other objects created from the framework, into a complete structure that forms the operating window.

This is a simplified account of a builder's activities; you will learn more details as they become relevant.

Inspecting the Prototype Window

At this point, the widgets in the Checkbook main window exhibit fairly generic behavior, because the rest of the application doesn't exist—there is no information for widgets to display and no actions for them to invoke. In spite of this, the various widgets respond minimally to mouse and keyboard input.

To see what response is built into the widgets themselves:

1. Select a menu item from the menu bar. Notice that when you click and drag on the menu title, the menu items are displayed correctly and your selection is highlighted appropriately.
2. Now click in the **Amount to Deposit:** field, type a number, and press <Return>. The input field knows how to accept the number you entered and redisplay it, formatted with a dollar sign and a decimal point. The input field also allows you to select and delete the number.

Notice that the number you enter in the deposit field has no effect on the balance field because the application has no notion of deposit and balance yet.

3. Try entering input in the **Balance:** field. The field prevents you from doing this because you selected its **Read Only** property on page 62.
4. Shrink and then enlarge the Checkbook main window. Notice that shrinking the window obscures some widgets, and enlarging the window exposes white space. This happens because the widgets, as created, have absolute sizes.
5. Close the running application (you'll have to use a window-management operation to close it because you haven't provided an action for **File?Close** yet).

In a real development situation, you might file out the application model as a backup or to share with another user. You can file the class out from the Resource Finder by selecting it, choosing **Class?File Out As...**, and specifying a filename.

Revising the Main Window

Most window designs undergo considerable revision through-out the development process. This occurs because the choice and placement of widgets is subject to numerous stylistic, usability, and aesthetic considerations. The canvas is a useful tool for window design because it allows you to experiment with different combinations and arrangements of widgets until you arrive at a prototype suitable for evaluation. You can iteratively edit and reinstall the canvas to incorporate recommendations.

The following sections describe how to make these improvements to the Checkbook main window:

- n Add an explicit control for invoking the deposit operation (some users may not realize they need to press <Return> in the deposit input field).
- n Refine the arrangement of widgets so that they align more precisely.
- n Make the window layout respond appropriately to resizing.

Adding More Widgets

There are many ways to provide an explicit control for the deposit operation. Assume that you have decided to add an action button for users to click after they have typed the desired deposit amount in the **Amount to Deposit:** field. To do this:

1. Open the canvas for the Checkbook main window by selecting both the `CheckbookInterface` class and its `windowSpec` resource in the Resource Finder and then clicking **Edit**.
2. If necessary, enlarge the canvas window vertically to make room for the new action button below the deposit field.
3. Select an action button from the Palette and place it below the deposit field.
4. With the action button selected, open the Properties Tool and apply the following property settings:

| Widget | Property | Setting |
|---------------|-------------------------|---------|
| Action button | Label: | Deposit |
| | Be Default: | On |
| | Size as Default: | On |

5. Enlarge the action button to accommodate its new label.

6. Enclose the three related widgets (the **Amount to Deposit:** label, the input field, and the action button) in a group box:
 - a. Select a group box from the Palette.
 - b. Position the box's upper-left corner on the canvas.
 - c. Press and hold the <Select> mouse button. The pointer moves to the lower-right corner of the box.
 - d. Drag the lower-right corner until the box fits around the desired widgets.
7. When the canvas looks something like Figure 4-13, install it in the `windowSpec` method.

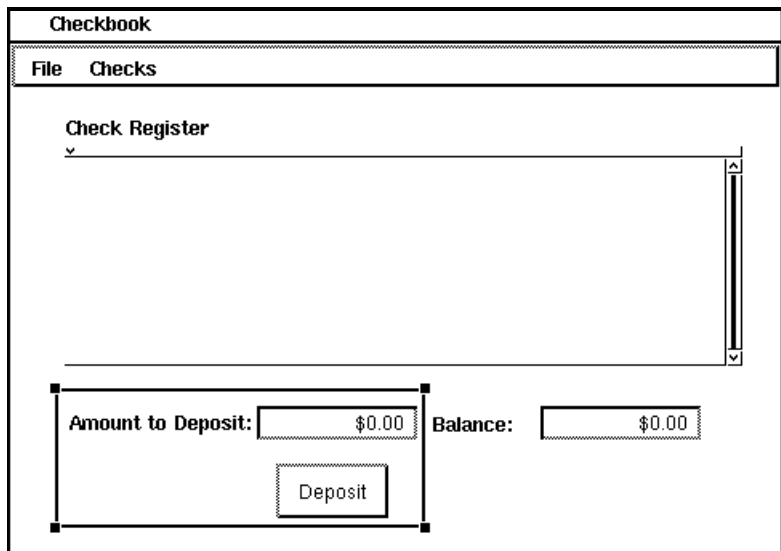


Figure 4-13 After adding an action button and a group box

8. Start the Checkbook application. Notice the **Deposit** action button's visual response when you click it.

Refining Widget Arrangement

So far, you've established the basic position and size of the widgets on the canvas using selection and dragging. In the following sections, you use the Canvas Tool and arrow keys to refine the arrangement of widgets so that they align more precisely.

Selecting Multiple Widgets

Most of the operations in the following sections involve selecting multiple widgets. In some operations, the order of selection counts; in others, all widgets are selected equally.

To select multiple widgets in order:

1. Click in the first widget to be selected. Its selection handles are solid squares.
2. <Shift>-click in each additional widget. Its selection handles are hollow. Note that <Shift>-clicking an already-selected widget turns off the selection.

To select multiple widgets in no particular order:

1. Put the mouse pointer on the canvas near one of the widgets.
2. While pressing the <Select> mouse button, drag the selection border around the desired widgets.
3. Release the mouse button; selection handles appear around each selected widget.

Equalizing Widget Sizes

You can use equalize operations to make two widgets the same size in one or both dimensions. These operations are available on the Canvas Tool's **Arrange** menu or on its tool bar:

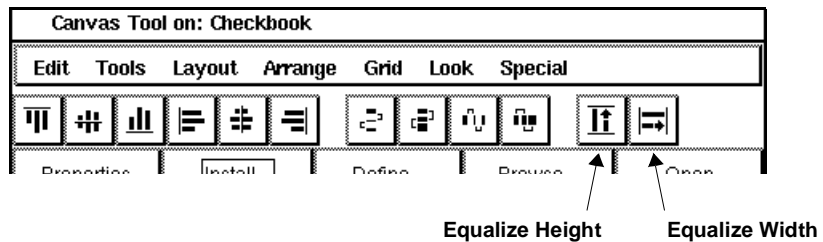


Figure 4-14 The Equalize buttons on the Canvas Tool

Resize one of the input fields, and then make the other field the same size:

1. Select one of the fields and drag a selection handle until the field is the desired size.
2. <Shift>-click to select the second field.

3. In the tool bar of the Canvas Tool, click the Equalize Height button (see Figure 4-14). The second field you selected is resized to match the height of the first field.
4. Leaving the widgets selected, click the Equalize Width button. The second field is resized to match the first field's width.

Aligning Widgets

You can use alignment operations to align widgets along their edges or centers, along their vertical or horizontal axes. These operations are available on the Canvas Tool's **Arrange** menu or on its tool bar:

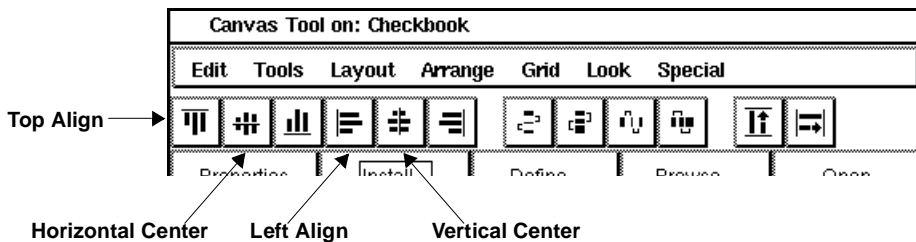


Figure 4-15 The Alignment buttons on the Canvas Tool

Align the left edges of the list and the **Check Register** label:

1. Select the list. This should automatically deselect the two input fields.
2. <Shift>-click to select the **Check Register** label.
3. Click the Left Align button (see Figure 4-15). The label is moved into alignment with the list.

Center the remaining labels relative to their fields. This means aligning these widgets around their horizontal centers:

1. Select the input field for deposits.
2. <Shift>-click to select the **Amount to Deposit:** label.
3. Click the Horizontal Center button (see Figure 4-15). The label is centered relative to the field.
4. Repeat steps 1–3 for the **Balance:** label and its field.

Center the action button relative to the deposit field above it. This means aligning these widgets around their vertical centers:

1. Make sure that the **Deposit** button is the desired size.
2. Select the input field for deposits.

3. <Shift>-click to select the **Deposit** button.
4. Click the Vertical Center button (see Figure 4-15). The button is centered relative to the field.

Spacing by Pixels

You can use arrow keys to move a selected widget a pixel at a time. For example, you can use the left and right arrow keys to adjust the space between the input fields and their labels.

Grouping Widgets

When you have arranged a set of widgets to your liking, you can *group* them into a single composite unit. Grouping a set of widgets prevents you from accidentally moving one of them out of alignment. Grouping also makes it possible to move an entire set of widgets while preserving their relative positioning.

Note: Grouping a set of widgets is different from painting a group box around them!

Group each input field with its label and then top-align the two groups:

1. Select the **Amount to Deposit:** label and the deposits field (order of selection doesn't matter).
2. Choose **Arrange?Group** in the Canvas Tool. Notice that a single set of selection handles surrounds the group.
3. Repeat steps 1 and 2 for the **Balance:** label and its field.
4. <Shift>-click to select the first group you created.
5. With both groups selected, click the Top Align button on the Canvas Tool.

Build up the group of deposit-related widgets:

1. Select just the group that contains the **Amount to Deposit:** label and its field.
2. <Shift>-click the **Deposit** action button.
3. Choose **Arrange?Group** again to include the action button in the group.
4. <Shift>-click on the group box widget surrounding the group.
5. Click the Horizontal Center and Vertical Center alignment buttons.
6. Choose **Arrange?Group** again to include the group box in the group.

Note that you cannot select an individual widget within a group. To make individual selections, you must select the group and then choose **Arrange?Ungroup** to dissolve the group.

Finally, align each group with the list and save the arrangement:

1. Left-align the deposits group with the list.
2. Right-align the balance group with the list.
3. Reinstall the canvas in the `windowSpec` method.

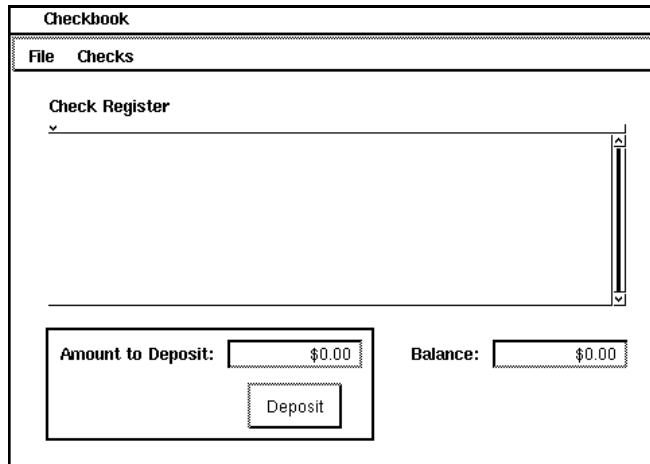


Figure 4-16 After fine-tuning the widget arrangement

Adjusting Window Layout

Recall from page 55 that the preferred size of the canvas determines the window's initial size when the application starts. If necessary, adjust the initial size by resizing the canvas and choosing **Layout?Window?Preferred Size**.

Now specify how the window should respond when users attempt to resize it. You can:

- n Allow users to resize the window. In this case, you probably want the widgets to resize or reposition themselves in proportion to the window. To specify this:
 - a. Select all of the widgets in the canvas.
 - b. Choose **Layout?Relative** from the Canvas Tool.

If you change your mind, reselect the widgets, if necessary, and choose **Layout?Fixed**.

- n Prevent users from resizing the window at all. To specify this, choose **Layout?Window?Fixed Size** in the Canvas Tool.

If you change your mind, choose **Layout?Window?Clear All**.

You are now finished with the Checkbook main window! Be sure to install the canvas and then save your image before going on to create the window for the Check dialog box.

Creating the Check Window

The Check window is a dialog box in which users enter information while writing a new check. Consequently, you design the window to resemble a physical check.

At this point, there is no difference between a dialog box and a persistent window. Both kinds of windows are built from interface specifications; the code that turns the Check specification into a dialog box will be written in Chapter 6.

The following sections briefly outline the steps for creating the Check window. Refer to previous sections if you need more detail.

Painting and Setting Properties

To paint the Check canvas:

1. Open a new, empty canvas. (If necessary, close the canvas for the Checkbook main window to make room.)
2. Using Figure 4-17 as a guide, select, position, and size the widgets on the canvas. Resize the canvas as necessary.

Hint: Two of the input fields are borderless (you specify this in their property settings below). You use dividers to simulate the partial borders around input fields 1 and 3.

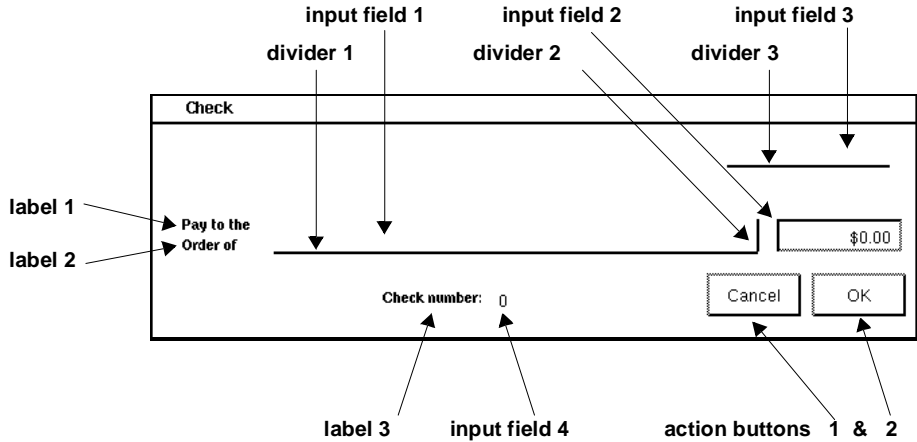


Figure 4-17 The widgets in the Check dialog window

3. Apply the nondefault property settings that are listed in the following table. (Look on the **Basics** and **Details** pages.)

| Widget | Property | Setting |
|---------------|---|---|
| Window | Label: | Check |
| Label 1 | Label: Font: | Pay to the Scaled Small |
| Label 2 | Label: Font: | Order of Scaled Small |
| Label 3 | Label: Font: | Check number: Scaled Small |
| Input field 1 | Type: Align: Bordered: | String Left Off |
| Input field 2 | Type: Format: Align: Bordered: | Number \$#,##0.00;[Red](\$#,##0.00) Right On |

| Widget | Property | Setting |
|-----------------|------------------------|----------------|
| Input field 3 | Type: | Date |
| | Format: | <your choice> |
| | Align: | Left |
| | Bordered: | Off |
| Input field 4 | Type: | Number |
| | Format: | 0 |
| | Align: | Left |
| | Bordered: | Off |
| | Read Only: | On |
| Divider 1 | Orientation: | Horizontal |
| Divider 2 | Orientation: | Vertical |
| Divider 3 | Orientation: | Horizontal |
| Action button 1 | Label: | Cancel |
| | Size as Default | On |
| Action button 2 | Label: | OK |
| | Be Default | On |
| | Size as Default | On |

4. Align widgets as necessary.
5. Adjust the window size and make the Check window a fixed size.
6. Install the Check canvas in a new method called `dialogSpec` in the `CheckbookInterface` class:
 - a. Click **Install...** in the Canvas Tool.
 - b. In the **INSTALL on Class:** field of the dialog box, type `CheckbookInterface`
 - c. In the **enter new Selector:** field, delete the default value and replace it with `dialogSpec` (if you leave `windowSpec` in this field, you will overwrite your main-window canvas).
 - d. Click **OK**.
7. Save your image.

Previewing a Window for Another Platform

If you are developing an application for use on several platforms, you can preview the canvas for each platform to see how the platform's look-and-feel will affect the appearance of the window.

To preview a canvas for a given platform:

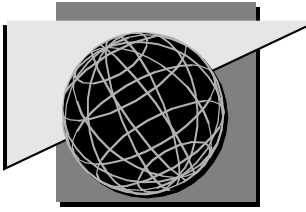
1. Display the canvas you want to preview.
2. From the **Look** menu on the Canvas Tool, choose the item that corresponds to the desired platform.

Changing the look from the Canvas Tool affects only the canvas; it does not affect the window's appearance when you run the application. The look of the running application is determined by the **UI Look** page of the Settings Tool (see page 35). This setting determines the look of all of your VisualWorks tools and applications.

What's Next: Programming in Smalltalk

So far, you have created specifications for the Checkbook application's graphical user interface and you have run the application in its current minimal form. Chapter 5 shows how to create the two Smalltalk classes that provide the basic processing for the application.

Because you will be working primarily with a System Browser and a Workspace, you can close any canvases, Resource Finders, Properties Tools, and Checkbook application windows that may still be open. Be sure to save your image if you want to take a break or exit VisualWorks.



Chapter 5

Developing the Domain Models


All Smalltalk code is bundled into classes. The most important role of a class is to create one or more instances of itself. A common analogy is that a class is like a factory that can manufacture a particular kind of object. For example, the `Float` factory can manufacture floating-point numbers.

In this chapter, you create the two Smalltalk classes that manufacture the domain models for the `Checkbook` application. Recall from “Designing Domain Models” on page 47 that these classes are:

- n The `Check` class, which manufactures check objects
- n The `Checkbook` class, which manufactures checkbook objects

What You Should Read

If You Are New to Smalltalk

To familiarize yourself with Smalltalk, you should work through the entire chapter. After you complete each task, be sure to read the related subsections whose titles begin with “ Analysis:”. These subsections highlight the Smalltalk rules and conventions that apply to the steps you performed. Note that this chapter is not a comprehensive introduction to Smalltalk. Consequently, you may find it helpful to consult any of the following for additional explanation:

- n Chapter 1 of the *VisualWorks Cookbook*
- n Chapters 2–8 of the *VisualWorks User’s Guide*
- n Any Smalltalk textbook

If You Already Know Smalltalk

To save time, you can work through just the main tasks in this chapter, skipping the Analysis sections. Alternatively, you can file in the completed application, browse its domain models, and then continue with Chapter 6. To do this:

1. In a File List, enter a pattern such as the following, where ***install-dir*** stands for the VisualWorks installation directory:

- n ***/install-dir/tutorial/basic/**** on a UNIX platform
- n ***vol:\install-dir\TUTORIAL\BASIC**** on a Windows or OS/2 platform
- n ***vol:install-dir:tutorial:basic:**** on a Macintosh computer

2. Select ***chkbk.st*** in the names view and choose **file in** from the <Operate> menu in that view.
3. In a System Browser, select the category **Examples-VWTutorial**. If necessary, choose **update** from the <Operate> menu in the category view.

This category now contains the **T_Check**, **T_Checkbook**, and **T_CheckbookInterface** classes. (The **T_** prefix prevents these classes from overwriting classes you have created, such as **CheckbookInterface**.)

4. Prepare the filed-in classes for use in Chapter 6:
 - a. Select **T_CheckbookInterface** in the class view and choose **remove...** from the <Operate> menu to delete this class.
 - b. Rename **T_Check** to **Check** by selecting **T_Check** and choosing **rename as...** in the class view's <Operate> menu.
 - c. In the dialog box, specify **Check** and click **OK**. A second dialog box informs you that existing methods reference the name you are changing.
 - d. Click **Rename** in the second dialog box. A browser is displayed, highlighting the old name in the referencing method.
 - e. In the browser, change the old name to the new name and choose **accept** from the <Operate> menu.
 - f. Repeat steps a–e to rename **T_Checkbook** to **Checkbook**.
5. Save your image.

Creating the Check Class

The **Check** class manufactures the check objects that are recorded in the checkbook. Each check object must store the information that defines a check and respond to messages that seek to obtain or change this information. Consequently, you program the **Check** class with *variables* for storing the required data and *methods* for manipulating it.

In the following sections, you will create the **Check** class by:

1. Locating the category for the Checkbook application's classes.
2. Defining the data structure (the class definition and instance variables) for the **Check** class.
3. Documenting the **Check** class with a class comment.
4. Defining methods that:
 - n Provide for access to the data in check objects
 - n Provide for character-based display of check objects

At various points in this process, you will create and inspect instances of the **Check** class.

Locating the Application's Category

In this tutorial, you will create all of the Checkbook application classes in a single class category—namely **Examples-VWTutorial**, where you created the **CheckbookInterface** class on page 64. This makes it easy to identify the application's pieces and to file them out as a group. Note, however, that you could scatter an application's classes among multiple categories, with no impact on the application's operation.

To locate the **Examples-VWTutorial** category:

1. Open a System Browser.
2. Scroll the category view until you find **Examples-VWTutorial**. It should be near the end of the list.

Alternatively, you can locate the desired category by searching for the **CheckbookInterface** class (see “Finding a Class by Name” on page 23).

3. Select **Examples-VWTutorial** in the category view. **CheckbookInterface** appears in the class view, as shown in Figure 5-1.

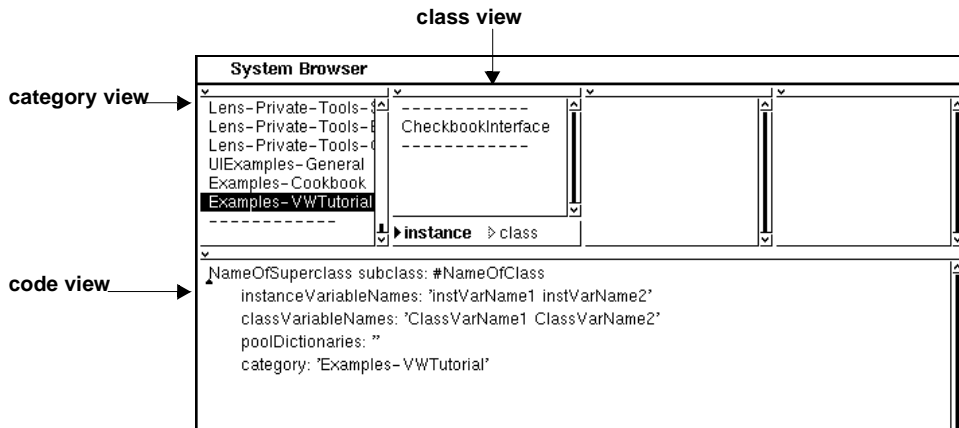


Figure 5-1 System Browser for Examples-VWTutorial category

Defining the Data Structure for the Check Class

Each check object must store a unique sequence number, a date, an amount, and a payee. You define the data structure for check objects when you create the **Check** class definition. The class definition specifies the names of the variables for holding data.

To create the class definition for the **Check** class:

1. Leave **Examples-VWTutorial** selected in the category view, and, if necessary, deselect **CheckbookInterface** in the class view.
The code view should display a *template* for a new class, as shown in Figure 5-1. The template is a formatted description of the basic parts of a class definition.
2. Leave the **instance** switch selected below the class view, because you are defining the data structure for all instances of **Check**.
3. In the code view, edit the class definition template as follows:
 - a. Replace **NameOfSuperClass** with **Object**.
Hint: Double-click on **NameOfSuperClass** to select it, and then type **Object** over the selection.
 - b. Replace **#NameOfClass** with **#Check**. Leave the pound sign (**#**), with no space between it and the class name.
 - c. Replace 'instVarName1 instVarName2' with the following list of instance variable names: 'number amount date payee'.
Hint: Position the mouse pointer between the initial quote and the following character. Double-click to select the entire quoted string, and then type the new instance variable names.
 - d. Delete **ClassVarName1 ClassVarName2**, leaving the empty quotes.
4. When the code view appears as in Figure 5-2, choose **accept** from the code view's <Operate> menu to compile the class definition. The new class name appears in the class view.

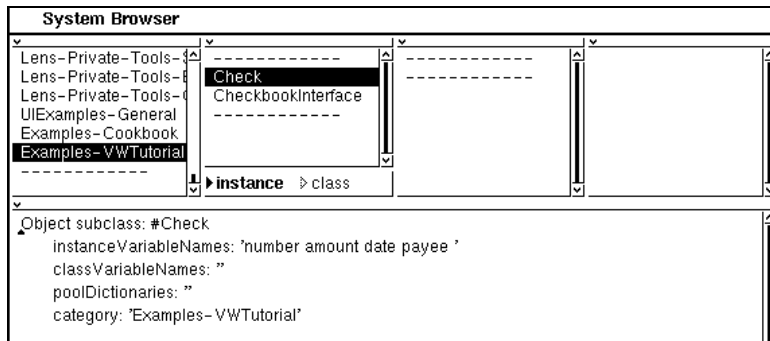


Figure 5-2 The Check class definition



Analysis: The Check Class Definition

You have just added a new class (**Check**) to the Smalltalk class library in your image. Because of step 3a, this class is a subclass of **Object**, from which it inherits variables and methods for printing, error handling, comparing, and so on. Similarly, as a class, **Check** inherits characteristic class behavior as well. Consequently, the new class can already respond to a number of messages, even though you have not yet defined any methods.

Each instance of **Check** will have *instance variables* named **number**, **amount**, **date**, and **payee**. These are called instance variables because they will exist for every instance created from the **Check** class—that is, for all objects manufactured by the **Check** factory. Each instance variable will eventually hold onto another object, which is its *value*. When an instance variable “stores” or “holds onto” an object, it essentially stores a reference to that object.

Each instance variable can hold an object of any type, although if a variable is initialized with an inappropriate type of object, that object probably won’t understand the messages it receives. In a later section, you will document the expected data types for these variables, and you will create the code that initializes them when you create the **Checkbook** class.

Creating a Check Instance

Although the `Check` class is incomplete (you have not defined its methods yet), the class definition you just created is capable of creating check objects. That is, like most classes, `Check` responds to the message `new` by creating a new instance of itself. To create and look at a check object:

1. Open a Workspace.
2. In the Workspace, type the following message expression:

```
Check new
```

3. Select the expression and choose **inspect** from the <Operate> menu. This opens an Inspector on the new check object. The Inspector displays a list of variables:

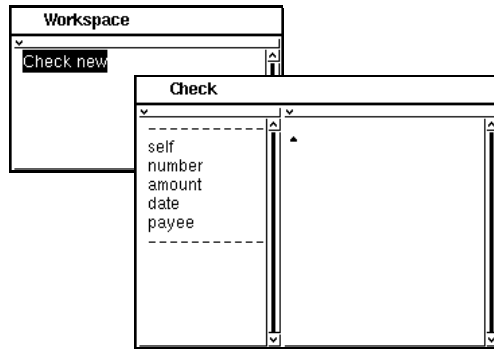


Figure 5-3 Inspecting a Check instance

The first variable in this list is `self`, which is a *pseudovvariable* that refers to the check object itself. The remaining variables are the instance variables in this object.

4. Select `self`. The phrase `a Check` is displayed as its value.
5. Select any of the listed instance variables to display its value. Because none of the variables are initialized, the value of each is `nil`.
6. Close the Inspector.



Analysis: Message Expressions

The message expression `Check new` consists of a *receiver* (the `Check` class) and a *message* (`new`). `new` is an example of a *unary message*, which consists of a single word, or *selector*. The selector is used for selecting the method that

is to be executed in response to the message. Thus, when the **Check** class receives the **new** message, the selector **new** is used to look up the method that creates a new instance.

Sending a message to a receiver always *returns* an object. The object to be returned is determined by the method that executes in response to the message. If the method doesn't specify an object to return, the default is to return the receiver itself. In the case of **new**, the object that is returned is the newly created instance of the receiver class. The Inspector displays the data structure of a returned object.



Analysis: Messages for Creating Instances

Most classes respond to **new** as part of their inherited class behavior. Many classes respond to other messages as well. For example, the class whose instances represent days in a year is **Date**. This class responds to the message **today** with an instance representing the current day. (Try entering **Date today** in a Workspace and inspecting it.)

Note that **new**, as inherited, creates new objects whose instance variables are empty (their values are *nil*). In contrast, a more specialized instance-creation method such as **today** typically creates an instance *and* assigns values to the instance variables. The **Check** class does not need such a method, although you will define one for the **Checkbook** class later in this chapter.

Documenting the Check Class

Whenever you create a new class, it is recommended that you document it for the benefit of other programmers who may read your code. At a minimum, the comment for the class should describe the class's purpose, variables, and methods.

To document the **Check** class:

1. Select **Check** in the class view, if necessary, and leave the **instance** switch selected.
2. Choose **comment** from the class view's <Operate> menu. The code view displays a default placeholder for a comment.
3. Replace the default comment with a comment such as the following:

The **Check** class is a container for the information that makes up a check. It has messages for accessing this information and for printing it on a **Stream**.

Instance Variables:

 number <Integer> Sequence number of check in checkbook
 amount <Integer> Amount of money for which the check is written
 date <Date> Date on which the check is written
 payee <String> Name of party receiving the check

4. Choose **accept** from the code view's <Operate> menu to incorporate the comment into the class.
5. Choose **definition** from the class view's <Operate> menu to redisplay the class definition in the code view.



Analysis: The Check Class Comment

The comment indicates the type of object that each variable is intended to hold. By convention, the expected object type is indicated by angle brackets. Note, however, that commenting the expected object type is not equivalent to declaring a data type for the variable, because Smalltalk allows any variable to hold any object.

When you explore the VisualWorks class library on your own, you can read the class comments for information about unfamiliar classes.

Providing for Access to Check Data

Whenever you create a class that has instance variables, you usually need to create methods for getting and setting the values of those variables. Such methods are called *accessing* methods, because they provide access to an object's data:

- n An accessing method that returns a variable's value is called an *accessor*.
- n An accessing method that sets a new value for a variable is called a *mutator*.

By convention, accessing methods are normally created in a protocol called **accessing**.

To create accessing methods for the **Check** class:

1. Select **Check** in the class view, if necessary, and leave the **instance** switch selected, because you are defining *instance methods*—methods that provide behavior for every instance of **Check**.
2. Create a new protocol:
 - a. Choose **add...** in the protocol view.
 - b. In the dialog box, enter the name **accessing** and click **OK**.

The code view displays a *template* for a new method. The template is a formatted description of the basic parts of a method definition.

3. Select the entire method template and replace it with the definition of the accessor method **amount**. The code for this method uses the return operator (^) to *return* the value of the **amount** instance variable:

```
amount
  ^amount
```

4. Choose **accept** from the code view's <Operate> menu. The name of the new method appears in the method view, as shown in Figure 5-4.

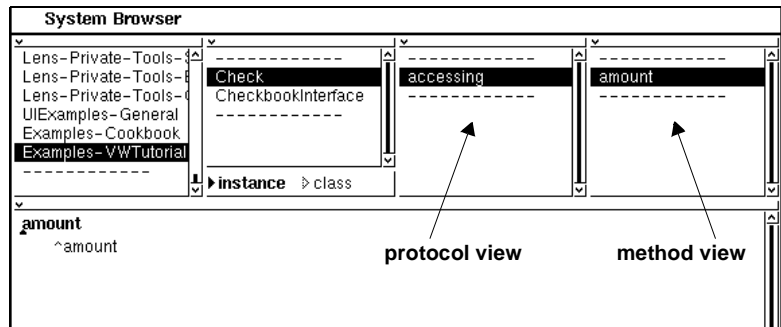


Figure 5-4 The Check Class with the amount method

5. Edit the text in the code view to define the mutator method **amount**:. Editing this text does not affect the previously accepted accessor method, because the colon you insert after **amount** defines a new selector for a new method:

```
amount: aValue
  amount := aValue
```

6. Choose **accept** from the code view's <Operate> menu. Notice that the method view now contains entries for two methods, **amount** and **amount:**.
7. Edit the code view and choose **accept** for *each* of the accessor methods shown below. You can create them in any order.

```
date
  ^date
```

```
number
  ^number
```

```
payee
  ^payee
```

8. Edit the code view and choose **accept** for *each* of the mutator methods shown below. You can create them in any order.

```
date: aValue
date := aValue
```

```
number: aValue
number := aValue
```

```
payee: aValue
payee := aValue
```

At this point, your Check class should look like Figure 5-5:

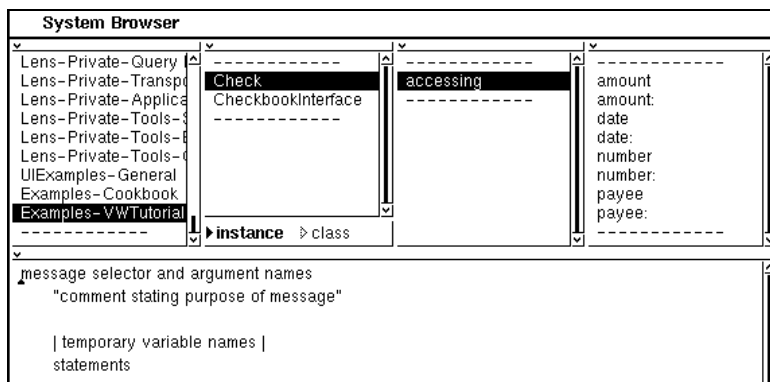


Figure 5-5 The Check class with complete accessing protocol



Analysis: Message Protocol

You have just created a number of instance methods in the Check class. Together, these methods define part of the *message protocol* for check objects. In general, an object's message protocol is its interface; it is the list of messages to which the object can respond. For example, a check object can now respond to a `date:` message by setting its `date` variable. Note that an object's complete message protocol also includes messages that are defined by inherited methods.



Analysis: Method Definitions

You created instance methods for the `Check` class by entering *method definitions* for them. In general, a method definition contains a *message pattern* and a sequence of one or more expressions.

In the following definition, the message pattern is `amount`, and the sole expression is `^amount`:

```
amount
    ^amount
```

When an instance of `Check` receives a message that matches this pattern (that is, when the check instance receives the message `amount`), this method executes by evaluating its expression.

Return Operator. The expression `^amount` uses the special return operator `^`. This operator causes the method to return the value of the following expression (in this case, the instance variable `amount`). Without an expression containing `^`, a method simply returns the receiver of the message.

Keyword Messages. The message patterns in the accessor methods are matched by unary (single-word) messages such as `amount`. In contrast, the message pattern for each mutator method is matched by a *keyword message*.

A keyword message is a message whose selector consists of one or more *keywords*, where a keyword is an identifier with a trailing colon (for example, `amount:`). Each keyword in a keyword message is followed by an *argument* expression (for example, in the keyword message `amount: 40`, the value `40` is the argument of the keyword `amount:`). Keyword messages provide a means of passing additional information for a method to use.

A keyword message such as `amount: 40` matches the message pattern in the following mutator method definition:

```
amount: aValue
    amount := aValue
```

When the method executes, the argument `40` is used in place of the argument name `aValue`.

Be sure to pay attention to the existence of colons in message selectors. The trailing colon signals the difference between a keyword message selector such as `amount:` and a unary message selector such as `amount`.

Assignment Operator. In the `amount:` method definition, the expression `amount := aValue` contains the assignment operator `:=`. This operator *assigns* the value of the expression on the right to be the new value of the instance variable named on the left. The expression in the `amount:` method definition causes the value of the message argument to be assigned to the instance variable `amount`.



Analysis: Naming Conventions

It is common practice to use minimally different names for an instance variable, its accessor, and its mutator. For example, for the instance variable `amount`, you created an accessor called `amount` and a mutator called `amount:.` (The only difference among these names is the colon in `amount:.`). The similarity of these names reinforces the notion that all three items pertain to the same aspect of a check object.

This naming convention is not required by Smalltalk. However, the convention is used throughout the Checkbook application because it helps to create more readable code, and, as you will see later, it makes it easier to set up certain objects from the framework that support the user interface.



Analysis: Method Compilation

When you enter a method in a code view and choose **accept**, the method is immediately parsed for syntax errors and then compiled into byte codes. At run time, invoking the method causes the byte codes to be translated to native machine code appropriate to the run-time platform. This native code is then cached so that subsequent invocations of the method do not require translation. Consequently, the performance of Smalltalk applications is comparable to that of statically compiled languages, but code is portable to any supported platform without recompilation.

Setting Check Information

Now that the `Check` class has accessing methods, you can create a check object and send it messages to set the values of its instance variables. To do this:

1. In a Workspace, type the following message expression:

```
Check new date: Date today
```

2. Select the expression and choose **inspect** from the <Operate> menu.
3. In the Inspector, select the `date` variable. Notice that its value is no longer `nil` but displays the current date. (The values of the remaining instance variables are still `nil`.)
4. Close the Inspector.
5. Replace the expression you typed in step 1 with the following sequence of expressions:

```
| aCheck |
aCheck := Check new.
aCheck date: Date today.
aCheck number: 1.
aCheck amount: 40.
aCheck payee: 'Fred'
```

6. Select all of these expressions and choose **inspect**. Now all of the instance variable have non-`nil` values.
7. Close the Inspector, but keep the Workspace.



Analysis: More about Message Expressions

Complex Expressions. Message expressions can serve as receivers or arguments in other message expressions. Thus, in step 1, you entered a keyword message expression in which:

- n The selector is `date:`.
- n The receiver is the check object returned by `Check new`.
- n The argument is the date object returned by `Date today`.

This complex message expression creates a new check object and sets its date to the current date.

Smalltalk evaluates complex message expressions according to a set of parsing rules. The rules that apply to unary and keyword expressions are:

- n Unary expressions are parsed from left to right.
- n Unary expressions take precedence over keyword expressions. Thus, the expression `Check new date: Date today` is parsed:

`(Check new) date: (Date today)`

Sequences of Expressions. You can resolve a complex expression into a sequence of simpler ones, typically by using *temporary variables*. For example, the expression you typed in step 1 can be written as follows (which forms the basis for the expressions in step 5):

```
| aCheck |  
aCheck := Check new.  
aCheck date: Date today
```

Here, `aCheck` is declared as a temporary variable. In the first expression, a new check object is created and then assigned as the value of `aCheck`. In the second expression, the `date:` message is sent to the value of `aCheck`. Notice that:

- n Declarations of temporary variables are enclosed between vertical bars.
- n In a sequence of expressions, all but the last one must end in a period.

Cascaded Expressions. When a number of messages are to be sent to the same receiver, they can be *cascaded* (separated by semicolons). For example, in step 5 you typed a sequence of expressions that send four messages to the same check object. Alternatively, you could enter a cascaded expression that sends the same four messages to the object returned by `Check new`:

```
Check new date: Date today; number: 1; amount: 40; payee: 'Fred'
```

Cascaded expressions are generally used sparingly; they are harder to read and debug than sequences of expressions.

Providing for Character-Based Display

Each instance of `Check` is capable of returning a string that describes itself. This is because the `Check` class inherits instance methods for *printing* from the `Object` class. (Note that in this context, “printing” refers to producing a displayable sequence of characters that describe an object, not to producing hard copy from a printer.) The basic way of obtaining a string description of an object is to send it the `printString` message.

Under the inherited implementation, instances of `Check` respond to `printString` with the default description `a Check`. (The Inspector displays this description when you select `self`.) You can cause check objects to print more informative descriptions by overriding their inherited behavior. You do this by reimplementing an instance method called `printOn:` in the `Check` class. This overrides the inherited `printOn:` method, which composes the actual string that `printString` prints. To reimplement the `printOn:` method:

1. Select `Check` in the class view, if necessary, and leave the **instance** switch selected, because you are defining another instance method.
2. Add a new protocol called `printing`.
3. Replace the method template with the following code. Enter the single quotes and commas exactly as shown:

```
printOn: aStream
```

```
"Print a description of this check on the provided stream.
```

```
Format of a sample description: #1, 4 August 1994:    $40 to Fred "
```

```
aStream nextPutAll:
```

```
'#, number printString, ', ', date printString, ':    $', amount printString, ' to ',  
payee displayString
```

Notice that:

- n This method contains a comment, which is enclosed in double quotes.
 - n The expression is broken across two lines. In general, you can format expressions with spaces, tab characters, and carriage returns to improve readability.
4. Choose **accept** from the code view’s <Operate> menu.
 5. You have completed the `Check` class! (Save your image.)



Analysis: Constructing a String

The last line in the method you entered is a single string, which serves as the argument in the `nextPutAll:` message. Strings are objects that represent sequences of characters. A literal string consists of one or more characters (including blank and tab characters) enclosed in single quotation marks, such as `' to '`.

You can build strings from other strings by concatenating them. To do this, you use the concatenation message, which is a comma (`,`). In the above method, four literal strings are concatenated with four expressions that return strings. Note that:

- n The comma inside single quotation marks is interpreted as a character in a literal string, not as the concatenation message.
- n The message `printString` produces strings describing the objects held by the `number`, `date`, and `amount` variables.
- n The message `displayString` is a variant of `printString` that returns the literal string held by the `payee` variable without the enclosing quotation marks.



Analysis: Streams

The `printString` method uses a *stream* to construct a string describing a check object. A stream is an object that holds onto a collection of elements (such as characters) and maintains a positional reference into this collection (for example, it knows which element is next). In general, streams are useful for constructing and retrieving sequences of elements, and for manipulating those elements in sequence.

`printString` creates an empty stream and then sends itself the `printOn:` message to insert an appropriate sequence of characters into the stream. (Note that the message pattern of the `printOn:` method expects a stream as an argument.) `printOn:` in turn asks the stream to store the specified characters by sending it the `nextPutAll:` message, which is part of the protocol understood by streams. When `printOn:` finishes, `printString` returns the contents of the stream—that is, the newly constructed string.

Displaying a Check Instance's Description

To see the effect of the new `printOn:` method, you can:

1. Select the code that you typed in the Workspace on page 97:

```
| aCheck |  
aCheck := Check new.  
aCheck date: Date today.  
aCheck number: 1.  
aCheck amount: 40.  
aCheck payee: 'Fred'
```

2. Choose **inspect** from the Workspace's <Operate> menu.
3. In the Inspector, select **self**. The object description now looks something like this (you may need to enlarge the Inspector window):
#1, 4 August 1994: \$40 to Fred
4. Close the Inspector.
5. With the the code still selected, choose **print it** from the Workspace's <Operate> menu. This prints the string description of the check object to the right of the last expression and highlights it.
6. Delete the highlighted string.
7. Now add another line of code in the Workspace to send the description to the System Transcript. The entire fragment should look like this:

```
| aCheck |  
aCheck := Check new.  
aCheck date: Date today.  
aCheck number: 1.  
aCheck amount: 40.  
aCheck payee: 'Fred'.  
Transcript show: aCheck printString; cr
```

8. Select the code and choose **do it** from the Workspace's <Operate> menu. The object description is now displayed in the System Transcript in the VisualWorks main window.



Analysis: The do it, print it, and inspect Commands

The <Operate> menu in a Workspace or a Browser code view contains several commands for executing Smalltalk code:

- n **do it** simply causes the selected expressions to be evaluated, declaring variables and evaluating message expressions as appropriate.
- n **print it** evaluates the selected expressions and prints a description of the object to which the last expression evaluates. The printed description appears to the right of the code and is highlighted so you can delete it easily.
- n **inspect** evaluates the selected expressions and opens an Inspector on the object to which the last expression evaluates.



Analysis: Method Lookup

When a message is sent to an object, Smalltalk uses a method-lookup mechanism to determine which method to execute. This “method finder” searches the methods in the receiver’s class for one with a matching selector. If none is found, the methods in that class’s superclass are searched next. The search continues up the superclass chain until a matching method is found. The search terminates with the **Object** class. If no matching method is found there, an error notifier reports that the message is not understood.

Thus, when you send the message `printString` to the object `aCheck`, the method finder:

1. Searches the methods in the **Check** class for a method whose pattern matches `printString`. No such method is found (you have defined `printOn: there`, not `printString`).
2. Searches the methods in **Check**’s superclass, which is **Object**. It finds the `printString` method there and executes it.

The same kind of lookup occurs for each message that is sent as part of the executing method. For example, the executing `printString` method contains the following message expression:

```
self printOn: aStream
```

When this message expression is evaluated, it:

1. Evaluates `self`, which stands for the receiver that initiated the lookup for the currently executing method. Because the currently executing method is `printString`, `self` refers to `aCheck`.
2. Sends the message `printOn:` to `aCheck`. This causes the method finder to search the methods in the `Check` class. Because you defined `printOn:` in the `Check` class, the search stops, and this method is executed.

Thus, by providing a local definition, the `Check` class overrides the inherited implementation of `printOn:`

Creating the Checkbook Class

The **Checkbook** class manufactures the checkbook object that is to be presented in the user interface. This checkbook object issues new checks, records the written checks in a register, and keeps track of the current account balance.

Like any Smalltalk object, the checkbook object does all this by storing data and responding to messages that are sent to it. Consequently, in the following sections you will create the **Checkbook** class by:

1. Defining the data structure (the class definition and instance variables) for the **Checkbook** class
2. Defining methods that:
 - n Provide for creating and initializing checkbooks
 - n Provide for accessing data in a checkbook
 - n Provide for checkbook transactions

At various points in this process, you will create and inspect instances of the **Checkbook** class.

Defining and Documenting the Checkbook Class

Each instance of the **Checkbook** class must store a register (a list of written checks), a balance (the amount of money available in the checking account), and the sequence number to be assigned to the next written check. You define this data structure by creating the **Checkbook** class definition:

1. Select the **Examples-VWTutorial** class category in the System Browser, making sure the **instance** switch is selected.

The code view may contain a new class template or the definition of an existing class in the category.

2. Edit the contents of the code view so that it contains the class definition shown below:

```
Model subclass: #Checkbook
  instanceVariableNames: 'balance register nextCheckNumber'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Examples-VWTutorial'
```

3. Choose **accept** from the code view's <Operate> menu to compile the class definition. The new class name (**Checkbook**) appears in the class view of the Browser.
4. Document the class by choosing **comment** from the class view's <Operate> menu and replacing the default comment with a comment such as the following:

Instances of the class **Checkbook** contain a register of written checks and a balance; they also assign sequence numbers to the checks listed in the register. The **Checkbook** class provides methods for creating and initializing new instances, accessing the information in them, and performing checkbook transactions such as making deposits and writing and canceling checks.

Instance variables:

 balance <Integer> The current balance of the checking account.

 register <OrderedCollection of: Check> The list of issued checks.

 nextCheckNumber <Integer> The sequence number of the next check.

5. Choose **accept** from the code view's <Operate> menu.
6. Choose **definition** from the class view's <Operate> menu to redisplay the class definition.



Analysis: Subclasses of Model

You have just created **Checkbook** as a subclass of the **Model** class. This means that **Checkbook** is automatically equipped with all the variables and methods defined in its superclass, **Model**, as well as those defined in **Model**'s superclass, **Object**.

Together, **Model** and **Object** provide **Checkbook** with the variables and methods that support the *dependency mechanism*, a widely used technique for coordinating the activities of different objects in an application. Because of this inherited mechanism, each **Checkbook** instance is capable of:

- n Maintaining a list of objects that depend on it for information
- n Notifying these dependent objects whenever the relevant information changes

In general, when an object is a *model* (that is, an instance of a subclass of **Model**), you can program other objects to set themselves up as dependents of it. You do this for objects that use information in the model and therefore need to know when that information changes so they can update themselves.

You make **Checkbook** a model because you know that the user interface will display checkbook information that is likely to change (such as the account balance). Later, in Chapter 6, you will set up dependencies that enable the user interface to update its display whenever the balance changes. Note that you did not make the **Check** class a model, because once check objects are created, the information in them never changes.

The term *model* now has two meanings:

- n Objects whose role is to store and manipulate data, as opposed to presenting it—for example, domain models and application models (see Chapter 3)
- n All subclasses of the **Model** class (and their instances)

These definitions generally overlap, because models (in the original sense) are typically implemented as subclasses of **Model**. For example, the application model **CheckbookInterface** is a subclass of the **ApplicationModel** class, which is a subclass of **Model**.

Creating a Checkbook Instance

At this point, you can create an instance of the `Checkbook` class the same way you created instances of `Check`—by invoking the inherited `new` method. To create and look at a checkbook object:

1. In a Workspace, type the following message expression:

```
Checkbook new
```

2. Select the expression and choose **inspect** from the <Operate> menu. The Inspector lists the checkbook object's variables:

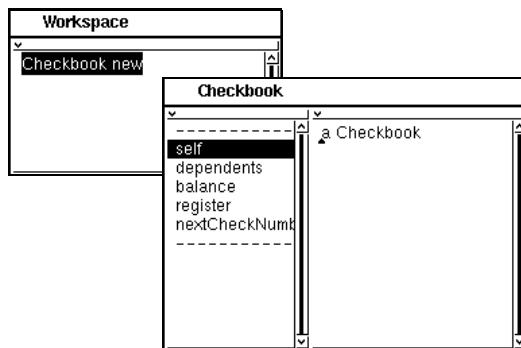


Figure 5-6 Inspecting a Checkbook instance

3. Notice that this list includes the variable `dependents`. This variable is part of the dependency mechanism inherited from `Model`.
4. Select any of the listed instance variables to display its value. Because none of the variables are initialized, the value of each is `nil`.
5. Close the Inspector.

Providing for Checkbook Initialization

When you create an instance of **Checkbook** using the inherited **new** method, its variables are all **nil**. You can override the inherited **new** method so that it both creates a new checkbook object and initializes its variables with objects of the proper types.

A typical reimplementaion of **new** is for it to send a message to initialize the new instance's variables. Consequently, in the following steps, you create two methods:

- n A class method **new**, which creates the new instance
- n An instance method **initialize**, which initializes the new instance

To create the appropriate methods:

1. Select **Checkbook** in the class view of the System Browser, if necessary.
2. Select the **class** switch so you can define a *class method*—a method that provides behavior for the class itself.
3. Add a protocol named **instance creation** (choose **add...** in the protocol view).
4. Replace the method template in the code view with the following method definition and choose **accept**:

```
new
  ^super new initialize
```

5. Select the **instance** switch so that you can define an instance method.
6. Add a protocol named **initialize-release**.
7. In the code view, enter the following method definition and choose **accept**:

```
initialize
  "Set up the checkbook with an empty register and a balance of $0"

  register := OrderedCollection new.
  balance := 0.
  nextCheckNumber := 1
```



Analysis: Initial Data Types

The `initialize` method you just created initializes a checkbook's instance variables by assigning objects to them. In particular:

- n `register` is assigned an empty instance of the `OrderedCollection` class. An ordered collection behaves like a dynamically expandable array whose elements are organized in the order in which they are added. As you will see, writing a new check adds an element to this ordered collection, and canceling a check removes it.
- n `balance` is assigned the object `0`, which is an instance of the `SmallInteger` class.
- n `nextCheckNumber` is assigned the object `1`, which is another instance of `SmallInteger`.
- n `dependents` is left uninitialized, because it is up to other objects to set themselves up as dependents on a model.

Initializing `balance` as an integer may seem odd, because dollar amounts usually have two-place decimals. However, `0` is simply an initial value to which another number (integer, floating point, or fixed point) can be added. In general, numbers of different types can be added or subtracted because their classes implement plus (+) and minus (-) methods that perform appropriate conversions. These implicit conversions illustrate that the `balance` variable is not strongly typed—that is, throughout the life of a checkbook, this variable may hold onto objects of different types.



Analysis: Class and Instance Methods

The `Checkbook` class now contains one instance method and one class method. In general:

- n Class methods define behavior for the class itself. Since only classes are capable of creating instances, instance-creation methods such as `new` are class methods.
- n Instance methods define behavior for every instance of the class. Since each instance is in charge of its own variables, the `initialize` method is an instance method.

You can view each kind of method in a System Browser by selecting the **instance** or **class** switch.

Creating an Initialized Checkbook Instance

Now you can test the `new` and `initialize` methods you just created:

1. In a Workspace, type the following message expression:

```
Checkbook new
```

2. Select the expression and choose **inspect** from the <Operate> menu.
3. Select each of the listed instance variables to verify that they have the correct initial values.
4. Close the Inspector.



Analysis: More about Method Lookup

When a message is sent to a class (such as `Checkbook`), Smalltalk’s “method finder” searches the receiver’s class methods for one with a matching selector. If none is found, the class methods for that class’s superclass are searched, and so on. Thus, when the receiver is an instance, instance methods are searched; when the receiver is a class, class methods are searched.

Sending the message `new` to the `Checkbook` class causes the method finder to search the class methods for `Checkbook`, where it finds the method you just implemented. The search stops here, instead of continuing through the class methods for `Checkbook`’s superclasses. The reimplemented `new` is executed, overriding the inherited one.

As it executes, the reimplemented `new` method evaluates the following message expression:

```
^super new initialize
```

This expression in turn:

1. Evaluates `super new`. As explained below, this creates a new `Checkbook` instance using the original, inherited implementation of `new`.
2. Sends `initialize` to the result of the previous step.
3. Returns the resulting initialized `Checkbook` instance.

In the expression `super new`, the role of `super` is similar to that of `self`, in that it stands for the receiver of the message that invoked the currently executing method. However, whereas `self` initiates a normal lookup for the message sent to it, `super` causes the method lookup to start with the super-

class of the class that defines the currently executing method. Thus, `super` allows you to reuse an inherited method even though you have overridden that method with a local reimplementation.

Returning to the example, in which the reimplemented `new` is the currently executing method, the `super new` expression:

1. Evaluates `super` to be the `Checkbook` class (the receiver of the message that invoked the currently executing method).
2. Sends `new` to `Checkbook`, thereby initiating a method lookup. Because `super` was used, this lookup skips `Checkbook`'s class methods and starts with the class methods for `Checkbook`'s superclass, namely `Model`.

This use of `super` allows the reimplemented `new` method to incorporate the inherited `new` method without duplicating its code.

***Note:** Although you might expect the `new` method to be listed among the class methods for the class `Object`, you will not find it there. The reason for this is rooted in the full explanation of class methods, which is beyond the scope of this tutorial.*

Providing for Access to Checkbook Data

Because the `Checkbook` class has instance variables, you need to consider providing accessing methods for them as you did for `Check`. For reasons described in the following subsection, you decide to create three methods:

- n An accessor method for the balance variable
- n An accessor method for the register variable
- n A mutator method for the balance variable

To create these methods:

1. Select `Checkbook` in the class view of the System Browser, if necessary, and leave the **instance** switch selected.
2. Add a protocol named `accessing` (choose **add...** in the protocol view).
3. Edit the code view and choose **accept** for *each* of the accessor methods shown below:

```
balance
  ^balance
```

```
register
  ^register
```

4. Edit the code view and choose **accept** for the mutator method shown below:

```
balance: anAmount
  "Set the balance to the specified amount, and notify
dependents of
  the change."

  balance := anAmount.
  self changed: #balance
```



Analysis: Limited Access to Variables

You just defined accessing methods for a subset of the instance variables in a `Checkbook` instance. With this message protocol, other objects can access a subset of a checkbook's information. For example, any other object can set the balance of a `Checkbook` instance simply by sending it a `balance:` message.

However, no other object can inspect or change the value of `nextCheckNumber`, because no accessing methods exist for this variable. The information in the `nextCheckNumber` variable is *private* to a check object because of the check's limited message protocol.

Deciding how much access to provide is often an iterative process. When you create a class, you may first give full access to all variables and later prune unnecessary accessors as you refine the application. In the case of `Checkbook`:

- n No access is provided for the `nextCheckNumber` variable, because no objects other than checkbooks need to know or change the sequence numbers for checks.
- n An accessor method for the `register` variable is provided for the user interface to obtain its collection of written checks. However, no mutator is provided, because once `register` is initialized, it should never be assigned another collection.
- n Full access is provided for `balance`. The accessor `balance` exists to support the user interface, and the mutator `balance:` is used in the implementation of other methods in `Checkbook`.

Strictly speaking, `balance:` is unnecessary, because no object other than a checkbook should ever set the balance. However, this method is useful because it consolidates code that would otherwise be repeated.



Analysis: Change Notification

The `balance:` method does more than set a checkbook's `balance` variable; it also notifies the checkbook that an aspect of its stored data has changed. Because the notified checkbook is a model (see page 107), it responds by broadcasting further notification to any objects that have made themselves dependent on it.

All this is set in motion by the `self changed: #balance` expression in the `balance:` method. For example, if you send a message such as `Checkbook new balance: 40`, then:

1. A new **Checkbook** instance is created (and initialized).
2. The **balance:** method:
 - n Changes the instance's **balance** variable from 0 to 40
 - n Sends a **changed:** message to **self** (in this case, the new **Checkbook** instance)
3. The new **Checkbook** instance finds the **changed:** method and executes it.
4. The **changed:** method causes the **Checkbook** instance to send an **update:** message to any objects listed in its **dependents** variable.

In this example, no objects have made themselves dependents of the **Checkbook** instance, so no **update:** messages are actually sent. When dependents exist, each executes its own **update:** method in response.

An important part of the **self changed: #balance** expression is the *symbol* (namely, **#balance**). A symbol is a string that is guaranteed to be unique in the system. For example, class and method names are symbols. Note that a symbol is expressed literally by prefixing it with the character **#**.

In the **changed:** message, the symbol **#balance** is an argument that represents the specific aspect of the checkbook's data that has changed. In this case, the aspect symbol **#balance** indicates that the value of the instance variable **balance** has changed. A **changed:** message passes its aspect symbol to each dependent, which uses the symbol to decide whether and how to respond to the change. If an object has multiple dependents that are concerned with different aspects of its data, each dependent can use the aspect symbol to filter out irrelevant change notifications.

You can pick any symbol as the aspect symbol in a **changed:** message. However, when the changed information is held in an instance variable that has an accessor (as in this example), it is common practice to choose an aspect symbol that matches the variable and accessor name. This practice makes it easier to set up dependent objects from the VisualWorks framework when you program the user interface.

Providing for Checkbook Transactions

So far, instances of **Checkbook** define an empty holder for check objects and store a balance of 0. Now you program the **Checkbook** class so that its instances can perform these transactions:

- n Deposit a specified amount of money into the checking account
- n Provide blank checks ready for issue
- n Record an issued check in the register
- n Cancel an issued check, removing it from the register

You can create the appropriate methods in any order. For example, you can:

1. Select **Checkbook** in the class view of the System Browser, if necessary, and leave the **instance** switch selected.
2. Add a protocol named **transactions** (choose **add...** in the protocol view).
3. Edit the code view and choose **accept** for the **deposit:** method shown below:

```
deposit: anAmount
    "Deposit the specified amount and update the balance
    accordingly"

    self balance: self balance + anAmount
```

The expression in the **deposit:** method:

- a. Obtains the checkbook's current balance
- b. Adds the specified amount to it
- c. Sets the result to be the new balance

4. Edit the code view and choose **accept** for the `makeNewCheck`: method shown below:

```
makeNewCheck
    "Create and initialize a new, blank check"

    | newCheck |
    newCheck := Check new.
    newCheck number: nextCheckNumber.
    newCheck date: Date today.
    newCheck amount: 0.
    newCheck payee: ''.

    ^newCheck
```

The expressions in the `makeNewCheck` method:

- a. Create a temporary variable and assign a new `Check` instance to it.
 - b. Initialize the `Check` instance by sending it cascaded messages from `Check`'s protocol. Notice that `payee` is set to the empty string (two single quotation marks).
Checks are initialized by a checkbook, because only a checkbook can determine the sequence number.
 - c. Return the resulting `Check` instance.
5. Edit the code view and choose **accept** for the `recordCheck`: method shown below:

```
recordCheck: aCheck
    "Add the check to the register."
    self register add: aCheck.

    "Update the balance to reflect the newly recorded check."
    self balance: self balance - aCheck amount.

    "Increment the sequence number."
    nextCheckNumber := nextCheckNumber + 1
```

The expressions in the `recordCheck:` method:

- a. Add the specified check to the end of the ordered collection held in the `register` variable. The `add:` message is part of the protocol understood by `OrderedCollection` instances.
 - b. Subtract the written check's amount from the current balance.
 - c. Calculate the sequence number that will be assigned by `makeNewCheck` to the next check it creates.
6. Edit the code view and choose **accept** for the `cancelCheck:` method shown below:

```
cancelCheck: aCheck
    "Remove the check from the register."
    self register remove: aCheck.

    "Update the balance."
    self balance: self balance + aCheck amount
```

The expressions in the `cancelCheck:` method:

- a. Remove the specified check from the ordered collection.
 - b. Add the amount of the canceled check back into the balance.
- Note that no adjustment is made to `nextCheckNumber`, because once a given sequence number is assigned, it is never reused. Canceling a check causes a gap in the sequence numbers.
7. You have completed the `Checkbook` class! (Save your image.)



Analysis: More about Complex Expressions

Three of the methods you just created (`deposit:`, `recordCheck:`, and `cancelCheck:`) contain message expressions that modify the checkbook's balance. These expressions are complex, in that each has arguments that are composed of other expressions. To see how these expressions are parsed, consider the following expression from the `deposit:` method:

```
self balance: self balance + anAmount
```

This expression contains the *binary message* `+`, which performs the addition operation. A binary message has one argument, and its selector is composed of one or two nonalphanumeric characters. Common binary messages include arithmetic operations (such as `+` and `-`) as well as the string concatenation message (`comma`).

When a message expression contains all three kinds of messages (unary, binary, and keyword):

- n Unary expressions are parsed from left to right.
- n Binary expressions are parsed from left to right.
- n Unary expressions take precedence over binary expressions.
- n Binary expressions take precedence over keyword expressions.

Returning to the example, assume that you send the `deposit:` message to a `Checkbook` instance. When `deposit:` executes:

1. The unary expression `self balance` is evaluated, returning the `Checkbook` instance's current balance.
2. The binary expression containing `+` is evaluated. Its receiver is the current balance returned in step 1; its argument is `anAmount`. The evaluated expression returns the resulting sum.
3. The keyword expression containing `balance:` is evaluated. Its receiver is `self` (the `Checkbook` instance), and its argument is the sum returned in the previous step. This expression sets the checkbook balance to be the new sum.

The evaluated expression can be written with parentheses as follows:

```
self balance: ((self balance) + anAmount)
```



Analysis: Alternative Implementation

The transaction methods (`deposit:`, `recordCheck:`, and `cancelCheck:`) use expressions such as `self balance` and `self balance:` to get and set the value of a checkbook's `balance` variable. Because these methods are defined in the `Checkbook` class, they can also use assignment expressions to manipulate `Checkbook`'s instance variables directly.

For example, the expression in the `deposit:` method could have been implemented as follows:

```
balance := balance + anAmount.
```

Deciding whether to reference variables directly or through accessing messages is often a matter of programming style. It is generally recommended that you use accessing messages because they promote maintainability—if, at some point, you decide to reimplement the structure of a class's data, you need to modify only the definitions of the relevant accessing methods, rather than modifying assignment expressions wherever they occur.

Furthermore, accessing methods promote reuse. For example, if the `Checkbook`'s transaction methods were to change the `balance` variable through assignment statements, the `self changed: #balance` expression would have to be repeated in each method definition.

Testing the Checkbook Transactions

You can test the transaction methods you just wrote by sending messages in the Workspace and displaying the results in the System Transcript. This technique is useful for testing domain models before the user interface is connected to them.

Use the following steps to test **Checkbook** transactions. If you encounter a syntax error, inspect the code for misspelled selectors or missing punctuation (see page 124 for hints).

1. For convenience, open a new Workspace and enlarge the VisualWorks main window.
2. Test the checkbook's ability to deposit into the account by entering the following code in the Workspace:

```
| b c1 c2 |  
  
b := Checkbook new.  
b deposit: 100.  
Transcript cr.  
Transcript show: 'Balance after 1st deposit: '  
               show: b balance printString;  
               cr.  
b deposit: 50.  
Transcript show: 'Balance after 2nd deposit: '  
               show: b balance printString;  
               cr.
```

Hint: Use **copy** and **paste** from the <Operate> menu to duplicate similar lines.

3. Select the code you entered and choose **do it** from the <Operate> menu. A dialog box informs you that the temporary variable **c2** is not used. In some cases, such a dialog box will alert you to an error or oversight in your code; in this case, the test code is incomplete, and the variable will be used later.
4. Click **proceed** in the dialog box for **c2** and again in the dialog box for **c1**.

5. Look at the System Transcript in the VisualWorks main window. It should contain these lines:

```
Balance after 1st deposit: 100
Balance after 2nd deposit: 150
```

6. Click on the code in the Workspace to deselect it.
7. Test the checkbook's ability to make and record checks by *adding* the following lines to the Workspace immediately *after* the code you entered in step 2:

```
c1 := b makeNewCheck.
c1 payee: 'Fred';
   amount: 70.
Transcript show: 'First check: ';
               show: c1 printString;
               cr.
b recordCheck: c1.
Transcript show: 'Balance after 1st check: ';
               show: b balance printString;
               cr.

c2 := b makeNewCheck.
c2 payee: 'Barney';
   amount: 20.
Transcript show: 'Second check: ';
               show: c2 printString;
               cr.
b recordCheck: c2.
Transcript show: 'Balance after 2nd check: ';
               show: b balance printString;
               cr.
```

Hint: Use **copy** and **paste** from the <Operate> menu to duplicate similar lines.

8. Select all the code in the Workspace (that is, all the code you entered in steps 2 and 7) and choose **do it** from the <Operate> menu.

9. Look at the System Transcript. The output should now include additional lines like the following:

```
First check: #1, 15 August 1994: $70 to Fred
Balance after 1st check: 80
Second check: #2, 15 August 1994: $20 to Barney
Balance after 2nd check: 60
```

10. Click on the code in the Workspace to deselect it.
11. Test the checkbook's ability to cancel checks by *adding* the following lines to the Workspace immediately *after* the code you entered in steps 2 and 7:

```
b cancelCheck: c1.
Transcript show: 'Balance after canceling 1st check: ';
    show: b balance printString;
    cr;
    show: 'Checks: ' ;
    show: b register printString;
    cr.
```

12. Select all the code in the Workspace (that is, all the code you entered in steps 2, 7, and 11) and choose **do it** from the <Operate> menu.
13. Look at the System Transcript. The output should now include additional lines like the following:

```
Balance after canceling 1st check: 130
Checks: OrderedCollection (#2, 15 August 1994: $20 to
Barney)
```



Analysis: Transcript Messages

You display strings in the System Transcript by sending messages to the receiver `Transcript`. `Transcript` is a predefined *global variable* in Smalltalk—a variable whose value can be accessed by all objects in the system. `Transcript` thus refers to a special instance of the class `TextCollector` that allows text to be displayed in the window region known as the System Transcript.

In this example, you sent:

- n The **show:** message with a literal string argument (such as 'First check:') or with an expression that returns a string (such as `b balance print-String`).
- n The **cr** message to insert a carriage return. This causes the next string to appear on a new line.

In a number of steps, you constructed a single line of output by sending multiple cascaded **show:** messages to **Transcript**. As explained on page 98, cascaded messages are separated by semicolons.

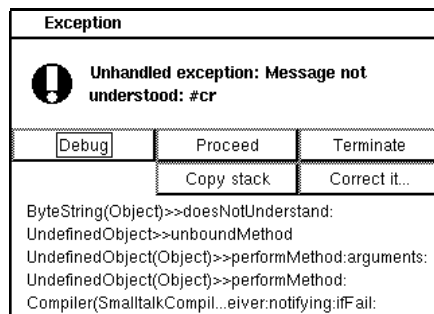


Analysis: Syntax Errors

If you accidentally misspell a word or leave out any punctuation (a single quotation mark, a colon, a period, or a semicolon), your expressions usually cannot be compiled. Depending on the context of the omitted punctuation, you could get any of the errors listed below.

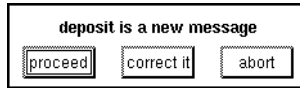
Note that a given type of error may occur for any of several reasons. To diagnose an error, you need to understand how Smalltalk is parsing your expressions (refer to the parsing rules on page 119).

- n An error notifier such as:



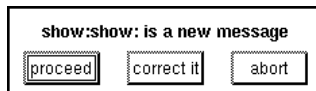
This notifier indicates that the message **cr** is being sent to a receiver that doesn't understand it. Click **Terminate** and inspect the code for missing punctuation in front of a **cr** message.

- n A dialog box such as:



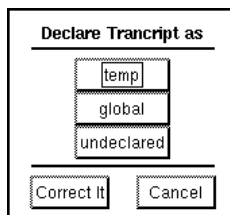
This dialog box indicates that the word **deposit** (highlighted in the Workspace) is parsed as a message selector that is not known to the system. Click **Abort** and inspect the code. The highlighted word may be misspelled or there may be missing punctuation around it. For example, the word may be a keyword selector that is missing a colon, or a period may be missing after a prior expression.

- n A dialog box such as:



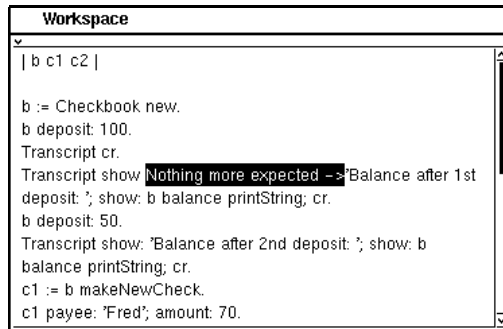
This dialog box indicates that the two adjacent occurrences of **show:** are parsed as a single keyword message that is not known to the system. Click **Abort** and inspect the code for missing punctuation in the highlighted text. For example, a semicolon may be missing between the argument of one **show:** message and the selector of the next.

- n A dialog box such as:



This dialog box indicates that a receiver is not known to the system and therefore is interpreted as the name of an undeclared variable. If the highlighted word is a misspelling of a known name, you can try clicking **Correct It**. Alternatively, you can click **Cancel** and correct the name.

- n The highlighted text **nothing more expected->** inserted in your code:



```
Workspace
| b c1 c2 |

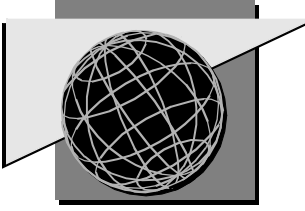
b := Checkbook new.
b deposit: 100.
Transcript cr.
Transcript show 'Balance after 1st
deposit: '; show: b balance printString; cr.
b deposit: 50.
Transcript show: 'Balance after 2nd deposit: '; show: b
balance printString; cr.
c1 := b makeNewCheck.
c1 payee: 'Fred'; amount: 70.
```

The highlighted text indicates that the preceding expressions are syntactically complete, even if they are not what you intended. Press <Delete> or <Backspace> to delete the highlighted error message and inspect the code for missing punctuation. The error may be in any expression prior to the highlighted text. In this case, the colon is missing after an occurrence of show.

What's Next: Programming the Interface

In Chapter 6, you will “glue together” all the pieces you have created so far. That is, you will program the application model (`CheckbookInterface`) so that it can connect the widgets in the interface specifications to appropriate checkbook information and actions.

As before, the work you will do in the next chapter is cumulative, so you should *save your image periodically*, especially before taking a break or exiting VisualWorks.



Chapter 6

Programming the Interface

In Chapters 4 and 5, you created:

- n Two interface specifications that describe the Checkbook application's graphical user interface
- n Two domain models (the `Checkbook` and `Check` classes) that provide the data and processing for checkbook and check objects

In this chapter, you program the application's graphical user interface to interact with checkbooks and checks. That is, you enable each widget in the interface to either display a particular piece of information (such as the checkbook balance) or invoke a particular action (such as writing a new check).

This chapter is divided into two major sections:

- n "VisualWorks Approach to Interface Programming," which gives an overview of the way behavior is defined for a VisualWorks graphical user interface
- n "Programming the Application Model," which guides you through the actual interface-programming steps

VisualWorks Approach to Interface Programming

In VisualWorks, you program a graphical user interface by programming one or more application models. As described in the following sections, you program these application models so that they:

- n Specify the interface's appearance and basic behavior
- n Supplement the interface's basic behavior with application-specific behavior

Specifying Basic Appearance and Behavior

You program an application model to specify an interface's basic appearance and behavior by installing one or more interface specifications in it. For example, in this tutorial, you programmed the application model (`CheckbookInterface`) to specify two windows by creating two canvases and installing them as interface specifications in the class methods `windowSpec` and `dialogSpec`. Each interface specification is a formula for generating an operational window containing particular widgets.

The generated widgets have built-in behavior that governs their appearance (size, location, color) and their response to user actions. Some of this behavior is fundamental to each widget (as when a selected menu item highlights itself). Other behavioral characteristics are explicitly chosen through property settings (as when a field either accepts input or is read-only).

More specifically, an interface specification contains information that tells a builder, in effect, how to choose widget classes from the application framework, create instances from these classes, and initialize these instances to endow them with the specified behavior. An interface specification is thus a means of requesting specific kinds of predefined interface behavior from the application framework.

The widgets produced by this level of programming have considerable functionality. For example, besides drawing itself in the proper size and location, the **Amount to Deposit**: field in the Checkbook main window knows how to accept input characters, format and display these characters as monetary amounts, convert these characters to numbers, and so on.

However, this functionality is necessarily limited to what is predefined in the widget classes. As part of a general-purpose framework, these classes simply cannot know anything about checkbooks, checks, or other domain objects you may create.

Programming Application-Specific Behavior

The next level of interface programming is to supplement the widgets' basic, predefined behavior with application-specific behavior—behavior that enables users to interact with the application.

For example, in the Checkbook application, you want users to make deposits by entering a deposit amount in a particular field and then clicking an action button. For this to work, the widgets in the interface must effectively know how to:

- n Put the entered amount where methods can access it
- n Find the relevant checkbook object and ask it to deposit the entered amount
- n Display the new balance

As created from framework classes, the widgets do not themselves hold onto data or carry out application-specific actions. Instead, they are designed to delegate these tasks to other objects. You set up these interactions by programming the application model to accommodate the predefined behavior of:

- n *Action widgets*—widgets such as action buttons and menu items that enable a user to invoke an application's actions
- n *Data widgets*—widgets such as input fields and lists that display some aspect of an application's data and/or collect it from the user

Note that widgets serving as purely visual elements, such as labels, dividers, and group boxes, do not require further programming.

Action Widgets

An action widget is designed to delegate its action to the application model from which it was built. Thus, when a user activates an action widget (for example, by clicking or selecting it), the widget knows to respond by sending a message to its application model, requesting that the application model carry out the desired action.

As shown later in this chapter, you set up this interaction by:

- n Telling the action widget which message to send
- n Providing the application model with a corresponding method that implements the desired action

Note that this action method may, in turn, send messages to domain models to pass information to them and/or invoke domain-specific operations.

Data Widgets

A data widget is designed to use an auxiliary object called a *value model* to manage the data it presents. That is, instead of holding onto the data directly, a data widget delegates this task to a value model:

- n When a data widget accepts input from a user, it sends this data to its value model for storage.

- n When a data widget needs to update its display, it asks its value model for the data to be displayed.

As shown later in this chapter, the basic way you set up this interaction is by:

- n Telling the widget the name of its value model.
- n Programming the application model to create and return the value model. Depending on the application's needs, the created value model either provides its own storage for data or accesses data that is held in a domain model.

When the interface is opened, the builder obtains the name of the value model from the widget and then uses this name to request the relevant value model from the application model. The builder then sets up the necessary connections between the widget and its value model.

More about Value Models

Value models are instances of the subclasses of `ValueModel`, a class in the application framework. Data widgets are designed to interact with value models because:

- n Value models define a uniform protocol for accessing data. This enables all data widgets to store and refresh their data in a standard way (by sending `value:` and `value` messages), regardless of where this data is held by the application.
- n Value models are specializations of the class `Model` (see page 107), so they notify their dependents whenever changes are made to the managed data. Each data widget is set up as a dependent on its value model, so it can receive this change notification and update its display in response.

You will learn more about value models when you set them up for the Checkbook application's data widgets.

Another Look at Application Structure

Value models introduce an additional layer of objects in the information model of the application (see Figure 6-1). Like other kinds of models, value models are concerned with storing and retrieving data, not providing display services. However, whereas other models tend to define complex aggregates of data, each of the value models manages a single piece of data for an individual data widget.

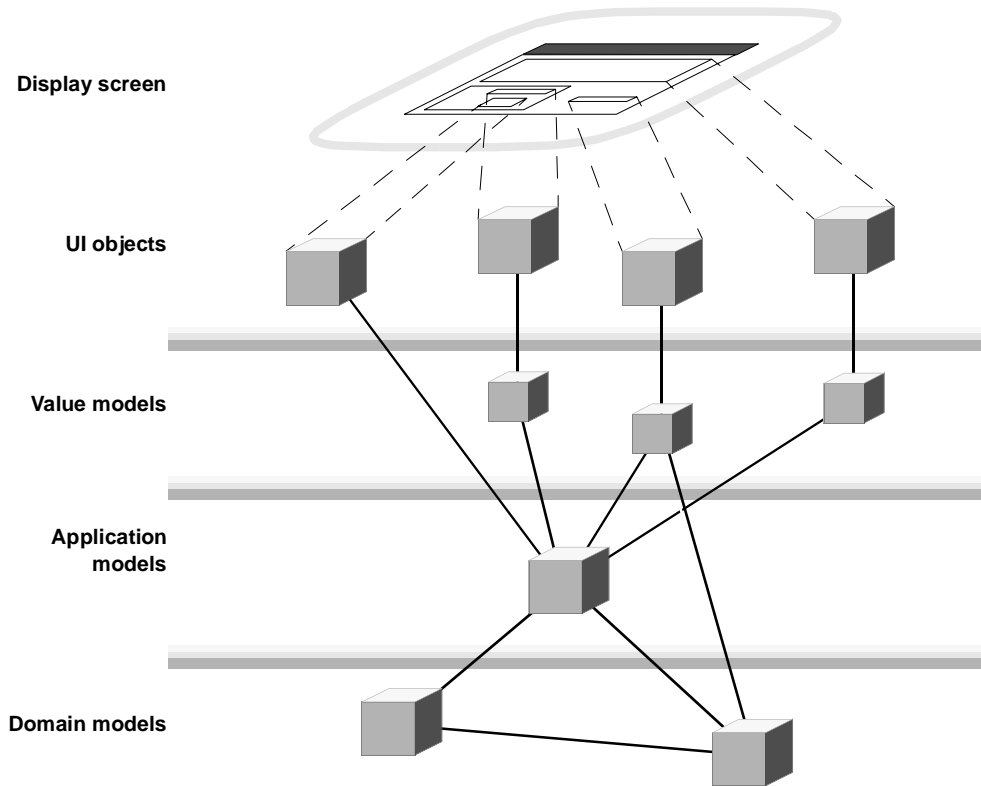


Figure 6-1 Value-model layer within the information model

Programming the Application Model

The application model for the Checkbook application is the class `CheckbookInterface`, which you created when you installed the interface specification for the application's main window. Through inherited behavior, this class already knows how to start the application—that is, how to create an instance of itself, tell this instance to create a builder, and give the builder the interface specification for the main window.

In the rest of this chapter, you will program `CheckbookInterface` so that it can carry out actions for the action widgets and provide appropriate value models for the data widgets. You do this through a combination of:

- n Refining the contents of the class methods (by setting additional widget properties and editing the menu bar)
- n Creating instance variables and methods (by using what you learned in Chapter 5 in combination with the VisualWorks interface-coding accelerator)

To program the class `CheckbookInterface`, you will:

1. Set up your environment with the relevant tools.
2. Browse `CheckbookInterface` to get acquainted with what is already there.
3. Provide the application model with a `Checkbook` instance.
4. Program the **Amount to Deposit:** field, the **Deposit** button, the **Balance:** field, and the **Check Register** list in the Checkbook main window.
5. Program the main window's menu bar.
6. Provide a method for writing a new check through the Check dialog box.
7. Program the fields in the Check dialog box.
8. Provide a method for check canceling.

At various points in this process, you will run the application to test its interface.

Setting Up Your Work

In the sections that follow, you will be setting widget properties and editing class and method definitions. To prepare for this work:

1. Arrange your screen so that it contains:

- n The VisualWorks main window
- n A System Browser
- n The canvas for the Checkbook main window


Hint: Open a Resource Finder; select both the `CheckbookInterface` class and the `windowSpec` resource, and then click the **Edit** button. You can close the Palette, but leave the Canvas Tool open.

- n The Properties Tool

Hint: Click the **Properties** button on the Canvas Tool.

2. Close any other windows you may have accumulated, such as the workspaces and inspectors you used in Chapter 5 and the Resource Finder you used above.

A Few Reminders

Most of the tasks that follow have related subsections whose titles begin with “ Analysis:”. As in Chapter 5, these subsections provide extra explanation about the steps you performed. Some of them highlight Smalltalk rules and conventions; others provide details about VisualWorks tools or the VisualWorks application framework. Depending on your learning style, you may read these subsections as you encounter them, or you may prefer to skip these subsections and return to them when you need to know more.

Remember that you can file in a completed version of the Checkbook application, as described on page 84, steps 1 through 3.

As always, be sure to save your image before taking a break or exiting VisualWork.

Browsing the Application Model

You created the application model `CheckbookInterface` when you installed the canvas for the main window. To familiarize yourself with this class, you:

1. Select `Examples-VWTutorial` in the category view of the System Browser.
2. Select `CheckbookInterface` in the System Browser's class view.
3. With the **instance** switch selected, examine the class definition. Notice that `CheckbookInterface`:
 - n Is a subclass of the class `ApplicationModel`
 - n Has no instance variables or instance methods at this point (other than what it inherits from `ApplicationModel`)
4. Select the **class** switch. Notice that `CheckbookInterface` has two class method protocols:
 - n The **interface specs** protocol, which contains the methods in which you installed the canvases for the main window and dialog box
 - n The **resources** protocol, which contains the method in which you installed the main window's menu bar

Providing the Checkbook Behind the Interface

The Checkbook main window exists to enable users to interact with a checkbook object. Consequently, instances of `CheckbookInterface` must hold onto an instance of `Checkbook` so that methods in `CheckbookInterface` can access the checkbook's data or invoke checkbook transactions.

To provide a checkbook for the interface:

1. Display the class definition for `CheckbookInterface` in the System Browser (make sure `CheckbookInterface` is still selected in the class view and then select the **instance** switch).
2. Edit the contents of the code view to add an instance variable called `checkbook` as shown:

```
ApplicationModel subclass: #CheckbookInterface
  instanceVariableNames: 'checkbook'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Examples-VWTutorial'
```

3. Choose **accept** from the code view's <Operate> menu.
4. Add a protocol named `initialize-release`.
5. In the code view, enter the following method definition and choose **accept**:

```
initialize
  "Create a new checkbook for the interface to manipulate."

  checkbook := Checkbook new
```

This method creates a new `Checkbook` instance and assigns it to the `checkbook` instance variable.



Analysis: Initializing an Application Model

`CheckbookInterface` inherits an instance-creation method `new` from the `ApplicationModel` class. This inherited method is like the one you implemented for `Checkbook`—it creates a new instance and then sends an `initialize` message to this instance.

Thus, when a `new` message is sent to `CheckbookInterface` (as happens when you start the Checkbook application):

1. The inherited `new` method creates an instance of `CheckbookInterface` and sends it an `initialize` message.
2. The `initialize` method defined in `CheckbookInterface` sends the message `new` to `Checkbook`.
3. The `new` method defined for `Checkbook` creates an instance of `Checkbook` and sends it an `initialize` message.
4. The `initialize` method defined in `Checkbook` assigns initial values to the instance's variables.
5. The resulting initialized instance of `Checkbook` is assigned to the `checkbook` variable of the `CheckbookInterface` instance.

The resulting structure is shown in Figure 6-2:

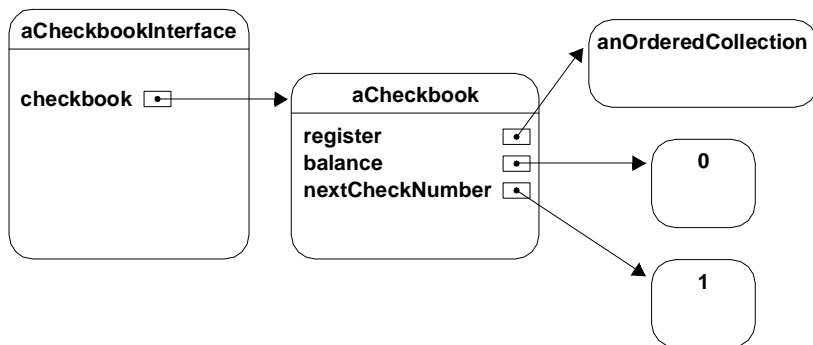


Figure 6-2 An instance of `CheckbookInterface` holding onto a `Checkbook` instance

Programming the Amount to Deposit: Field

The **Amount to Deposit**: input field on the Checkbook main window is where users enter the amount to be deposited into the application's checkbook object. Because this field is a data widget, you need to program the application model so the builder can set up the field with an appropriate value model.

Put another way, instances of `CheckbookInterface` must be able to create the required value model and make it available to the builder on request. The standard implementation is for the application model to have:

- n An instance variable that holds onto the value model
- n An accessor method for the instance variable
- n Expressions (in some method) that create the value model, tell it what data to manage, and assign it to the instance variable

The following steps show how to use the VisualWorks coding accelerator (the Definer) to generate the code for this implementation, based on the input field's property settings. Thus, to program `CheckbookInterface` for the **Amount to Deposit**: field:

1. Decide on a name for the method that will return the value model. Because the value model will manage data entered as a deposit amount, you choose the name `depositAmount`.
2. In the canvas for the Checkbook main window, select the relevant input field (the field immediately to the right of the **Amount to Deposit**: label). The Properties Tool displays the properties for the selected input field.
Hint: If necessary, choose **Arrange?Ungroup** to ungroup widgets so you can select just the field.
3. In the Properties Tool, type the name you chose in step 1 (namely, `depositAmount`) as the value of the **Aspect**: property; then click **Apply**. This associates the widget with the name of the method that returns its value model.
4. Reinstall the canvas in `windowSpec` to make the new property setting part of the interface specification.
Reminder: Click **Install...** in the Canvas Tool.
5. With the input field still selected in the canvas, generate the supporting code for it by clicking **Define...** in the Canvas Tool.
A dialog box appears, as shown in Figure 6-3. Notice that it:

- n Lists the name of the instance variable and accessor method to be generated (`depositAmount`)
- n Indicates that the generated code will create a value model and initialize the instance variable with it

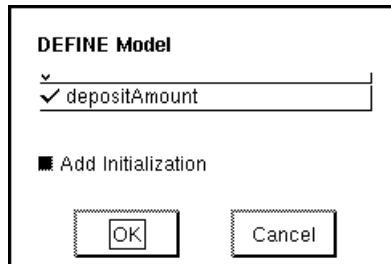


Figure 6-3 The Definer's dialog box

6. Click **OK** to generate code.
7. Refresh the System Browser by choosing **update** from <Operate> menu in the category view. A new protocol called **aspects** appears in the protocol view.
8. Examine the class definition (select **CheckbookInterface** in the class view and choose **definition** from the <Operate> menu). Notice the new instance variable **depositAmount**.
9. Select the **aspects** protocol and the **depositAmount** method. The code view displays the generated method definition:

depositAmount

"This method was generated by UIDefiner. Any edits made here may be lost whenever methods are automatically defined. The initialization provided below may have been preempted by an initialize method."

```

^depositAmount isNil
  ifTrue:
    [depositAmount := 0 asValue]
  ifFalse:
    [depositAmount]

```

This method returns the value of `depositAmount`, initializing it, if necessary, with a specialized kind of value model (a *value holder*) that holds an initial deposit amount of 0.

 **Analysis: Aspect Property**

Every data widget has an aspect property (labeled **Aspect:** in the Properties Tool). In the most general terms, this property is where you identify the *aspect* of the information model that the widget presents to the user. Because the widget must use a value model to manage its relationship to the presented information, the aspect property is effectively where you identify the widget's value model.

In more concrete terms, an aspect property associates a data widget with a message selector, which is recorded in the interface specification. When the interface is opened, the builder uses the selector to obtain a value model for the widget. That is, the builder sends the selector as a message to the application model, which must therefore have a method that returns an appropriate value model.

 **Analysis: The Definer**

TheDefiner is an accelerator for programming the application model to support both data and action widgets. You invoke the Definer by selecting one or more widgets on a canvas and then either clicking the **Define...** button on the Canvas Tool or else choosing **define...** from the <Operate> menu in the canvas. Selecting multiple widgets allows you to generate code for all of those widgets in a single operation.

For a data widget such as the **Amount to Deposit:** input field, the Definer uses information in the widget's properties to generate the standard implementation for creating and returning a value model. Thus, in this example, the Definer obtains the name `depositAmount` from the aspect property and generates:

- n The `depositAmount` instance variable
- n The `depositAmount` accessor method in the `aspects` protocol
- n Expressions in the accessor method that perform *lazy initialization* of the instance variable

Because the widget is an input field whose **Type:** property is **Number**, the generated code initializes the variable with the most basic kind of value model that a field can use and provides the number 0 as the initial data for the field to display.

Using the Definer is optional—it is a useful shortcut for typing code “by hand,” and it ensures, for a given data widget, that the application model has an appropriate method whose name matches the widget's aspect property. As

a novice VisualWorks user, you can use the Definer to learn the basic techniques for programming an application model.

As you become more experienced with VisualWorks, you may prefer different techniques for supporting data widgets. For example, you may want to write initialization code that creates a different kind of value model or you may want to put this code in an `initialize` method such as the one you created on page 135. In such cases, you can choose to:

- n Use the Definer and then modify the generated code (as you will do for the **Balance:** field).
- n Use the Definer with the **Add Initialization** checkbox deselected (this generates an accessor method containing no initialization code).
- n Simply enter all the desired code by hand.



Analysis: Lazy Initialization, Booleans, Blocks

As generated by the Definer, the accessor method `depositAmount` performs *lazy initialization* of the instance variable `depositAmount`. This means that the instance variable is initialized only when needed—the first time it is accessed. In contrast, the instance variable `checkbook` is initialized as soon as a new instance of `CheckbookInterface` is created.

The `depositAmount` method works by first testing whether the `depositAmount` instance variable is uninitialized. In Smalltalk, variables that have no other object assigned to them hold onto the undefined object `nil`. Consequently, the method determines whether the variable has the value `nil` using the following expression:

```
depositAmount isNil
```

The message `isNil` returns the object `true` if the variable's value is `nil`, and the object `false` if the variable evaluates to another object.

Booleans. The objects `true` and `false` are called *Boolean* objects. They are special Smalltalk objects that represent the answers to yes-no questions; thus, they are returned in response to querying messages such as `isNil`, which asks whether an object is the same as `nil`.

Like other objects, Boolean objects respond to a variety of messages. Among these are messages that function as conditional control structures, such as the `ifTrue:ifFalse:` message. The example uses this message to specify what to do if the

`depositAmount` variable is `nil` (initialize it) and what to do otherwise (return it).

Block Expressions. The `ifTrue:ifFalse:` message has two keywords (`ifTrue:` and `ifFalse:`), each of which has an argument that is a *block expression*. A block expression consists of one or more expressions enclosed in brackets (multiple expressions must be separated by periods). Thus, in the `depositAmount` accessor method:

- n The argument of the `ifTrue:` keyword is the block expression `[depositAmount := 0 asValue]`.
- n The argument of the `ifFalse:` keyword is the block expression `[depositAmount]`.

When a block expression is encountered, the statement(s) within the brackets are not executed immediately; rather, they are evaluated only on request. For example, when the `ifTrue:ifFalse:` message is sent to a Boolean object:

- n The object `true` responds by requesting that the first argument block be evaluated, but not the second.
- n The object `false` responds by requesting that the second argument block be evaluated, but not the first.

Thus, when the variable is uninitialized, the test expression `depositAmount isNil` evaluates to `true`, so the first argument block is evaluated. This block creates a value holder and assigns it to the variable, which is returned by the method. After the variable is initialized, the test expression evaluates to `false`, so the second argument block is evaluated. This block evaluates to the current value of the variable, which is returned by the method.



Analysis: Value Holders

The initialization code `depositAmount := 0 asValue` creates a specialized kind of value model called a *value holder*. As the term implies, a value holder holds the data it manages. That is, when a widget such as an input field accepts input from a user and sends this data to a value holder, the value holder responds by storing the data in an instance variable called `value`.

The generated code creates the value holder by sending the message `asValue` to the number `0`. Using behavior inherited from `Object`, the number `0` responds by “wrapping itself” in a new value holder—that is, by asking the framework class `ValueHolder` to create a new instance of itself with `0` as the held value. When the **Amount to Deposit:** input field is eventually set up, it will ask the value holder for the held value and display it until the user enters another value.

Like any value model, a value holder responds to `value` and `value:` messages. For a value holder, these messages access and change the value of its `value` instance variable. Thus, the **Amount to Deposit:** field sends the message `value` when it wants to get the value to display, and it sends the message `value:` to store data entered by a user.

A value holder is an appropriate kind of value model for the **Amount to Deposit:** input field. This is true because an entered deposit amount is simply a temporary piece of information that the interface must hold onto until it can be further processed. In contrast, you will see that the **Balance:** field uses a different kind of value model, because the data it presents is already held elsewhere in the application (namely, in the `balance` instance variable of a `Checkbook` instance).

Programming the Deposit Button

The **Deposit** action button on the Checkbook main window is what users click to actually deposit the amount specified in the **Amount to Deposit:** input field. Because the **Deposit** button is an action widget, you need to program the application model so it can carry out the deposit action. That is, you need to define a method in `CheckbookInterface` that obtains the entered amount from the field's value holder and passes this amount to the checkbook object for deposit. To define this method:

1. Decide on a name for the method to be defined. Because of the action it implements, you choose the name `makeDeposit`.
2. Select the action button labeled **Deposit** in the canvas for the Checkbook main window. If necessary, ungroup widgets so you can select just the button. The Properties Tool displays the properties for the selected button.
3. In the Properties Tool, type the name you chose in step 1 (namely, `makeDeposit`) as the value of the **Action:** property; then click **Apply**. This tells the button what message to send to carry out its action.
4. Reinstall the canvas in `windowSpec` to make the new property setting part of the interface specification.
5. With the action button still selected in the canvas, generate a *method stub* for it by clicking **Define...** in the Canvas Tool. (A method stub is a placeholder method you can later fill in with meaningful code.)
The Definer's dialog box appears, this time listing the selector for the method stub to be generated (`makeDeposit`).
6. Click **OK** to generate code.
7. Refresh the System Browser by choosing **update** from the <Operate> menu in the category view. A new protocol called `actions` appears in the protocol view.
8. Select the `actions` protocol and then the `makeDeposit` method. The code view displays the following:

makeDeposit

"This method stub was generated by UIDefiner"

`^self`

9. In the `makeDeposit` method definition, replace the expression `(^self)` with expressions that implement the deposit action, and choose **accept:**

```

makeDeposit
    self depositAmount value > 0
        ifFalse: [^Dialog warn: 'Enter a positive number'].
    checkbook deposit: self depositAmount value.
    self depositAmount value: 0

```

This method tests whether the user entered a valid amount in the **Amount to Deposit:** field. If so, the method asks the checkbook object to deposit the amount, and then it resets the field's display to 0; if not, the method opens a dialog advising the user to enter a positive number.



Analysis: Action Property

Every action button has an action property (labeled **Action:** in the Properties Tool). The action property associates the button with a message selector, which is recorded in the interface specification.

When the interface is opened, the builder creates the button and sets it up so that the specified message is sent when the user clicks the button. As the receiver of this message, the application model must have an appropriate method whose name matches the button's action property.



Analysis: makeDeposit Logic

The purpose of the `makeDeposit` method is to obtain, test, and, if appropriate, deposit the amount entered in the **Amount to Deposit:** field. Because the entered amount is stored in the field's value holder, the `makeDeposit` method is able to use the following expression to obtain it:

```
self depositAmount value
```

This expression sends the message `value` to the field's value holder, which is returned by the application model's `depositAmount` accessor method. The value holder responds to the `value` message by returning the data it holds. The value holder is thus a link between the field and the application model—the field sends input data to the value holder, and methods in the application model obtain this data by asking the value holder for it.

The `makeDeposit` method tests whether the entered amount is greater than 0 by using the binary message `>` (greater than):

```
self depositAmount value > 0
```

The result of this test expression is a Boolean object (`true` or `false`), to which the keyword message `ifFalse:` is sent:

```
self depositAmount value > 0
  ifFalse: [^Dialog warn: 'Enter a positive number'].
```

The response of each Boolean object to this message determines whether the argument block is evaluated:

- n If the test expression evaluates to `false` (the deposit amount is 0 or less), the argument block is evaluated. This block displays a *warning dialog* and returns from the method (notice the return character `^`). Because of the return, the remaining expressions in the method are not evaluated.
- n If the test expression evaluates to `true` (the deposit amount is greater than 0), the argument block is ignored, and the method continues by evaluating its remaining expressions.

The first of the remaining expressions actually carries out the deposit action:

```
checkbook deposit: self depositAmount value.
```

This expression obtains the entered amount from the value holder and passes it to the checkbook object held by the `checkbook` instance variable. Notice that the `deposit:` message is part of the protocol you defined in the `Checkbook` class.

The last expression resets the input field's display to 0 by changing the value in the field's value holder:

```
self depositAmount value: 0
```

As you will see, in the running application, the value holder automatically notifies the field of changes to its held value, and the field updates its display. Thus, a value holder is a way for the application model to programmatically control what a widget displays—the application model sends data to the

widget's value holder, which notifies the widget. The widget responds by obtaining the new data from the value holder for display.



Analysis: Warning Dialog

The expression in the `ifFalse:` argument block creates an instance of the class `Dialog`. This instance automatically opens a small warning dialog that displays the specified text and provides a single button labeled **OK**. The warning dialog remains displayed until the user clicks **OK**.

The `Dialog` class provides class methods for several other kinds of special-purpose dialog. For example, an expression such as `Dialog confirm: 'Do you really want to do that?'` opens a dialog box containing the specified text along with two buttons (**Yes** and **No**) for answering the displayed question.

Testing the Deposit Widgets

At this point, you can test the **Amount to Deposit:** field and the **Deposit** button in the Checkbook main window. To do this:

1. Click **Open** from the Canvas Tool to start the application.
2. Click in the **Amount to Deposit:** field to give it *keyboard focus*. This makes the field receptive to input from the keyboard and turns off the output formatting, causing the displayed amount to change from \$0.00 to 0 (input formatting).
3. Type a positive number in the **Amount to Deposit:** field. (Do not include a dollar sign or any commas).
4. Click the **Deposit** button.

Notice that:

- n The **Balance:** field does not reflect the deposit, because this widget does not yet have a value model.
 - n The **Amount to Deposit:** field is reset to \$0.00 because of the last expression in the `makeDeposit` method.
5. While the Checkbook application is still running, add the following expression to the definition of the `makeDeposit` method (insert it just above the `depositAmount value: 0` expression) and choose **accept:**

```
Transcript cr; show: 'Deposited ', self depositAmount value  
printString,  
                ' New balance ', checkbook balance printString.
```

6. Click in the **Amount to Deposit:** field and enter another positive number; then click the **Deposit** button. The System Transcript reports:
 - n The amount you deposited, obtained from the field's value holder.
 - n The current balance, obtained from the checkbook object itself. Note that the reported balance reflects both deposits you made, because the checkbook object holds onto the balance, even if the interface does not display it.
7. Click in the **Amount to Deposit:** field, enter another positive number, and press the <Return> key instead of clicking the **Deposit** button.

Because the **Deposit** button is the window's default button (recall its **Be Default:** property setting), the <Return> key:

- n Activates the **Deposit** button.

- n Keeps the keyboard focus on the **Amount to Deposit:** field so you can continue to enter input without clicking. This resets the field to 0 instead of \$0.00.
8. Delete the expression added in step 5 and choose **accept**.
9. Enter a negative number in the **Amount to Deposit:** field; then click the **Deposit** button. A warning dialog advises you to enter a positive number.
10. Click **OK** in the warning dialog. The negative amount remains displayed, with the appropriate output formatting.
11. Terminate the Checkbook application by closing its window with a window-management operation.



Analysis: Behind the Scenes During Setup

When you click **Open** on the Canvas Tool:

1. The Canvas Tool sends the `open` message to `CheckbookInterface`. This class:
 - a. Creates an instance of itself
 - b. Tells the instance to create a builder
 - c. Passes the builder the interface specification stored in the `window-Spec` class method

When initialized, the `CheckbookInterface` instance creates the `Checkbook` object it represents to the user.

2. The builder creates and sets up the various objects that form the `Checkbook` main window.

For the **Amount to Deposit:** input field, the builder:

- a. Gets the field's aspect property (`depositAmount`)
- b. Sends the `depositAmount` message to the `CheckbookInterface` instance, which responds by initializing its `depositAmount` instance variable with a value holder holding the value 0
- c. Assigns the new value holder to an instance variable in the field so that the field can send it messages
- d. Makes the field a dependent of its value holder by listing it in the value holder's `dependents` instance variable

For the **Deposit** action button, the builder:

- e. Gets the button's action property (`makeDeposit`)

- f. Sets up the button so that it responds to activation by sending the `makeDeposit` message to the `CheckbookInterface` instance

Figure 6-4 shows the portion of the resulting structure that supports the **Amount to Deposit:** field:

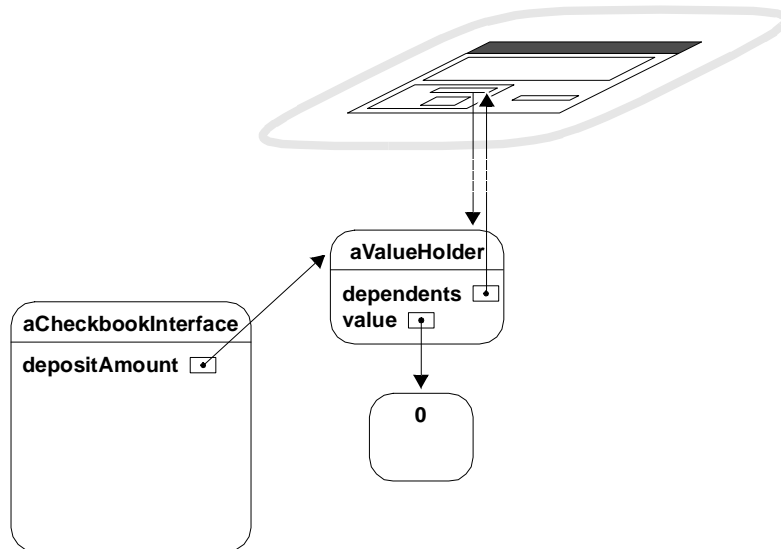


Figure 6-4 Object structure supporting the **Amount to Deposit:** field



Analysis: Behind the Scenes During Operation

When you enter a positive amount in the **Amount to Deposit:** field and click the **Deposit** button:

1. The field sends a `value:` message to put the entered amount in its value holder.
2. The button sends a `makeDeposit` message to the `CheckbookInterface` instance.
3. The `CheckbookInterface` instance responds by executing the `makeDeposit` method, which sends:
 - a. A `value` message to the value holder to get the amount
 - b. A `deposit:` message to the `Checkbook` instance to deposit the amount
 - c. A `value:` message to the value holder to reset the held amount to 0

4. The value holder responds to the `value:` message by notifying its dependents that its value has changed.
5. The field responds to notification by sending a `value` message to the value holder to obtain the new value for display.



Analysis: Widgets as Dependents

When the builder sets up a data widget with a value model, it makes the widget a dependent of the value model. This enables the value model to use the dependency mechanism to notify the widget when the relevant data changes.

In the example, the builder adds the input field to the `dependents` instance variable of the field's value holder. Then, during operation, the application model sends a `value:` message to the value holder to reset the deposit amount programmatically.

The `value:` method triggers the dependency mechanism by executing a `self changed: #value` expression, similar to the expression you entered for the `balance:` method on page 114. The value holder responds to the `changed:` message by sending change notification (a form of `update:` message) to any objects listed in the `dependents` variable. Because the field is listed there, it receives the `update:` message. Like any widget, a field responds to an `update:` message from its value model by asking the value model for the current data and then updating its display.



Analysis: Modifying a Running Application

As part of testing the application, you added some statements to the `makeDeposit` method while the application was running. In general, you can edit the definition of a method and see the effects without having to restart the application. However, if you change anything that affects an interface specification (for example, you change the canvas or a widget property), you must close the application and restart it (thereby rebuilding the interface) for the change to take effect.

Programming the Balance: Field

The **Balance:** field on the Checkbook main window displays the account balance that is stored in the application's checkbook object. Because this field is a data widget, you need to program the application model so the builder can set up the field with an appropriate value model.

As before, you use the Definer to provide `CheckbookInterface` with:

- n An instance variable that holds onto a value model
- n An accessor method for the instance variable
- n Initialization code in the accessor method that creates the value model, tells it what data to manage, and assigns it to the instance variable

However, for the **Balance:** field, you will modify the generated initialization code to create a different kind of value model than the one used for the **Amount to Deposit:** field. Instead of creating a value holder to manage the field's data, the modified code will create an *aspect adaptor*. An aspect adaptor:

- n Resembles a value holder because it responds to the same protocol and notifies its dependents of changes to the managed data
- n Differs from a value holder in that it accesses data that is held by some other object (such as a checkbook object), rather than holding the managed data itself

To program `CheckbookInterface` for the **Balance:** field:

1. Decide on a name for the method that will return the value model. Because the value model will manage data that represents the amount of the current balance, you choose the name `balanceAmount`.
2. Select the relevant input field in the canvas for the Checkbook main window (select the field immediately to the right of the **Balance:** label). Ungroup widgets if necessary. The Properties Tool displays the properties for the selected field.
3. In the Properties Tool, type the name you chose in step 1 (namely, `balanceAmount`) as the value of the **Aspect:** property; then click **Apply**. This associates the widget with the name of the method that returns its value model.
4. Reinstall the canvas in `windowSpec` to make the new property setting part of the interface specification.
5. With the input field still selected in the canvas, generate supporting code for it by clicking **Define...** in the Canvas Tool.

The Definer's dialog box appears, listing the name for the instance variable and accessor method (`balanceAmount`) and indicating that initialization code will be generated.

6. Click **OK** to generate code.
7. Refresh the System Browser by choosing **update** from the <Operate> menu in the category view.
8. Examine the class definition for `CheckbookInterface`. Notice the new instance variable `balanceAmount`.
9. Select the `aspects` protocol and then select the new `balanceAmount` method. The code view displays the following:

balanceAmount

"This method was generated by UIDefiner. Any edits made here may be lost whenever methods are automatically defined. The initialization provided below may have been preempted by an initialize method."

```

^balanceAmount isNil
  ifTrue:
    [balanceAmount := 0 asValue]
  ifFalse:
    [balanceAmount]

```

This is the standard accessor method generated for an input field; its initialization code creates a value holder.

10. In the `balanceAmount` method definition, keep the basic structure, but change the initialization code to create an aspect adaptor instead of a value holder; then choose **accept:**

```

^balanceAmount isNil
  ifTrue:
    [balanceAmount :=
      (AspectAdaptor subject: checkbook sendsUpdates:
true)
      forAspect: #balance]
  ifFalse:
    [balanceAmount]

```

Analysis: The Definer Revisited

In this section, you used the Definer to generate an accessor method, which you modified by entering code of your own. From now on, you must avoid using the Definer for the **Balance:** field, as long as its aspect property is `balanceAmount`. If you regenerate code for this selector, the Definer will overwrite your modifications with the standard accessor method.

You can either make sure the **Balance:** field is deselected in the canvas before invoking the Definer, or else you can deselect the `balanceAmount` selector in the Definer dialog box, if it appears there.

Analysis: Aspect Adaptors

The initialization code you entered creates a specialized kind of value model called an *aspect adaptor*. Like a value holder, an aspect adaptor manages a widget's access to the data it presents. However, rather than holding onto the data directly, an aspect adaptor accesses data that is held in some other object, called its *subject*. In general:

- n When a widget accepts input from a user and sends it to an aspect adaptor, the aspect adaptor responds by asking its subject to hold onto the data.
- n When a widget asks an aspect adaptor for data to display, the aspect adaptor responds by asking its subject to return the data.

An aspect adaptor effectively translates the messages sent by the widget (namely `value`, `value:`) into messages that are understood by the subject. That is, an aspect adaptor *adapts* the standard value-model protocol to match the accessor and mutator protocol for a particular *aspect* of a domain model.

Creating an Aspect Adaptor for the Balance: Field. The purpose of the **Balance:** field is to display the data held in the `balance` instance variable of the application's `Checkbook` instance. Therefore, the aspect adaptor for this field must have the `Checkbook` instance as its subject and it must translate the messages `value` and `value:` to `balance` and `balance:`, respectively. (This example exploits only the translation from `value` to `balance` because the field is read-only.)

Furthermore, the data in the `balance` variable changes with every `checkbook` transaction, and each change to the `balance` causes the `Checkbook` instance to broadcast change notification in the form of `update:` messages to its dependents (see page 114). The aspect adaptor must therefore be told that its

subject sends `update:` messages, so that it can relay the change notification to the field.

The following expressions create and initialize an aspect adaptor that is appropriate for the **Balance:** field:

```
(AspectAdaptor subject: checkbook sendsUpdates: true) forAspect:  
#balance
```

The expression within the parentheses sends a `subject:sendsUpdates:` message to the class `AspectAdaptor`. The class responds to this message by creating a new aspect adaptor that:

- n Has the application's `Checkbook` instance as its subject
- n Sets itself up as a dependent on its subject so it can respond when the subject sends `update:` messages

The outer expression then sends a `forAspect:` message to the new aspect adaptor, which responds by initializing itself with the aspect symbol `#balance`. An aspect adaptor uses its aspect symbol to:

- n Construct the accessor and mutator messages it will send to its subject. These message names are stored in the aspect adaptor's `getSelector` and `putSelector` instance variables.
- n Filter the `update:` messages it receives from its subject, so it can identify (and respond to) just those that pertain to it.

Thus, you set the aspect symbol to be `#balance` because it matches:

- n The name you gave the relevant accessor method in `Checkbook`, which is the same as the name of the mutator method except for the colon
- n The aspect symbol you gave the `changed:` message in the `Checkbook`'s `balance:` method (page 113)

Note that additional protocol is available for creating aspect adaptors when the subject's accessors, mutators, and aspect symbols have unrelated names.

Testing the Balance: Field

Now you can test the **Balance:** field to see whether it reflects deposits made to the checkbook:

1. Click **Open** from the Canvas Tool to start the application.
2. Click in the **Amount to Deposit:** field and type a positive number.
3. Click the **Deposit** button. Notice that the **Balance:** field displays the deposited amount.
4. Enter a second deposit amount. The **Balance:** field now displays the sum of the two deposited amounts.
5. Terminate the application by closing the window using the window manager.



Analysis: Setup of the Aspect Adaptor

When you run the application again, it is set up as described on page 149, with the addition of the **Balance:** field. That is:

1. **CheckbookInterface** creates an instance of itself, which creates a builder and a **Checkbook** instance.
2. The builder creates and sets up the various objects that form the **Checkbook** main window.

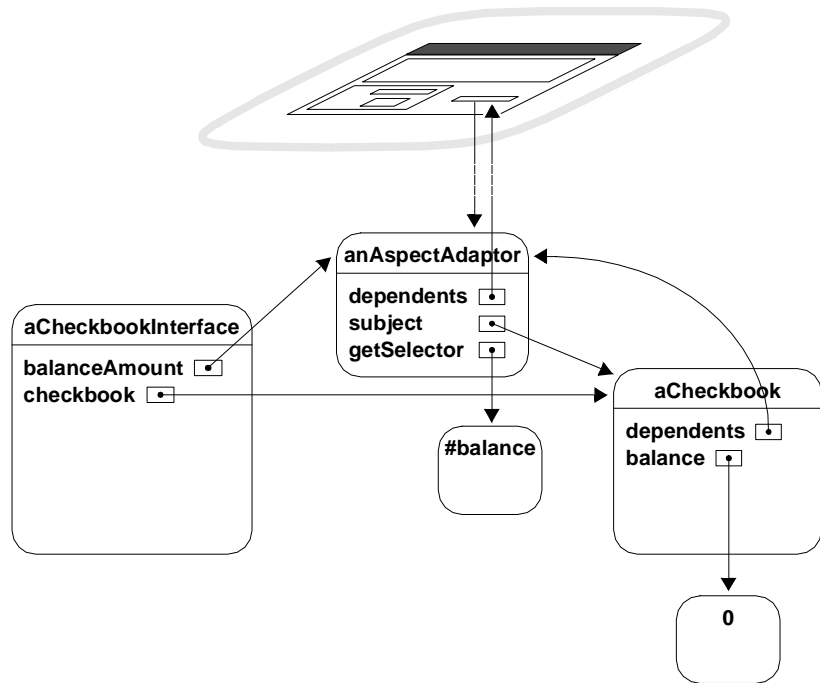
For the **Balance:** field, the builder:

- a. Gets the field's aspect property (**balanceAmount**).
- b. Sends the **balanceAmount** message to the **CheckbookInterface** instance, which responds by initializing its **balanceAmount** instance variable with an aspect adaptor.

As created, this aspect adaptor's subject is the **Checkbook** instance, its aspect symbol is **#balance**, and it is a dependent of its subject (that is, it is listed in the checkbook's **dependents** variable).

- c. Assigns the new aspect adaptor to an instance variable in the field so that the field can send it messages.
- d. Makes the field a dependent of its aspect adaptor by listing it in the aspect adaptor's **dependents** variable.

Figure 6-5 shows the portion of the resulting structure that supports the **Balance:** field:



*Figure 6-5 Object structure supporting the **Balance:** field*

Notice that this structure has two levels of dependency between the **Balance:** field and the data it displays:

- n The field is a dependent of the aspect adaptor (because widgets are set up to depend on their value models).
- n The aspect adaptor is a dependent of the Checkbook instance (because you specified `sendsUpdates: true` when you created the aspect adaptor).



Analysis: Operation of the Aspect Adaptor

During operation, the dependency mechanism works at both levels of dependency to propagate a changed balance to the field that displays it. Thus, when you enter a deposit amount:

1. The **Amount to Deposit:** field and the **Deposit** button operate as described on page 150. Among the messages sent, the **CheckbookInterface** instance sends a `deposit:` message to the **Checkbook** instance.

2. The **Checkbook** instance:
 - a. Adds the specified amount to its current balance
 - b. Assigns the new balance to its **balance** instance variable by sending itself a **balance:** message
 - c. Sends itself the **changed: #balance** message as part of executing the **balance:** method
 - d. Responds to the **changed:** message by sending the **update: #balance** message to its dependents (in this case, the aspect adaptor)
3. The aspect adaptor responds to change notification by sending an **update: #value** message to its dependents (in this case, the field).
4. The field responds to change notification by sending a **value** message to the aspect adaptor to obtain the new value.
5. The aspect adaptor responds to the **value** message by sending a **balance** message to the **Checkbook** instance and passing the returned amount to the field.
6. The field displays the new balance.

Programming the Check Register List

The **Check Register** list on the Checkbook main window displays the collection of written checks that is stored in the application's checkbook object. In addition to displaying this collection, the list enables a user to select one of the checks in it. The list must therefore keep track of two pieces of data—the collection of checks and the index of the currently selected check. Accordingly, you must program the application model so the builder can set up the list with two value models.

You use the Definer to generate the code in `CheckbookInterface` that supports the list. As with an input field, the Definer generates:

- n An instance variable
- n An accessor method for the instance variable
- n Initialization code in the accessor method that creates an appropriate object and assigns it to the instance variable

However, the initialization code generated for a list differs from that generated for an input field:

- n For an input field, the generated initialization code creates a single value holder.
- n For a list, the generated initialization code creates an auxiliary object that contains the required pair of value holders (one for the displayed collection and one for the selection index).

You then modify the generated code to initialize the appropriate value holder with the desired collection (namely, the collection of checks stored in the `Checkbook` instance). The other value holder is automatically initialized with 0, indicating there is no selection.

To program `CheckbookInterface` for the **Check Register** list, you:

1. Decide on a name for the method that will return the list's auxiliary object. Because this object will manage data that represents the current list of checks, you choose the name `checksList`.
2. Select the list in the canvas for the Checkbook main window. Ungroup widgets if necessary. The Properties Tool displays the properties for the selected list.
3. In the Properties Tool, type the name you chose in step 1 (namely, `checksList`) as the value of the **Aspect:** property; then click **Apply**. This associates the widget with the name of the method that returns its auxiliary object.

4. Reinstall the canvas in `windowSpec` to make the new property setting part of the interface specification.
5. With the list still selected in the canvas, generate supporting code for it by clicking **Define...** in the Canvas Tool.
The Definer's dialog box appears, listing the name for the instance variable and accessor method (`checksList`) and indicating that initialization code will be generated.
6. Click **OK** to generate code.
7. Refresh the System Browser by choosing **update** from the <Operate> menu in the category view.
8. Examine the class definition (select `CheckbookInterface` in the class view and choose **definition** from the <Operate> menu). Notice the new instance variable `checksList`.
9. Select the `aspects` protocol and then select the new `checksList` method. The code view displays the following:

checksList

"This method was generated by UIDefiner. Any edits made here may be lost whenever methods are automatically defined. The initialization provided below may have been preempted by an initialize method."

```
^checksList isNil
  ifTrue:
    [checksList := SelectionInList new]
  ifFalse:
    [checksList]
```

This is the standard accessor method generated for a list; its initialization code creates an instance of the framework class `SelectionInList`, which, in turn, creates two value holders.

10. In the `checksList` method definition, replace the instance-creation message `new` with the `with:` message shown below, and choose **accept**:

checksList

"This method was generated by UIDefiner. Any edits made here may be lost whenever methods are automatically defined. The initialization provided below may have been preempted by an initialize method."

```

^checksList isNil
  ifTrue:
    [checksList := SelectionInList with: checkbook register]
  ifFalse:
    [checksList]

```

The `with:` message specifies the collection for the list to display (namely, the collection of checks held by the checkbook's `register` variable).



Analysis: Setup of the List

If you run the application at this point, it is set up as described on page 156, with the addition of the **Check Register** list. That is:

1. `CheckbookInterface` creates an instance of itself, which creates a builder and an initialized `Checkbook` instance.
2. The builder creates and sets up the `Checkbook` main window. For the **Check Register** list, the builder:
 - a. Gets the list widget's aspect property (`checksList`)
 - b. Sends the `checksList` message to the `CheckbookInterface` instance, which responds by initializing its `checksList` instance variable with a `SelectionInList` instance

This instance has two value holders, one holding the checkbook's empty collection of checks and the other holding the value 0.
 - c. Assigns each value holder to an instance variable in the list widget so the list widget can send it messages
 - d. Makes the list widget a dependent of each value holder

Figure 6-6 shows the portion of the resulting structure that supports the **Check Register** list.

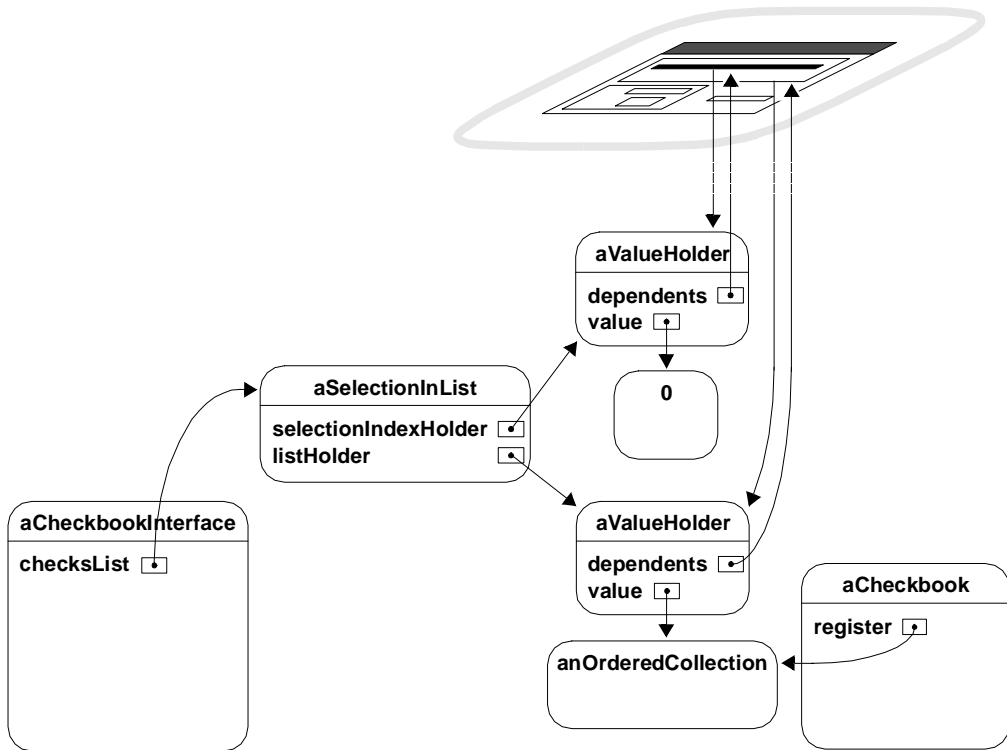


Figure 6-6 Object structure supporting the *Check Register list*



Analysis: SelectionInList Instances

The initialization code in the `checksList` method creates an instance of the framework class `SelectionInList` to support the list widget. Although a `SelectionInList` instance is not itself a kind of value model, it creates the two value models that manage the list widget's data. More specifically, a `SelectionInList` instance has:

- n A value holder held by an instance variable called `listHolder`; this value holder supplies the collection to be displayed and is initialized by the `with:` message.
- n A value holder held by an instance variable called `selectionIndexHolder`; this value holder stores the index of the user's current selection in the displayed collection.

A list widget communicates directly with each of these value holders. When the list widget needs data to display, it sends a **value** message to the appropriate value holder, which returns the collection it holds. When the user makes a selection, the list widget sends a **value:** message to the other value holder to store the new selection index.

An application model communicates indirectly with these value holders when it needs to manipulate the list widget's display programmatically. For example, an application model can change the displayed collection by sending a **list:** message to the **SelectionInList** instance. This instance, in turn, sends a **value:** message to the **listHolder** value holder. You can use the System Browser to find additional **SelectionInList** protocol.

The list widget is set up as a dependent of each of the value holders in the **SelectionInList** instance. Consequently, the list widget receives change notification from each value holder whenever that value holder receives a **value:** message.



Analysis: When the Collection Changes

When the Checkbook application starts, the **listHolder** value holder is initialized with the ordered collection of checks that is held in the **Checkbook** instance's register. The same ordered collection is held by two other objects (the value holder and the **Checkbook** instance), so it can potentially be changed through either of these objects (in fact, changes such as adding or removing checks are made only by the **Checkbook** instance).

When a **listHolder** value holder holds onto an ordered collection that will be changed by another object, extra programming is needed to trigger change notification to the widget. For example, if the **Checkbook** instance were to add a check to the collection at this point, the value holder would not be able to notify the list widget, even though it holds onto the changed collection. This is true because the value holder only sends change notification to its widget in response to receiving a **value:** message. Put another way, the value holder notices when the object it holds is reset, but it cannot detect changes that are internal to that object.

In later sections, you will provide code for updating the value holder that supports the **Check Register** list. In particular, you will program the application model to:

1. Obtain the collection of checks every time the **Checkbook** instance adds or removes a check from it.

2. Send the collection to the value holder by sending a `list:` message to the `SelectionInList` instance. This instance sends a `value:` message to the value holder, which notifies the list widget to update its display.

Note that no extra code was necessary for the value holder used by the **Amount to Deposit:** field because this value holder holds onto data that has no other storage in the application. Similarly, no extra code is necessary for an aspect adaptor, because an aspect adaptor knows to notify its widget upon receiving change notification from its subject.

Programming the Menu Bar

The menu bar on the Checkbook main window contains three menu items. A user selects:

- n The **File?Close** menu item to terminate the application and close its main window
- n The **Checks?Write...** menu item to write a new check and add it to the register
- n The **Checks?Cancel** menu item to remove a check from the checkbook register

You need to program the application model with methods that carry out the desired actions. In fact, `CheckbookInterface` already has a method for carrying out the first of these actions—namely, the `closeRequest` method inherited from `ApplicationModel`. Consequently, you need to define only two methods (one for writing checks and one for canceling them).

In this section, you program the menu bar to specify the messages that are sent to the application model when the menu items are selected. (You will implement the required methods in later sections.) To specify these messages:

1. Decide on the name of the message you want each menu item to send. For the **File?Close** item, you choose the name of the existing method, `closeRequest`. For the other two, you choose `writeNewCheck` and `cancelSelectedCheck`.
2. Select the canvas itself (click anywhere in the canvas other than on a widget). This deselects all the widgets.
3. Choose **Tools?Menu Editor** from the Canvas Tool to open the Menu Editor.
4. Click **Read** to read in the entries for the canvas's menu bar. (If the **Read** button is disabled, click in the canvas to deselect all widgets.)
5. In each of the entries for menu items, replace `nil` with the appropriate message name (use Figure 6-7 for a guide). Be sure to leave the `<Tab>` characters as is, and ignore the instances of `nil` after the menu titles:

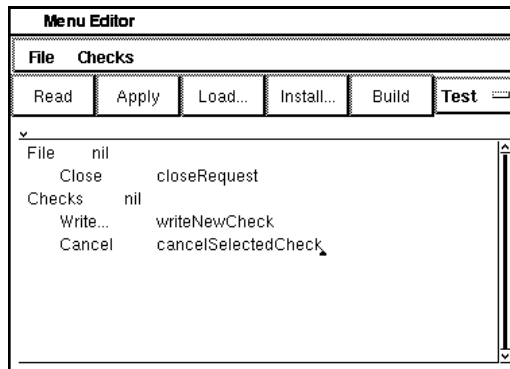


Figure 6-7 The Menu Editor with message names filled in

6. In the Menu Editor, click **Build** to generate new code for building a menu object.
7. In the Menu Editor, click **Install...** to install the menu code in the application model. A dialog appears, indicating that the code will be installed in the `menuBar` class method. Click **OK**.
Note that you do *not* need to click **Apply** because you have not changed anything that affects the canvas, such as a menu label.
8. Close the Menu Editor.
9. Test the menu bar:
 - a. In the Canvas Tool, click **Open** to start the Checkbook application.
 - b. Choose **File?Close** from the Checkbook main window to close the application.
10. Congratulations! You have finished setting the properties and editing the menu bar for the Checkbook main window. Save your image.

Setting Up for the Remaining Work

In the remaining sections, you will be setting properties for the widgets in the Check dialog box, as well as continuing to edit class and method definitions. To prepare for this work:

1. Close the canvas (and Canvas Tool) for the Checkbook main window.
2. Open the canvas for the Check dialog window:
 - a. Open a Resource Finder.
 - b. Select both the `CheckbookInterface` class and the `dialogSpec` resource.
 - c. Click the **Edit** button.
 - d. Close the Palette, but leave the Canvas Tool and the Resource Finder open.
3. Leave the System Browser and the Properties Tool open.

Providing for Writing New Checks

The **Checks?Write...** menu item in the Checkbook main window is what users choose to write a new check and add it to the checkbook's register. When chosen, this menu item sends a `writeNewCheck` message to the application model.

Consequently, you will program `Checkbookinterface` with a method called `writeNewCheck` that:

- n Creates a blank check (a new, initialized `Check` instance)
- n Opens the Check dialog box so the user can edit the blank check—that is, fill in the check's payee and amount
- n Records the completed check in the checkbook's register if the user clicks **OK** in the dialog box
- n Discards the check if the user clicks **Cancel**

In addition to creating the `writeNewCheck` method, you will also program the widgets in the interface of the Check dialog box. For example, you must provide the input fields with value models that store the user's input in the blank check.

The next three sections present the steps for incrementally defining the `writeNewCheck` method and programming the dialog box's widgets. That is, these sections will guide you through the following tasks:

1. Setting up the basic behavior for the Check dialog box. This includes both writing code and setting properties for the **OK** and **Cancel** buttons.
2. Programming the input fields in the Check dialog box.
3. Writing the code that creates the blank check and records the completed check.

These tasks are divided this way so that this tutorial can explain them separately. Note, however, that you can write method code and set widget properties in any order.

Setting Up the Check Dialog Box's Basic Behavior

A dialog box is essentially a mechanism for allowing a user to specify whether and how an application action is to proceed. Thus, a method that needs to process user-supplied information can open a dialog box to gather the required information. Such a dialog box normally contains:

- n An action widget (such as an **OK** button) that the user clicks to accept the dialog box. This in effect tells the executing method to process the gathered information.
- n An action widget (such as a **Cancel** button) that the user clicks to cancel the dialog box. This in effect tells the executing method to discard the information and return.

In the following steps, you define the first part of the `writeNewCheck` method—the part that opens the Check dialog box and determines whether the user has accepted or canceled it. You then program the **OK** and **Cancel** buttons to invoke predefined accept and cancel actions:

1. In the System Browser, select the `actions` instance protocol in the `CheckbookInterface` class.
2. In the code view, replace the method template with the following and choose **accept**:

```
writeNewCheck
|userHasAccepted|

    userHasAccepted := self openDialogInterface: #dialogSpec.
    userHasAccepted ifTrue: [
        self unimplemented]
```

The expressions in this method:

- a. Define a temporary variable, `userHasAccepted`.
- b. Open a dialog box from the interface specification stored in the `dialogSpec` class method.
- c. Assign the dialog box's result to the `userHasAccepted` variable. This result is `true` if the dialog box is accepted and `false` if the dialog box is canceled.
- d. Evaluates or ignores the argument block, depending on the value of `userHasAccepted`.

The expression in the argument block (`self unimplemented`) is a placeholder for the code you will write later to complete the check-writing action.

3. In the Check canvas, select each action button and fill in its action property as specified below; then click **Apply**. (Ungroup widgets, if necessary, so you can select each button individually.)

| Action Button | Action: property setting |
|------------------------------|--------------------------|
| Button labeled Cancel | cancel |
| Button labeled OK | accept |

4. Reinstall the canvas in `dialogSpec` to make the new property settings part of the interface specification.
5. Test the basic behavior of the dialog box:
 - a. Start the Checkbook application.

Hint: In the Resource Finder, select the `CheckbookInterface` class and the `windowSpec` resource; then click **Start**.
 - b. In the Checkbook main window, choose **Checks?Write...** This sends the `writeNewCheck` message to the `CheckbookInterface` instance, which brings up the Check dialog box. Note that the input fields are all empty, because they have no value models yet.
 - c. In the dialog box, click **Cancel**. This invokes the cancel action, which closes the dialog box and causes the `openDialogInterface:` expression to return the value `false`. The `writeNewCheck` method terminates because there are no further expressions to evaluate.
 - d. Choose **Checks?Write** to invoke the `writeNewCheck` method again.
 - e. In the dialog box, click **OK**. This invokes the accept action, which closes the dialog box and causes the `openDialogInterface:` expression to return the value `true`. As a result, the `writeNewCheck` method evaluates the expression `self unimplemented`, which opens an error notifier.
 - f. Click **Terminate** in the error notifier.
 - g. Close the Checkbook application.

 **Analysis: Actions for OK and Cancel Buttons**

The action property settings **accept** and **cancel** cause the **OK** and **Cancel** buttons to invoke **accept** and **cancel** actions that are predefined by the dialog box (see below). These actions close the dialog box and determine the result (**true** or **false**) that is returned by the expression containing the **openDialogInterface: message**. Because these actions are predefined, you do not have to create corresponding **accept** and **cancel** methods in **CheckbookInterface**. In fact, if you do create such methods, they will be ignored.

If, however, the dialog box had other action widgets, you would have to program them as you did the **Deposit** button on the Checkbook main window—by filling in their action properties and defining the corresponding methods in **CheckbookInterface**.

 **Analysis: Setup of the Dialog Box**

The **openDialogInterface: message** is part of the interface-opening protocol that **CheckbookInterface** inherits from **ApplicationModel**. When a **CheckbookInterface** instance receives this message, it creates an instance of the framework class **SimpleDialog** and tells this instance to create a window from the interface specification stored in **dialogSpec**.

Because **SimpleDialog** is a subclass of **ApplicationModel**, the **SimpleDialog** instance creates its own builder, which, in turn, creates and sets up the Check dialog box's internal structure, including its widgets. This builder opens the dialog box's interface in a *modal* window, which means that a user can invoke no operation in any other VisualWorks window until the dialog box is closed (for example, by being accepted or canceled).

Because the **SimpleDialog** instance is created as a result of an **openDialogInterface: message**, it initializes its builder to recognize the application model (that is, the **CheckbookInterface** instance) as its *source*. This means that the dialog's builder asks the **CheckbookInterface** instance to supply any value models required for setting up data widgets. (In a later section, you will program **CheckbookInterface** accordingly.)

Similarly, action widgets send their messages to the builder's source, unless their action properties are set to **accept** or **cancel**. In this case, the dialog's builder sets up the widgets so that the **SimpleDialog** instance carries out the **accept** and **cancel** actions.

Figure 6-8 gives a general idea of the objects that set up the Checkbook main window and the Check dialog box:

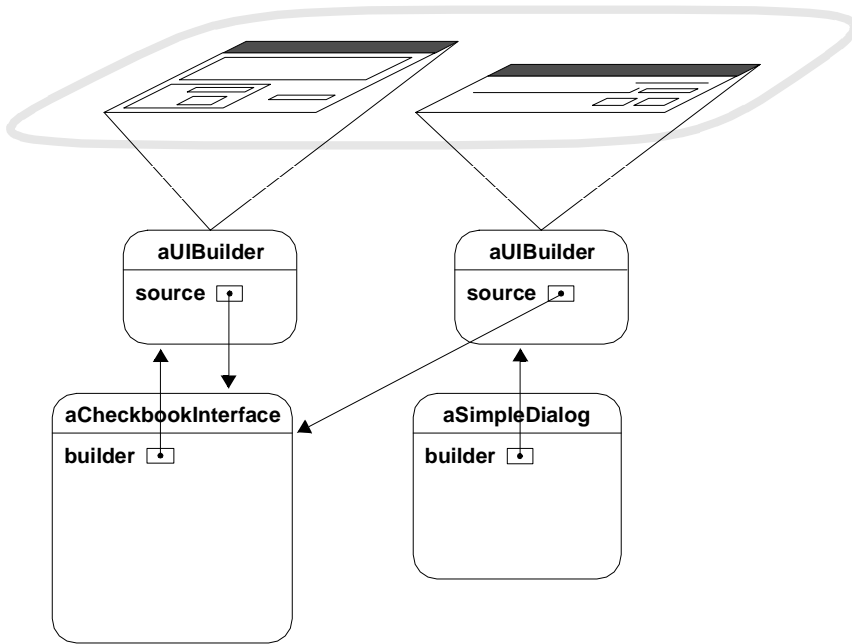


Figure 6-8 After opening the Check dialog box

Programming the Input Fields in the Check Dialog Box

The Check dialog box is where application users enter the data that makes up a new check. More specifically, the Check dialog box presents a blank check that users can edit by entering data in the appropriate input fields. Consequently, you need to program `CheckbookInterface` so that the dialog's builder can set up these input fields with appropriate value models.

Each of the input fields in the Check dialog box presents some aspect of a particular check object: its date, number, amount, or payee. Therefore, you can program these input fields by setting them up with aspect adaptors—for example, by programming `CheckbookInterface` so that its instances:

- n Create a new `Check` instance when the dialog box is opened
- n Create, for each input field, an aspect adaptor whose subject is the new `Check` instance and whose aspect symbol is the name of the `Check`'s accessor for the relevant data (`#date`, `#payee`, `#amount`, or `#number`)

One way to accomplish this is to use the basic technique you used for the **Balance:** field. That is, you can (1) write code that assigns the `Check` instance to an instance variable (as you did for the `Checkbook` instance), (2) use the Definer to generate an instance variable and an accessor method for each input field, and then (3) modify each accessor method to initialize the relevant instance variable with an appropriate aspect adaptor.

Aspect Paths. As an alternative to this basic technique, you can take advantage of a shortcut technique that employs *aspect paths*. An aspect path is a way of filling in a widget's aspect property that causes the builder to create not only the widget, but also its aspect adaptor. This means you do not need to write code in `CheckbookInterface` to create each aspect adaptor.

Rather, you program `CheckbookInterface` to provide a *subject channel* for the builder to use when it creates the aspect adaptors. A subject channel is simply a value holder that holds onto a subject for one or more aspect adaptors. In this case, all four aspect adaptors are to be created with a subject channel that holds onto a `Check` instance.

In the steps that follow, you will specify the appropriate aspect path for each input field, use the Definer to generate the code for creating a subject channel, and then add the code that puts a `Check` instance into the value holder:

1. Decide on a name for the method that is to create and return the subject channel; you choose the name `checkHolder`.

2. Select each input field in the Check canvas and fill in its aspect property with the aspect path specified below; then click **Apply**. (Ungroup widgets, if necessary.)

| Input Field | Aspect: property |
|---|-------------------------|
| Date field (upper-right corner of the canvas) | checkHolder date |
| Payee field (next to the Pay to the Order of labels) | checkHolder payee |
| Amount field (next to the payee field) | checkHolder amount |
| Check number field (next to the Check number: label) | checkHolder number |

Notice that all four aspect paths have the same *head* (namely, **checkHolder**) because all four aspect adaptors are to share the same subject channel.

3. Reinstall the canvas in `dialogSpec` to make the new property settings part of the interface specification.
4. With any of the input fields selected in the canvas, click **Define...** in the Canvas Tool.

The Definer's dialog box appears, listing the name of the instance variable and method to be generated (**checkHolder**).

5. Click **OK** to generate code.
6. Refresh the System Browser by choosing **update** from the <Operate> menu in the category view.
7. Examine the class definition for **CheckbookInterface**. Notice the new instance variable **checkHolder**.

The Definer creates an instance variable only for the head of an aspect path; because all four paths have the same head, only one instance variable is created.

8. Select the `aspects` protocol and then select the new `checkHolder` method. The code view displays the following:

checkHolder

"This method was generated by UIDefiner. Any edits made here may be lost whenever methods are automatically defined. The initialization provided below may have been preempted by an initialize method."

```

^checkHolder isNil
  ifTrue:
    [checkHolder := nil asValue]
  ifFalse:
    [checkHolder]

```

This method accesses the `checkHolder` variable, initializing it if necessary with an empty value holder. The builder will use this value holder as the subject channel for the aspect adaptors it creates.

9. Select the `writeNewCheck` method in the `actions` protocol and add the expression indicated below by bold type; then choose **accept**:

```

writeNewCheck
  |userHasAccepted|
  self checkHolder value: checkbook makeNewCheck.
  userHasAccepted := self openDialogInterface: #dialogSpec.
  userHasAccepted ifTrue: [
    self unimplemented]

```

This expression asks the `Checkbook` instance to create a new check and then puts this check in the subject channel (value holder assigned to the `checkHolder` instance variable).

Note that you put this expression in the `writeNewCheck` method because you want a new blank check to be created every time the dialog box is opened.

10. Test the dialog box's input fields:
 - a. Start the `Checkbook` application from the Resource Finder.

- b. In the Checkbook main window, choose **Checks?Write...** to bring up the Check dialog box. Notice that the input fields display the values of an initialized check:
 - The date field contains the current date.
 - The amount field contains \$0.00.
 - The **Check number:** field contains the number 1.
- c. In the dialog box, click **Cancel**. You may leave the application running.



Analysis: Aspect Paths

In previous sections, you specified aspect property settings that contain a single element, such as `balanceAmount`. A single-name setting is essentially a message for the builder to send to *obtain* a value model for a data widget. In this section, you entered a multielement *aspect path* in the aspect property of each input field in the dialog box. When the builder encounters an aspect path, it *creates* the required value model, rather than obtaining it from the application model.

More specifically, the builder uses an aspect path's elements to create an aspect adaptor. For example, when the builder encounters the aspect path `checkHolder amount`, it:

- n Sends the path's head (`checkHolder`) as a message to the application model to obtain the subject channel for the aspect adaptor
- n Uses the subsequent element (`amount`) to initialize the aspect adaptor with an aspect symbol

The resulting aspect adaptor obtains its subject from the subject channel and then responds to `value` and `value:` messages by sending its subject `amount` and `amount:` messages.

In general, when you specify an aspect path, its head must correspond to a method in the application model that returns a suitable subject channel for the resulting aspect adaptor. The element following the head must correspond to an appropriate accessor message that is defined for the object held by the subject channel.



Analysis: Setup of an Aspect Path

When you run the Checkbook application and choose the **Checks?Write** command:

1. The `CheckbookInterface` instance responds to the `writeNewCheck` message by:
 - a. Sending itself the `checkHolder` message to access the value holder in the `checkHolder` instance variable. (This value holder is created the first time the variable is accessed.)
 - b. Asking the `Checkbook` instance to create a new `Check` instance, which is placed in the `checkHolder` value holder.
 - c. Creating a `SimpleDialog` instance and passing it the `dialogSpec` interface specification.
2. The `SimpleDialog` instance creates a builder, which, among other things, builds the dialog box's input fields and their aspect adaptors. For example, the dialog's builder creates and sets up the amount field by:
 - a. Obtaining the aspect path `checkHolder amount` from the interface specification.
 - b. Sending a `checkHolder` message to the `CheckbookInterface` instance, which returns the value holder that contains the check.
 - c. Creating an aspect adaptor whose subject channel is the returned value holder and whose aspect symbol is `#amount`. This information provides the aspect adaptor with its subject (the `Check` instance) and its accessor and mutator messages (`amount` and `amount:`).
 - d. Assigning the new aspect adaptor to an instance variable in the input field so the input field can send it messages.
 - e. Making the input field a dependent of its aspect adaptor.
3. The input fields obtain the initial data to display by sending a `value` message to their respective aspect adaptors. Each of these responds by sending its accessor message to the `Check` instance and passes the relevant data to the input field.
4. When a user enters data into one of the input fields, the field sends a `value:` message to its aspect adaptor. In response, the aspect adaptor sends its mutator message to the `Check` instance.

Figure 6-9 shows the portion of the resulting structure that supports the amount field:

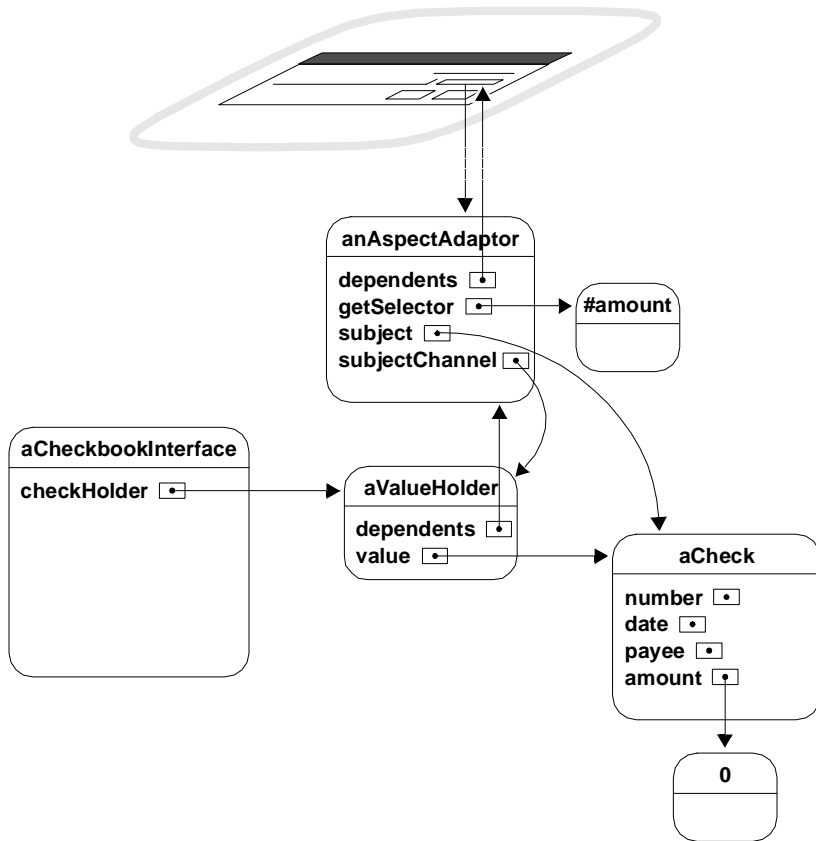


Figure 6-9 Object structure for the dialog box's amount field



Analysis: Subject Channels

In general, when you create an aspect adaptor, you can:

- n Specify its subject directly (as you did for the **Balance:** field's aspect adaptor)
- n Set it up with a subject channel—a value model (typically a value holder) from which the aspect adaptor obtains its subject

Aspect adaptors built from aspect paths are always set up with subject channels.

The advantage of using a subject channel is that it makes it easier to programmatically introduce a new subject for one or more existing aspect adaptors.

This advantage is not exploited in the tutorial example, because each opened Check dialog box displays only a single check (the dialog box and its aspect adaptors are rebuilt whenever a new check is needed). If, however, you wanted the same open Check dialog to display a series of checks, you could accomplish this by changing the contents of the subject channel.

To see why this works, notice from Figure 6-9 that an aspect adaptor is a dependent of its subject channel. When the contents of the subject channel change (as the result of a `value:` message), the subject channel, as a value holder, notifies its dependent aspect adaptor, which responds by obtaining the new subject and notifying its widget. The widget then asks its aspect adaptor to obtain the relevant data from the new subject, so it can update its display.



Analysis: Advantages of Aspect Paths

Aspect paths are especially useful whenever multiple widgets are to present different aspects of the same object. Using aspect paths in this case reduces the amount of code in the application model—instead of having a separate instance variable and accessor method to deliver each aspect adaptor to the builder, the application model has just the code required to deliver a suitable subject channel. Figure 6-10 shows the aspect adaptors (shaded in gray) that are built from the aspect paths in the tutorial example.

Although the tutorial example uses aspect paths for the widgets in a dialog box, they are not limited to this context—they can be used in main windows as well. In fact the VisualWorks database tools provide aspect paths for the widgets they generate. Aspect paths have a general syntax that you can use to generate aspect adaptors (and various other kinds of value models) for accessing data in very complex structures. For more information, see “Using an Aspect Path,” in the *VisualWorks’ Database Tools Tutorial and Cookbook*.

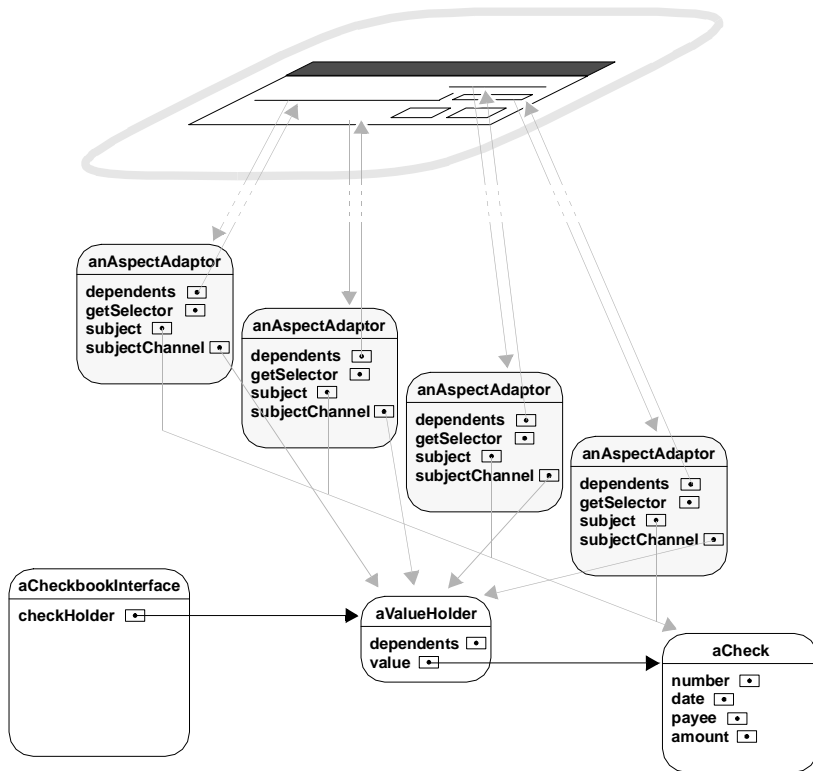


Figure 6-10 Aspect adaptors built from aspect paths

Analysis: Limitations of Aspect Paths

Aspect paths are best suited for widgets that simply need to display initial values and then accept user input. This is true because the aspect adaptors created by the builder are *not* set up to depend on their subjects, so they do not receive any change notification that their subjects may send.

For widgets such as the **Balance:** field, which presents data that is changed programmatically, you can set up the aspect adaptor to depend on its subject by creating the aspect adaptor explicitly with a `subject:sendsUpdates:` (or `subjectChannel:sendsUpdates:`) message.

Finishing the writeNewCheck Method

In this section, you finish the `writeNewCheck` method by defining how it responds when the Check dialog box is accepted by the user:

1. Make sure that the `writeNewCheck` method is still selected in the System Browser.
2. In the code view, replace the expression `self unimplemented` with the expressions indicated below by bold type; then choose **accept**:

```
writeNewCheck
|userHasAccepted|
self checkHolder value: checkbook makeNewCheck.
userHasAccepted := self openDialogInterface: #dialogSpec.
userHasAccepted ifTrue: [
    checkbook recordCheck: self checkHolder value.
    self checksList list: checkbook register]

```

The expressions you added:

- n Obtain the edited check from the `checkHolder` value holder
 - n Ask the checkbook to record the edited check in the register
 - n Inform the relevant value holder in the `SelectionInList` instance that the register has changed, so that this value holder can notify the list widget (see page 163)
3. Test the completed `writeNewCheck` method:
 - a. If necessary, start the Checkbook application from the Resource Finder.
 - b. In the Checkbook main window, choose **Checks?Write...**
 - c. Write a generous check to a deserving party. (Press <Tab> to shift the keyboard focus among input fields.)
 - d. Click **OK**. Notice that:
 - An entry for the check appears in the **Check Register** list. This entry is printed in the format you specified in the `Check's printOn:` method.
 - The **Balance:** field displays the negative balance.
 You may leave the application running.

Providing for Check Cancellation

The **Checks?Cancel** menu item in the Checkbook main window is what users choose to cancel a selected check from the checkbook's register. When chosen, this menu item sends a `cancelSelectedCheck` message to the application model. You program `CheckbookInterface` with a `cancelSelectedCheck` method as follows:

1. Make sure that the `actions` protocol of the `CheckbookInterface` class is selected.
2. Replace the current contents of the code view with the following method definition and choose **accept**:

```
cancelSelectedCheck
  self checksList selection isNil
    ifTrue: [^Dialog warn: 'Select a check to cancel.'].
  checkbook cancelCheck: self checksList selection.
  self checksList list: checkbook register
```

The expressions in this method:

- n Obtain the object that is held at the current selection index in the list widget's collection (note the use of the `selection` message from the `SelectionInList` protocol)
 - n Test whether the object is `nil`—that is, whether any check is currently selected
 - n If no check is selected, display a warning dialog and return from the method
 - n Otherwise, ask the `Checkbook` instance to remove the selected check from its register, and inform the relevant value holder in the `SelectionInList` instance that the register has changed
3. Test the **Checks?Cancel** menu item:
 - a. Restart the Checkbook application, if necessary.
 - b. Add a check.
 - c. Try canceling the check without selecting it.
 - d. Select and cancel the check.

What's Next?

Congratulations! You have completed the Checkbook application. At this point, you can either:

- n Keep the code you wrote in your image; be sure to save your image
- n Archive the code you wrote by filing out the **Examples-VWTutorial** category and then delete the category from your image

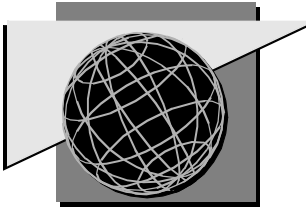
If you choose to keep the Checkbook application code, you can use it for further exploration. For example, with the help of the *VisualWorks Cookbook*, you can try:

- n Changing the tab order of the widgets in the Check dialog box
- n Disabling the **Checks?Cancel** menu item when no check is currently selected
- n Replacing the list widget in the Checkbook main window with a table or dataset that displays check information in four columns
- n Providing a new, redesigned application interface by creating new interface specifications and programming a new application model

Besides reading through topics in the *VisualWorks Cookbook*, you can browse the *VisualWorks User's Guide* for more information about:

- n Smalltalk classes you can use in your programs
- n Tools such as the Debugger
- n The VisualWorks application framework

Finally, see the *VisualWorks' Database Tools Tutorial and Cookbook* for information about developing applications that interact with databases.



Appendix A

Glossary

This glossary defines the main VisualWorks and Smalltalk terms that were introduced in this tutorial. Within a definition, terms that appear in italic type are also defined in this glossary.

accessing method A *method* that is either an *accessor* or a *mutator*; a way of referring to operations whose purpose is to either get or set the *value* of a *variable*.

accessor An *accessing method* that gets, or *returns*, the *value* of a *variable*. See also *mutator*.

action widget A *widget* that enables a user to invoke an application's action. Action widgets include action buttons and menu items. Action widgets are designed to ask an *application model* to carry out their actions.

application A complete program that enables users to define, process, store, and/or retrieve data in various ways. Applications help to automate various aspects of operation in some *domain*. Typical applications include word-processing systems, spreadsheets, calculators, and payroll systems.

VisualWorks applications are composite, in that they can be composed of other applications. Consequently, the term “application” may refer to a single *application model* (plus its associated *user interface* and *domain models*) or to a combination of multiple interacting application models.

application framework A set of *classes* that provide a core structure from which to build a complete *application*.

The VisualWorks application framework includes the `ApplicationModel` class and its *subclasses*, the `UIBuilder` class, policy classes for various platform “look-and-feels,” and classes for the various *widgets* and *value models*.

application model A *model* in a Smalltalk program that provides *application-specific* information and services. At a minimum, an application model provides the code required to support the mechanics of the *graphical user interface*. For example, an application model establishes the connections between *widgets* and *domain models*, and it defines the interactions between widgets.

Application models are usually created from the VisualWorks *application framework*. An application model refers to a *subclass* of the *Application-Model class* or to an *instance* of such a subclass.

argument An *object* that specifies additional information for an operation. Arguments are specified as *expressions* in *binary* or *keyword messages*.

aspect adaptor A kind of *value model* that accesses a *value* held in another *object*, called its *subject*. An aspect adaptor responds to *value* and *value: messages* by sending appropriate *accessor* and *mutator* messages to its subject. An aspect adaptor is an *instance* of the *class Aspect-Adaptor*, which is a *subclass* of *ValueModel*.

aspect path A way of filling in a *data widget's aspect property* to cause the *builder* to create an *aspect adaptor* for the widget. An aspect path contains multiple elements, in which the first element (the head) refers to a *subject channel* for the aspect adaptor, and subsequent elements specify *accessor* names.

assignment An *expression* that makes a change to a *variable's value*—for example *quantity := 19*.

binary message A *message* that has one *argument* and whose *selector* is made up of one or two special characters. For example, in the *message expression* *3+4*, the binary message is *+4*, where *+* is the selector and *4* is the argument.

block expression A description of a deferred sequence of actions. Block expressions consist of one or more *expressions* enclosed in square brackets.

Boolean objects The Smalltalk *objects* *true* and *false*, which serve as the answers to yes-no questions and which respond to *messages* that request logical operations and conditional control structures (if-then-else operations).

browser A *window* that displays portions of the Smalltalk class library for viewing or editing. A browser displays its information in multiple *views*.

builder An *object* that builds an operating *window* from an *interface specification*. A builder is created by an *application model* when the *application* needs to open a window. The builder, in turn, creates and assembles appropriate *user-interface objects* according to the contents of the interface specification and a specified *look policy*.

A builder holds onto the user-interface objects it creates. Consequently, an application can send *messages* to a builder to obtain programmatic access to a given *widget* or to the window itself.

Builders are *instances* of the class `UIBuilder`, which is part of the VisualWorks *application framework*.

canvas A special work area in which you *paint* the contents and layout of a *window* (or part of a window) for an *application*. You also affect the appearance of a canvas by setting *properties*. A canvas is the graphical form of an *interface specification*.

Canvas Tool The VisualWorks tool for fine-tuning a *canvas*'s appearance and for invoking additional canvas-preparation tools. A Canvas Tool is automatically opened when you open a canvas.

cascaded messages Multiple *messages* sent to one *object* in a single *message expression*. A cascaded message expression consists of one description of the *receiver* followed by several messages separated by semicolons. For example:

```
OrderedCollection new add: 1; add: 2; add: 3
```

results in three `add:` messages being sent to the result of `OrderedCollection new`.

category A group of *classes*. Every class in the system belongs to exactly one category. Classes are grouped into categories purely for organizational purposes; all classes in all categories are globally available. See also *protocol*.

Change List The VisualWorks tool that displays the changes stored in a *changes file*. To open a Change List, choose **Changes?Open Change List** from the *VisualWorks main window*.

changes file A file that lists all the changes made to the *ParcPlace Small-talk system* in your *image(1)*. The changes file is located in the same directory as the corresponding image file and has the file extension `.cha`. You view your changes file using the *Change List*.

class A description of a group of similar *objects*. A class serves as a “template” for defining the data and operations for these objects, which are its *instances*. A class defines:

- n The *instance variables* in which the instances store their data
- n The *instance methods* that describe how instances carry out their operations

Every class is itself a kind of object and therefore has its own data (*class variables*) and operations (*class methods*). One of the primary operations of a class is to create the objects that are its instances.

class hierarchy The structure formed by the *inheritance* relationships among *classes*. The hierarchy of classes is rooted in the class **Object**, which defines the state and behavior common to all *objects* in the system. **Object** does not inherit from any other class.

class method A *method* that defines a particular operation that is carried out by a *class*, such as creating an *instance* of itself. Class methods are invoked by sending *messages* to a class rather than to one of its instances.

class variable A *variable* that is shared by a *class* and all its *instances*. Class variables maintain information that is the same for all instances.

component See *widget*.

controller An *object* in a Smalltalk program that enables the user to interact with information displayed by a *view*. Together, view-controller pairs form *user-interface objects* such as *widgets*. A controller manages a *widget*'s response to mouse or keyboard input. See also *MVC architecture*.

data widget A *widget* that displays some aspect of an *application*'s data and/or collects it from the user. Data widgets include input fields, lists, datasets, and so on. Data widgets are designed to use *value models* to manage their access to the data they present.

Definer The VisualWorks tool for generating Smalltalk code that supports *widgets*. The code is generated in an *application model*. You invoke the Definer from a *Canvas Tool*.

dependency mechanism A widely used technique for coordinating the activities of different *objects* in an *application*, whereby one object, usually an *instance* of a *subclass* of **Model**, maintains a list of objects that depend on it for information and notifies these objects whenever the relevant information changes. See also *model*.

domain The area of endeavor that an *application* helps to automate—for example, employee payroll, billing, inventory control, accounting, and so on.

domain model A *model* in a Smalltalk program that defines data and operations that are relevant to the *application's domain*. For example, an accounting application might include domain models such as Customer, Account, and so on. Domain models are generally kept free of user-interface code, so that they can be reused with other interfaces.

expression A sequence of characters that describes an *object*, which is the *value* of the expression. See also *message expression* and *block expression*.

file in To load one or more files from a disk into the current VisualWorks *image(1)*. When Smalltalk files are filed in, any *class* and *method* definitions they contain are compiled into the *image(1)*.

File List The VisualWorks tool for interacting with your operating system's file-management facilities. You use a File List to locate and select files in your file system and then read them into your *image(1)*. To open a File List, choose **Tools?File List** in the *VisualWorks main window*.

file out To store the source code for one or more *classes*, *methods*, or *categories* in a disk file that is separate from the VisualWorks *image(1)* file. When you file out Smalltalk code, you normally append the `.st` extension to the filename. Filing out is a common means of backing up your work, preserving intermediate versions, or transferring code to another image (by *filing in*).

global variable A *variable* whose *value* can be accessed by all *objects* in the system.

graphical image A bitmapped illustration. You create graphical images using the *Image Editor*. You can use graphical images in a variety of ways—for example, as labels for action buttons. See also *mask*.

graphical user interface A *user interface* that consists of a collection of *windows* containing visual controls, or *widgets*.

In VisualWorks, a graphical user interface includes not only the windows and widgets themselves, but also various supporting *objects* that are supplied by the *application framework*. In addition, a graphical user interface includes the code that implements the widgets' application-specific behavior—the functionality that enables them to interact with *domain*

models. A VisualWorks graphical user interface is normally implemented using one or more *application models* and *interface specifications*.

Hierarchy Browser A kind of *browser* that displays the *superclasses* and *subclasses* of a particular *class*. To open a Hierarchy Browser, you choose **Browse?Class Named...** in the *VisualWorks main window* and then specify the class.

image(1) A file that stores the entire state of an individual *ParcPlace Smalltalk system*, including all the current *objects*, all the information on the screen, and any pending instructions to the system. A VisualWorks image preserves objects between VisualWorks sessions. When you start VisualWorks, the *object engine* reads the image file and restores the system to its previous state. You share information between images by *fling* source code *out* of one image and *fling* it *into* another. See also *standard image* and *working image*.

image(2) See *graphical image*.

Image Editor The VisualWorks tool for creating and modifying *graphical images*, with pixel-level control. The Image Editor replaces the Mask Editor in VisualWorks 1.0. You open the Image Editor from a *Canvas Tool*.

inheritance A mechanism whereby *classes* can make use of the *methods* and *variables* defined in all classes above them on their branch of the *class hierarchy*.

inheritance hierarchy See *class hierarchy*.

Inspector The VisualWorks tool for examining the *values* of the *variables* in an *object*. To open an Inspector, you choose **inspect** from an <Operate> menu.

install To save a *resource* (such as a painted *canvas*) in an *application model*. Installing a resource creates a *resource method*, which makes the resource available to the running *application*.

instance An individual *object* described by a *class*. An instance:

- n Has private memory consisting of *instance variables*
- n Responds to *messages* by invoking *methods* defined or inherited by its class

Every object in the Smalltalk system is an instance of a class. All instances of a given class are identical in form and behavior, although they generally hold different data in their instance variables.

instance method A *method* that describes how a particular operation is carried out by every *instance* of a *class*. See also *class method*.

instance variable A *variable* that stores data for an *instance* of a *class*. Collectively, an *object's* instance variables describe the object's data structure. Instances of the same class have the same number of instance variables with the same names; these instance variables generally store different *values* for different instances of the class. See also *class variable*.

interface See *graphical user interface*.

interface specification A symbolic description of a *window* (or part of a window) that is created when you *install* a painted *canvas* in an *application model*. An interface specification contains a description of the *widgets* you painted in the canvas, plus the *properties* you set for them. When the *application* runs, the interface specification serves as the *builder's* blueprint for constructing an operational window.

keyboard focus The state of a *widget* that enables it to receive input from the keyboard. You can move the keyboard focus among widgets by clicking them or by pressing the <Tab> key.

keyword An identifier with a trailing colon, such as `ifTrue:`. [Keywords are used in keyword messages.](#)

keyword message A *message* with one or more *arguments* whose *selector* is made up of one or more *keywords*. For example, in the following *message expression*:

```
aRunArray copyFrom: startIndex to: stopIndex
```

the selector is `copyFrom:to:` (consisting of keywords `copyFrom:` and `to:`) and the arguments are `startIndex` and `stopIndex`.

Launcher The VisualWorks 1.0 window for starting various tools. In VisualWorks 2.0, the Launcher has been replaced by the *VisualWorks main window*.

lazy initialization A technique for initializing an *instance variable*. Initialization code is put in an *accessor* so that the *variable* is initialized the first time it is accessed. This technique is used in code generated by the *Definer*.

look policy The platform-specific “look-and-feel” of an *application's* *interface*, which determines the appearance and behavior of buttons,

scroll bars, and so on. You set the look policy by selecting it in the *Settings Tool*.

mask A monochrome *graphical image*.

Mask Editor The VisualWorks 1.0 tool for editing *masks*. In VisualWorks 2.0, the Mask Editor has been replaced by the *Image Editor*.

Menu Editor VisualWorks tool for creating and editing menus. You open the Menu Editor from a *Canvas Tool*.

message A request for an *object* to carry out one of its operations. A message consists of a *selector* and possibly one or more *arguments*. See also *binary message*, *keyword message*, and *unary message*.

message category See *protocol*.

message expression A description of a *message* to a *receiver*. When a message expression is evaluated, the receiver carries out the operation requested by the message and *returns* an *object* to the sender; this object is the *value* of the message expression. The value is determined by the *method* that the message invokes. That method is found in the *class* of the receiver.

message protocol The list of *messages* to which an *object* can respond.

message selector See *selector*.

method A description of how to perform one of an *object's* operations. [This description contains a sequence of one or more *expressions*, which are evaluated when the method is executed.](#) Methods are analogous to procedures or functions in other programming languages.

A method is executed when a *message* matching its message pattern is sent to an *instance* of the *class* in which the method is found. A method determines the *value* of a *message expression*, either by explicitly specifying the object to be *returned* or by allowing a default value to be returned. See also *instance method* and *class method*.

method lookup The mechanism used to determine which *method* to execute when a *message* is sent to an *object*.

modal dialog box A dialog box that must be accepted, canceled, or closed before the user can invoke any other *application* actions.

model An *object* in a Smalltalk program that is concerned with defining and processing data. The data in a model is usually presented to users via *user-interface objects*. A typical VisualWorks *application* contains a

number of different kinds of models, including *domain models*, *application models*, and *value models*. See also *MVC architecture*.

Models are composite, in that they can be composed of other models. In particular, the term “model” may refer to a single piece of information presented by an individual *widget* or to the entire portion of the application that stores and processes information, independent of the presentation services provided by the *user interface*.

Models are generally created from *classes* in the VisualWorks *application framework*. Consequently, the term “model” also refers to any *subclass* of the Model class, or to an *instance* of such a subclass. As such, models inherit an implementation of the *dependency mechanism*.

mouse pointer See *pointer*.

mutator An *accessing method* that sets a new *value* for a *variable*. See also *accessor*.

MVC architecture The classic Smalltalk programming *method* of decomposing an *application* (or a portion of an application) into *models* (M), *views* (V), and *controllers* (C).

object A software unit that contains storage for a collection of related data plus operations for manipulating that data. Fundamental kinds of objects are *classes* and *instances* of classes.

object engine The executable program that runs the *ParcPlace Smalltalk system* on your platform; it essentially “sets in motion” the system *objects* in an *image*.

open To start an *application* by sending an *open message* to create an *instance* of an *application model*. The term “open” also means causing a *window* to display.

paint To specify the layout and contents of a *window* (or part of a window) by selecting *widgets* from a *Palette* and positioning them appropriately on a *canvas*. You can also affect the appearance of a canvas by setting *properties*.

Palette The VisualWorks tool that supplies the *widgets* you can *paint* on a *canvas*. By default, a Palette is opened automatically when you open a canvas.

ParcPlace Smalltalk language The general-purpose, *object-oriented* computer programming language that is provided by VisualWorks. The

applications you build with VisualWorks are implemented in the ParcPlace Smalltalk language, as is VisualWorks itself.

In the ParcPlace Smalltalk language, every entity is an object, and all processing is carried out as *messages* sent among the objects. Because the VisualWorks implementation of the language also provides a large set of predefined objects, the language is considered part of the *ParcPlace Smalltalk system*.

ParcPlace Smalltalk system The collection of interacting *objects* implemented in the *ParcPlace Smalltalk language*. Some of these objects provide functions that make up the VisualWorks software development system: the compiler, debugger, *browsers*, and so on. Other objects in the system exist so that you can incorporate them into your own *applications*, as, for example, when you use *classes* in the VisualWorks *application framework*. You extend this system whenever you create new objects.

pointer A graphic, usually in the shape of an arrow, that you move on the screen using a pointing device, such as a mouse, trackball, or joystick. You use a pointer to interact with *widgets* in VisualWorks *windows*.

properties Attributes of *widgets* and *windows* that define a variety of visual characteristics, such as font, color, borders, and so on. For widgets that display data, properties also indicate the nature of the data to be displayed and how that data is to be referenced by the *application*.

Properties Tool The VisualWorks tool for setting the various *properties* for individual *widgets*. Properties are displayed in a notebook containing pages of related properties. You open a Properties Tool from a *Canvas Tool*.

protocol A group of *methods* in a *class* definition. Every method in a class belongs to exactly one protocol. Methods are grouped into protocols for organizational purposes only. Also called *message category*.

receiver The *object* to which a *message* is sent in a *message expression*. The receiver is described by an *expression*. It is up to the receiver to decide how to respond to the message.

resource An *object* or description needed by the *builder* to assemble a particular *window* for a running *application*. Resources include *interface specifications (canvases)*, menu bars, *graphical images*, and database queries. An application's resources are normally stored in separate *resource methods* in an *application model*.

Resource Finder The VisualWorks tool for locating *classes* that contain *resources*. You can use a Resource Finder to start *applications* or to open individual resources for editing. To open a Resource Finder, choose **Browse?Resources** in the *VisualWorks main window*.

resource method A *method* (usually a *class method*) in an *application model* that *returns a resource*. Resource methods are normally invoked by the *builder* when an *application opens a window*.

return To communicate information back to the sender of a *message*. Whenever a *message expression* is evaluated, the *receiver* of the message always responds by returning an *object*, which becomes the *value* of the message expression. Returning a value indicates that the receiver's response to the message is complete.

The object that is returned by a receiver is determined by the *method* that is invoked by a message. The method may specify the returned object explicitly through an expression containing the return operator (^); otherwise, the default value returned is usually the receiver itself.

selector The name of a *method*; the portion of a *message* that determines which of the *receiver's* methods will be invoked.

Settings Tool The VisualWorks tool for customizing various global parameters of an *image(1)*, such as the default size, look, and behavior for VisualWorks tools. To open the Settings Tool, choose **File->Settings** in the *VisualWorks main window*.

Smalltalk See *ParcPlace Smalltalk language*.

snapshot A saved *image(1)* file. "Taking a snapshot" of an image refers to saving that image periodically.

sources file A file that contains the source text of the compiled Smalltalk *objects* in an *image(1)*. Every image consults a sources file to display *class* and *method* definitions.

standard image The *image(1)* that is delivered with the VisualWorks product. The first time you start VisualWorks, you use the standard image; thereafter, you normally do your work in your own *working image*.

subclass A *class* that inherits *variables* and *methods* from some other class (its *superclass*). A subclass is lower in the *class hierarchy* than its superclass. A subclass is generally a specialization of its superclass—its *instances* have the same kind of data and behavior as instances of the superclass, plus some of their own. A subclass may also override any of its inherited behavior by redefining inherited methods.

subject An *object* that holds onto information to be accessed by an *aspect adaptor*. Every aspect adaptor is created either with a subject or a *subject channel*.

subject channel A *value model* that holds onto a *subject* for an *aspect adaptor*. Subject channels provide a convenient mechanism for changing a subject that is shared by multiple aspect adaptors.

superclass The *class* from which *variables* and *methods* are inherited. A superclass is higher in the *class hierarchy* than its *subclasses*.

symbol A string that is guaranteed to be **unique in the system**. *Class* and *method* names are symbols. A symbol is expressed literally by prefixing it with the character # (for example, #balance).

System Browser The principal VisualWorks tool for creating and viewing *class* and *method* definitions. To open a System Browser, choose **Browse?All Classes** in the *VisualWorks main window*.

system classes The set of *classes* that come with the *ParcPlace Small-talk system*. The system classes provide the standard functionality of a programming language (arithmetic, data structures, control structures, and input/output facilities) and development environment (editor, compiler, debugger, window system, and so on).

System Transcript The display area for informational messages generated by VisualWorks or your code. By default, the System Transcript is in the area below the tool bar of the *VisualWorks main window*. To close or reopen a System Transcript, choose **Tools?System Transcript** from the VisualWorks main window.

tab chain A sequence of *widgets* (in a single window) whose properties are set so that the *application* user can move *keyboard focus* among them using the <Tab> key.

temporary variable A *variable* that provides temporary storage for a *value* referenced in one or more *expressions*, usually in a *method* definition. A temporary variable is declared between vertical bars—for example | aCheck |.

text cursor A small triangular graphic that shows where typed input will be inserted. A text cursor appears at the base of a line of text, between two characters.

UI object See *user-interface object*.

unary message A *message* without *arguments*. In a *message expression* such as `0 asValue`, the unary message is `asValue`.

user interface The means by which a user can control the behavior of an *application*; the software that handles input and output. See also *graphical user interface*.

user-interface object An *object* in a Smalltalk program that is concerned with presenting information and enabling users to interact with it. User-interface objects include *windows* and *widgets*. Each user-interface object is a complex object containing a *view* coupled with a *controller* and associated supporting objects.

value The *object* that is described by an *expression*. The value of a *variable* name is the object that is referenced by the variable. The value of a *message expression* is the object *returned* by the invoked *method*.

In discussions concerning *value models*, “value” usually refers to the object that is returned by sending the *message value* to a value model.

value holder A kind of *value model* that holds onto its *value* through an *instance variable*. A value holder is an *instance* of the class `ValueHolder`, which is a *subclass* of `ValueModel`. See also *aspect adaptor*.

value model An *object* that contains or refers to some other object (its *value*) and:

- n Responds to a standard *protocol* (the *messages value* and `value:`) for accessing the value
- n Notifies other interested objects when the value changes

Data widgets normally depend on value models to store or retrieve the data they collect or display.

Value models are created from the VisualWorks *application framework*. The term refers to a *subclass* of the class `ValueModel`, or to an *instance* of such a subclass. See also *aspect adaptor* and *value holder*.

variable A storage place within an *object* for a reference to another object. A variable’s name is an *expression* that describes the referenced object.

The *methods* in a *class* have access to different kinds of variables (see *class variable*, *global variable*, *instance variable*, and *temporary variable*). These kinds of variables differ in terms of how widely they are available (their scope) and how long they persist.

view An *object* in a Smalltalk program that displays text or graphics representing information in a *model*. A view is tightly coupled with a *controller*; together, view-controller pairs form *user-interface objects* such as *widgets*. See also *MVC architecture*.

Views are composite, in that they can be composed of other views. Consequently, the term may refer to the display of a single widget or to the portion of an entire *graphical user interface* that is devoted to displaying.

A view also refers to any of the display regions of a Smalltalk *browser*. For example, in a *System Browser*, the *category* view displays a list of categories in the system, whereas the code view displays textual lines of code.

visual component See *widget*.

VisualWorks main window The *window* that serves as the starting point for your work. The VisualWorks main window is identified by the title “VisualWorks” in its title bar, and it contains a menu bar and a tool bar for invoking VisualWorks’ main tools. Formerly known as the *Launcher*.

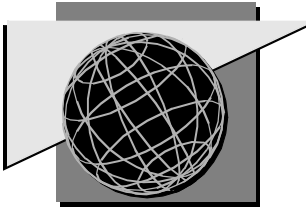
widget A control that appears in an *application’s graphical user interface*—for example, an action button, an input field, or a scrollable list. Widgets enable the application user to view information, enter information, or invoke operations. For a description of the widgets that are provided by VisualWorks, see Appendix B. See also *action widget* and *data widget*.

Each widget is a *user-interface object* that provides a characteristic display and visual response to keyboard and mouse input. Each widget consists of a *view* coupled with a *controller* and associated supporting objects. Widgets are also called *components* (or *visual components*) in some VisualWorks documentation.

window A display area on the screen that is part of an *application’s graphical user interface*. A window presents the user with information and controls for invoking operations. You create a window for an application by painting a *canvas* and *installing* it in an *application model*.

working image A copy of the *standard image* in which a user does his or her own work.

Workspace A *window* in which you can enter text and/or evaluate fragments of Smalltalk code. To open a Workspace, choose **Tools?Workspace** in the *VisualWorks main window*.



Appendix B

Widget Quick Reference

This appendix describes the various widgets available to you through the VisualWorks Palette.

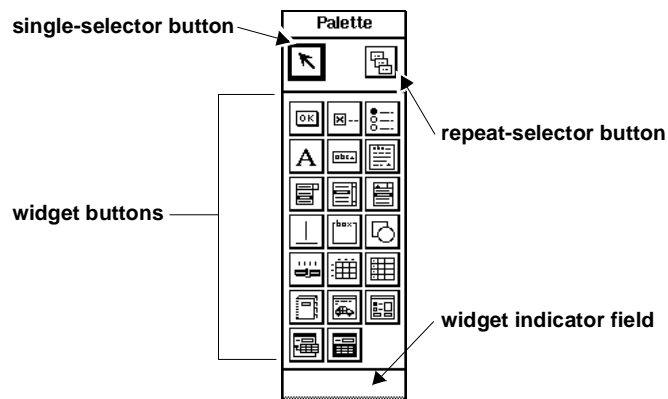


Figure B-1 The VisualWorks Palette

You can get the name of a widget by clicking on a button in the Palette. The name appears in the widget indicator field. Click the button again to deselect it.

In the definitions that follow, terms that appear in *italic type* are also defined in this quick reference. The icon shown to the left of a widget's definition appears on that widget's button in the Palette.



action button (See Figure B-2) Also called a “push button” on some platforms. Triggers a short action, such as printing, saving, deleting, or opening a dialog window. Action buttons are generally not used to set a persistent property or to set a mode.

Action buttons are convenient for the user, but they take up space in a window. When space is an issue, you can cause actions to be triggered from a menu.



check box (See Figure B-2) A toggle that enables the user to turn on or turn off some attribute in the application. For example, VisualWorks uses a check box in the Properties Tool to control widget properties such as **Can Tab**.

Check boxes are often used in a group to represent a set of related attributes. However, selecting one check box has no effect on others in the set, so you can select more than one check box at a time. When you want only one attribute to be selected at a time, use *radio buttons* instead.

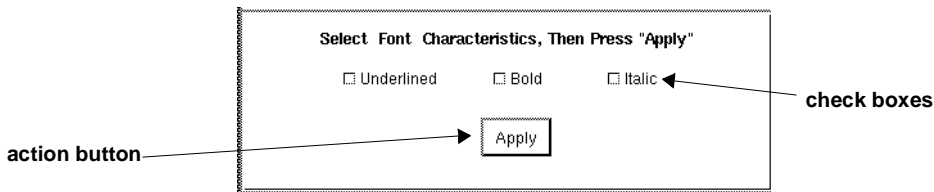


Figure B-2 An interface to modify a font



combo box (See Figure B-3) Called a “combination box” on some platforms. Provides a user-modifiable *input field* with a drop-down list of standard field entries. The application user can select from among the standard entries or fill in a nonstandard one. A combo box is similar to a *menu button*, except that a menu button does not provide an input field for nonstandard entries.

A combo box provides a customizable pop-up menu for searching and editing the data in the box. The user activates this menu using the <Operate> mouse button.

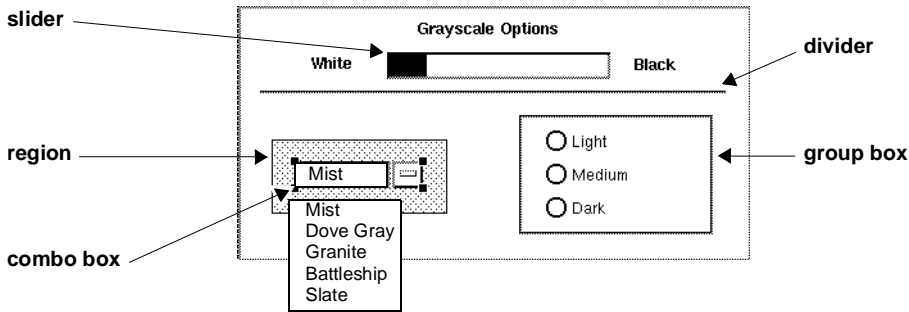


Figure B-3 Three ways to choose a gray tone



dataset (See Figure B-8) A *table* with extra features such as in-column editing, dynamic resizing of column widths (both when the dataset is being added to the interface and when the application is running), and easy reordering of columns while the dataset is being added to the interface. In addition, any column in the dataset can be specified to be read-only, an *input field*, a *combo box*, or a *check box*.

Datasets are more useful than tables when the data being presented in them is likely to be edited. Use a table when the data is unlikely to be edited, or when you want to display a possibly disparate assortment of data in a collection that allows two-dimensional access.



divider (See Figure B-3) A line segment that can be used to provide visual connection or separation between widgets. A divider is one pixel thick.



embedded data form (See Figure B-4) A special-purpose *subcanvas* used in database applications to connect a data form to the main application window or another data form. For more information about embedded data forms, see the *VisualWorks' Database Tools Tutorial and Cookbook*.

field See *input field*.



group box (See Figure B-3) A rectangle that surrounds groups of related widgets. The sides of the group box are one pixel thick. The group box optionally has a *label* embedded in its top border.



input field (See Figures B-4 and B-7) Called a “text input” or “text entry” field on some platforms. A region for presenting and/or entering data that is only one line long, such as a filename or other string.

An input field provides a customizable pop-up menu for searching and editing the input data. The user activates this menu using the <Operate> mouse button.

Note that input fields can be made read-only, so that text can be displayed but not entered.

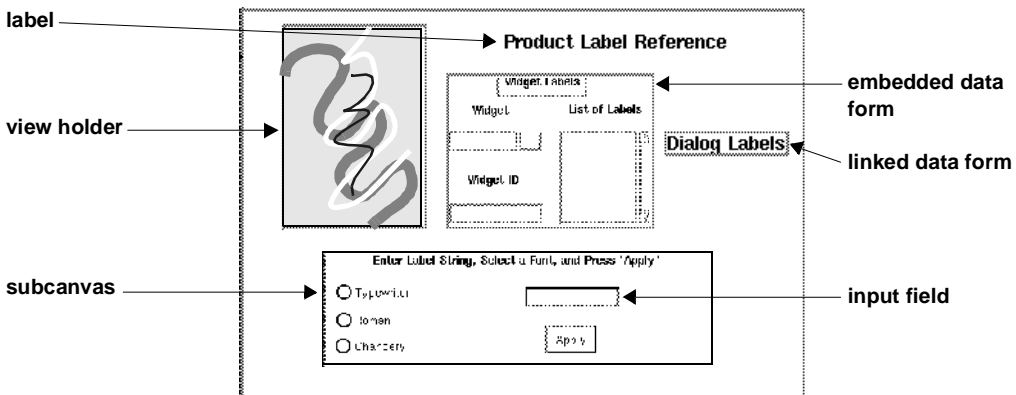


Figure B-4 Embedded interfaces



label (See Figure B-4) A single line of text or a graphic image that is typically used:

- n In conjunction with another widget, such as a field, to describe that widget’s purpose
- n As titles for groups of widgets
- n For a read-only display

For a multiline label, use a read-only *text editor*.



linked data form (See Figure B-4) A special-purpose *action button* that, when clicked, displays a data form canvas in a separate window. Linked data forms are used in database applications to connect a data form to the main application window or another data form. For more information about linked data forms, see the *VisualWorks Database Tools Tutorial and Cookbook*.



list (See Figure B-5) A list widget is useful for displaying any collection of objects. As an input device, the list enables the user to select one or more elements in the list as targets for operations such as browsing.

Lists have a built-in search ability, so the user can type the beginning letters of an item to find it in a large list (<Escape> starts a new search).

You can arrange for a list to have a custom menu that provides commands that act on the selections or commands that act on the list itself (updating or filtering its contents).

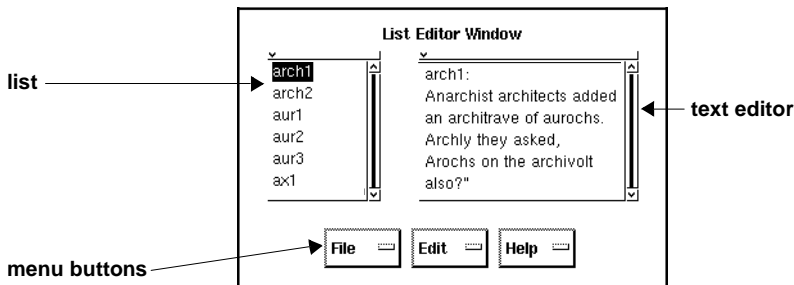


Figure B-5 A list linked to a text editor



menu button (See Figure B-5) Provides the user with a well-defined set of options. A menu button can present a menu of commands or a menu of values. It is similar to a submenu in a menu bar, with two advantages: it can be placed anywhere on the canvas, and its *label* can change to reflect the current selection. A menu button is more visible to the user than a pop-up menu, but it uses space in the canvas. The menu that a menu button invokes is a “pull-down” menu (called a “drop-down” menu on some platforms).



notebook (See Figure B-6) A means of presenting a hierarchy of information. This hierarchy may have one or two tiers, which are called the notebook’s major keys and its minor keys, respectively.

The minor keys either refine the subdivision imposed by the major keys or filter the information along a separate dimension. They may be

connected to the major keys in such a way that each major key may display a different set of minor keys.

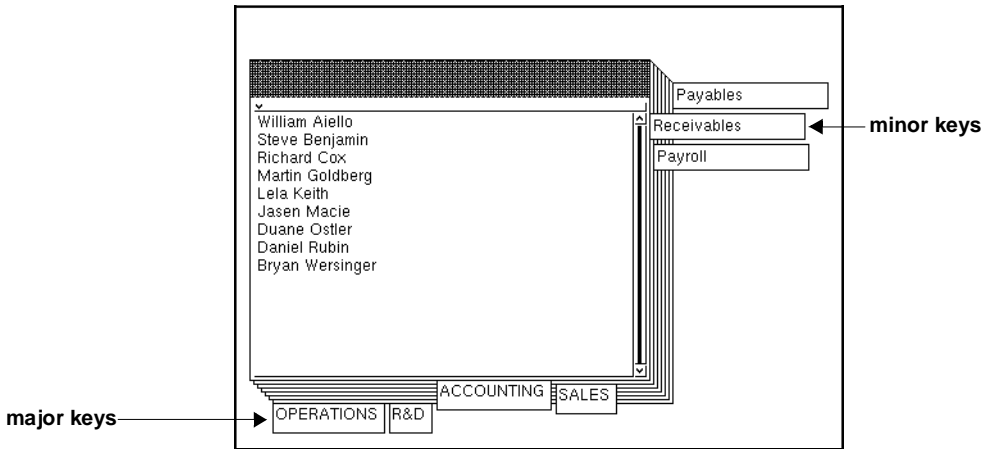


Figure B-6 A notebook with interdependent tabs



radio button (See Figure B-7) Enables the user of your application to make a single selection from a limited list of choices. Selecting a radio button causes any other button in its group to be deselected.

Radio buttons have the advantage of displaying a full set of choices at all times. However, if the list of choices is long or needs to be reconfigured dynamically, you should use a *list* widget instead of a set of radio buttons. If you want users to be able to select multiple items on the list, you should use *check boxes* (or a list that permits multiple selection).

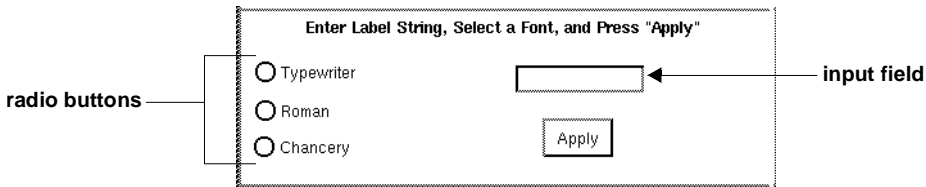


Figure B-7 Creating a label



region (See Figure B-3) A shape that surrounds a group of widgets. A region may be rectangular or elliptical. Its borders can be one of four thicknesses, and its interior can be filled with a color.



slider (See Figure B-3) A device for selecting and displaying a value from a range of values, such as the volume setting for a music program. It simulates the sliding switch found on some electronic devices, where changing the position of the switch changes the value of some property, such as volume.



subcanvas (See Figure B-4) An interface that has been included or embedded in another interface. A subcanvas can itself include another subcanvas. You can nest as many levels of interfaces as you like within the parent interface.

By using subcanvases, you can create a set of application modules that can be plugged into larger applications as needed. This approach avoids wasteful duplication of effort for generic modules, enforces uniformity of interface design, and makes changes much easier to implement, since you have to change only the core module to propagate the changes to all the applications that use that module.



table (See Figure B-8) A means of displaying data that can be organized usefully in rows and columns.

By default, a table is bordered and has both vertical and horizontal scroll bars. You can turn off any of these features in the Properties dialog box. You can also set the font to be used with text that is displayed in the table's cells, connect an <Operate> menu to the table, and turn on vertical and horizontal grid lines to separate rows and columns.

| | | |
|---------|-------|------|
| ByCo | 9.25 | 6.0 |
| Gaard | 10.50 | 8.5 |
| SteinHF | 9.75 | 5.0 |
| TryggAS | 17.00 | 11.0 |
| Vannet | 8.25 | 5.75 |

table

| | | |
|---------|-------|------|
| ■ ByCo | 9.25 | 6.0 |
| Gaard | 10.50 | 8.5 |
| SteinHF | 9.75 | 5.0 |
| TryggAS | 17.00 | 11.0 |
| Vannet | 8.25 | 5.75 |

dataset

Figure B-8 Presenting columns of information



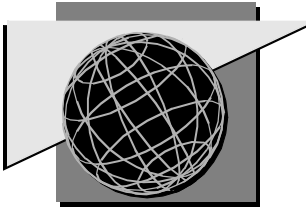
text editor (See Figure B-5) A *region* for displaying and editing text. A text editor is especially useful for text that does not fit within an *input field*, especially when it is expected to have multiple lines. A text editor

has built-in facilities for line wrapping, changing the text style, cutting, copying, pasting, searching and replacing, undoing, and optionally executing Smalltalk expressions.

A text editor provides a customizable pop-up menu for searching and editing text. The user activates this menu using the <Operate> mouse button.



view holder (See Figure B-4) A means of including a graphic image in an interface. It allows you to treat a graphic like any other widget: you can arrange its layout and apply borders and scroll bars. However, you must supply the code that connects the graphic to your application's domain model.



Index

Symbols

<Control>-click xiv
<Meta>-click xiv
<Operate> button xiii, 7
<Select> button xiii, 7, 9
<Shift>-click xiv
<Window> button xiii, 7

A

accessing method 92, 112
 creating 92–94
 defined 183
 limited access to variables 113
accessor 92, 112
 defined 183
action button 52
 creating 73–74
 defined 200
 programming 143–146
 testing 147–150
action property 59, 144
 accept and cancel settings 170
action widget 129
 defined 183
adding, *see* creating 24
aligning widgets 76
application 28
 building 39–48
 database 46
 defined 183
 finding 29
 layered structure 40–43, 131
 modifying while running 150
 multiwindow 44
 opening 70–72
 programming 127–181
 running 28
 starting 70–71
 UI-based structure 44
application framework 40
 defined 183
application model 42, 131
 browsing 134
 creating 64–66
 defined 184
 designing 48
 initializing 135–136
 programming 127–181
applying changed property 60
argument 95
 defined 184
aspect adaptor 151, 153
 creating for field 153–154
 defined 184
 operation 156–157
 setup 155–156
aspect path 172–179
 advantages 178–179
 defined 184

- limitations 179
- setup 176–177
- aspect property 59, 139
- assignment expression 119
 - defined 184
- assignment operator (:=) 96
- asterisk (*) 23, 27

B

- backup 26
- binary message 118
- block expression 141
 - defined 184
- Boolean objects 140–141
 - defined 184
- browser, defined 185
- browsing
 - application model 134
 - class hierarchy 24–25
 - inheritance hierarchy 25
 - online documentation 30–34
 - Smalltalk class library 19–25
- builder 71
 - defined 185
- bulletin boards xvii
- buttons, mouse, *see* mouse buttons

C

- canvas 63
 - creating 53–54
 - defined 185
 - finding an installed 66
 - installing 64–66
 - opening a blank canvas 53–54
 - previewing for another platform 82
 - sizing 55
- Canvas Tool 6, 54

- defined 185
- cascaded messages 98
 - defined 185
- category 20, 86
 - adding to class library 24
 - defined 185
 - locating 86
- category view 20
- Change List 36–37
 - defined 186
- change notification 113–114
- changes file 36
 - defined 186
- changes view 36
- character-based display 99
- check box, defined 200
- class 19, 83
 - commenting 91, 105
 - creating 85–125
 - creating an instance 89–90, 107
 - creating definition 87–88, 105–106
 - defined 186
 - defining data structure 87–88
 - documenting 91, 105
 - editing definition 87
 - entity class 46
 - finding by name 23
- class definition
 - creating 87–88, 105–106
 - viewing 21
- class hierarchy
 - browsing 24–25
 - defined 186
- class library
 - adding a category 24
 - browsing 19–25
- class method 109
 - creating 108–109
 - defined 186

- class variable, defined 186
- class view 20
- click xiv, 9
- closing windows 11
- code view 20
- collapsing a VisualWorks window 12
- combo box, defined 200
- comma (concatenation message) 100
- comments 91, 105
- compilation 96
- complex expressions 97–98, 117–119
- component, defined 186
- concatenation message 100
- constructing a string 100
- contents view 28
- controller, defined 186
- conventions
 - naming 96
 - screen xii
 - typographic x–xii
- copying and pasting
 - text 11
 - widgets 57
- creating
 - action button 73–74
 - application model 64–66
 - aspect adaptor 153–154
 - canvas 53–54
 - category 24
 - class 85–125
 - class definition 87–88
 - class method 108–109
 - instance 89–90, 135–136
 - instance method 92, 108–109
 - instance variable 87–88
 - menus 67–69
 - protocol 92
 - strings 100
 - windows 79–81

customizing a working image, *see* working image

D

- data forms 46
- Data Modeler 6
- data type 109
- data widget 129, 130
 - defined 187
- database applications 46
- dataset 52
 - defined 201
- Definer 139–140, 153
 - defined 187
 - dialog box 138
- deleting
 - text 11
 - widgets 58
- dependency mechanism 106, 107
 - defined 187
- deselecting
 - text 10
 - widgets 56
- dialog box, setting up basic behavior 168–171
- disk files 26–28
- displaying
 - descriptive string 99, 102
 - properties of a widget 59
- divider, defined 201
- documentation, *see* VisualWorks documentation
- domain 42
 - defined 187
- domain model 42, 131
 - defined 187
 - designing 47
 - developing 83–125
 - testing 120–125
- double-click xiv, 9

E

- editing
 - menu bar 67–69
 - template for class definition 87
 - text 8, 10
 - see also* painting
- electronic bulletin boards xvii
- electronic mail xvii
- embedded data form, defined 201
- entity class 46
- error 123–125
- evaluating Smalltalk expressions 18, 102
- exiting VisualWorks, *see* VisualWorks
- expression
 - defined 187
 - see also* message expression

F

- fax support xvii
- field, *see* input field
- file in 26, 28
 - defined 187
- File List 6, 27
 - defined 187
- file out 26
 - defined 187
- files
 - disk 26–28
- finding
 - application 29
 - category 86
 - class 23
 - installed canvas 66
- fonts x–xii
- format of output 62
- Format:** property 62

G

- global variable 122
 - defined 188
- glossary 183–198
- graphical image, defined 188
- graphical user interface 7, 39
 - creating 51–82
 - defined 188
 - designing 47
 - programming 127–181
 - programming application-specific behavior 128–130
 - specifying basic appearance and behavior 128
- group box, defined 202
- grouping widgets 77–78

H

- help file 14
- Hierarchy Browser 24–25
 - defined 188

I

- image 2
 - defined 188
 - saving 13–15
 - standard 2, 3
 - working 2
 - creating 13
 - customizing 35
 - starting
- Image Editor, defined 188
- indicator field 55
- information model 40, 43, 131
- inheritance 24
 - defined 189
- inheritance hierarchy 24
 - browsing 25

defined 189
 initialization code 151, 158
 initializing
 application model 135–136
 lazy initialization 139, 140–141
 variables 108–111
 input field 51, 58
 creating aspect adaptor for 153–154
 creating value holder for 137–138
 defined 202
 programming 137–142, 151–154, 172–179
 setting properties 62
 testing 147–150, 155–157
 inspecting
 default widget properties 61
 value of expression, *see* Inspector
 Inspector 89, 101
 defined 189
 opening 102
 installing
 canvas 64–66
 defined 189
 menu bar 69
 instance 19, 24
 creating 89–90, 135–136
 defined 189
 displaying description 101–103
 initializing variables 108–111
 message for creating 90
 instance method 19, 21, 109
 creating 92, 108–109
 defined 189
 instance variable 19, 21, 88, 92
 creating 87–88
 defined 189
 setting value 97
 interface 189
 opening 70–72
 programming 127–181

see also graphical user interface
 interface specification 64, 128
 defined 189

K

keyboard focus, defined 190
 keyword 95
 defined 190
 keyword message 95, 118
 defined 190

L

label 51, 58, 60
 defined 202
 Launcher, defined 190
 layered structure of VisualWorks
 application 40–43
 layout, adjusting window 78
 lazy initialization 139, 140–141
 defined 190
 library, *see* class library
 linked data form, defined 203
 list widget 51, 58
 defined 203
 initialization code 158
 programming 158–163
 setup 160–161
 literal string 100
 look policy, defined 190
 lookup 102–103, 110–111

M

Macintosh platforms 3, 5, 12, 13, 14, 15, 16
 mail
 electronic xvii
 main window, *see* VisualWorks, main window
 Mask Editor, defined 190

- mask, defined 190
- menu 51
- menu bar 51, 63
 - editing 67–69
 - installing 69
 - programming 164–165
- menu button, defined 203
- Menu Editor 67–69, 164–165
 - defined 190
- menus
 - creating 67–69
 - VisualWorks main window 5
 - see also* menu bar
- message 90
 - binary 118
 - defined 190
 - for creating instance 90
 - keyword 95, 118
 - sending to Smalltalk objects 18
 - transcript 122–123
 - unary 90, 118
- message category, defined 191
- message expression 18, 23, 90, 97–98
 - cascaded 98
 - complex 97–98, 117–119
 - defined 191
 - evaluating 102
 - sequences of expressions 98
- message pattern 95
- message protocol 94
 - defined 191
- message selector, defined 191
- method 19, 24, 85, 92, 108
 - class method 109
 - compilation of 96
 - creating 92, 108
 - defined 191
 - generated by the Definer 138
 - incrementally defining 167–179
 - instance 109
 - lookup 102–103, 110–111
 - testing 120–125
 - see also* accessing method
- method definition 95
- method lookup 102–103, 110–111
 - defined 191
- method stub 143
- method view 22
- modal dialog box 170
 - defined 191
- model 41, 106
 - application model 42, 131
 - designing 48
 - defined 191
 - domain model 42, 131
 - designing 47
 - developing 83–125
 - information model 40, 43, 131
 - subclass of Model class 106
 - value model 130, 131
- mouse buttons xii
 - <Operate> button xiii
 - <Select> button xiii
 - <Window> button xiii
 - functions 7
 - one-button mouse xiii
 - three-button mouse xiii
 - two-button mouse xiii
 - using 8–11
- mouse operations xiv
 - <Control>click xiv
 - <Meta>-click xiv
 - <Shift>-click xiv
 - click xiv
 - double-click xiv
- mouse pointer, defined 192
- moving selection to next widget 61
- multiwindow application 44

mutator 92, 112

defined 192

MVC architecture, defined 192

N

names view 27

naming conventions 96

new 108, 110

nil 140

notational conventions x–xii

notebook 35

defined 204

O

object 2

defined 192

Object Behavior Analysis and Design (OBA/D)

methodology 46

object engine 2

defined 192

Online Documentation Browser 6, 31–34

online documentation, *see* VisualWorks

documentation

opening

application 70–72

blank canvas 53–54

defined 192

interface 70–72

OS/2 platforms 3, 12, 13, 14, 15

output formatting 62

P

painting

defined 192

multiple copies of widget 58

properties 79–81

widget 55

Palette 54

defined 192

ParcPlace Smalltalk language 2

defined 192

ParcPlace Smalltalk system 2

defined 193

pattern view 27

pixels, spacing by 77

pointer, defined 193

positioning widgets 57

primary windows 44

printing

displaying a descriptive string 99

programming

action button 143–146

application model 127–181

data widget 151–154

graphical user interface 127–181

input field 137–142, 151–154, 172–179

list widget 158–163

menu bar 164–165

properties

action 144

applying changed 60

aspect 139

defined 193

displaying a widget's 59

inspecting the defaults 61

painting 79–81

setting 59–63, 79–81

setting input field 62

setting window 63

Properties Tool 59–60

defined 193

protocol 21

creating 92

defined 193

protocol view 21

pseudovisible 89

R

radio button, defined 204
receiver 18, 90
 defined 193
region, defined 205
repeat-painting 58
repeat-selection button 58
resizing
 canvas 55
 widgets 57
 window 12
resource 28
 defined 193
Resource Finder 6, 29
 defined 194
resource method, defined 194
retrieving information from disk files 26–28
return 90, 92
 defined 194
return operator (^) 95

S

saving
 image 13–15
 viewing changes since last save 36
screen conventions xii
secondary windows 44
selecting
 multiple widgets 74
 text 10
 widget 56
selection handle 56
selector 90
 defined 194
self 102–103
setting
 properties 59–63, 79–81
 values of instance variables 97

Settings Tool 35
 defined 194
single-selection button 55
sizing
 canvas 55
 widgets 75
slider, defined 205
Smalltalk programming language, *see* ParcPlace
 Smalltalk language
Smalltalk, defined 194
snapshot 15
 defined 194
sources file 14, 170
 defined 194
spacing by pixels 77
special symbols x–xii
standard image 2, 3
 defined 195
starting application 70–71
starting VisualWorks, *see* VisualWorks
starting working image, *see* working image
storing information in disk files 26
stream 100
string
 concatenation message 100
 constructing 100
 streams 100
subcanvas, defined 205
subclass 24, 106
 defined 195
subject 153
 defined 195
subject channel 172, 177–178
 defined 195
super 110–111
superclass 24, 106
 defined 195
support, technical xvi
 electronic bulletin boards xvii

- electronic mail xvii
- fax xvii
- telephone xvii
- World Wide Web xvii
- symbol 114
 - defined 195
- symbols used in documentation x–xii
- syntax error 123–125
- System Browser 6, 19–25
 - defined 195
 - opening 19
- system classes, defined 195
- system objects 2
- System Transcript 4, 6, 122
 - defined 195

T

- tab chain, defined 196
- table 52
 - defined 205
- technical support xvi
 - electronic mail xvii
 - electronic bulletin boards xvii
 - fax support xvii
 - telephone support xvii
 - World Wide Web xvii
- telephone support xvii
- template 87, 92
- temporary variable 98
 - defined 196
- testing
 - action button 147–150
 - domain models 120–125
 - input field 147–150, 155–157
 - methods 120–125
 - widgets 147–150
- text cursor, defined 196
- text editor, defined 206

- transcript message 122–123
- typographic conventions x–xii

U

- UI-based structure of VisualWorks application 44
- unary message 90, 118
 - defined 196
- UNIX platforms 3, 12, 13, 14, 15
- user interface 40
 - defined 196
 - designing 47
 - see also* graphical user interface
- user-interface object 41
 - defined 196

V

- value 88, 92
 - defined 196
- value holder 138, 142
 - defined 196
- value model 130, 131
 - defined 197
- variable 19, 24, 85, 92
 - defined 197
 - global 122
 - initializing 108–111
 - limited access to 113
 - temporary 98
- view 20
 - category 20
 - changes 36
 - class 20
 - code 20
 - contents 28
 - defined 197
 - method 22
 - names 27

- pattern 27
- protocol 21
- view holder, defined 206
- VisualWorks
 - application building 39–48
 - layered structure 40–43
 - UI-based structure 44
 - approach to interface programming 127–131
 - collapsing a window 12
 - defined 1
 - exiting 15
 - features 1
 - interacting with 7–13
 - main window 4, 52
 - defined 197
 - menus 5
 - managing windows 11–13
 - saving an image 13–15
 - Smalltalk environment 2
 - starting 2
 - on Macintosh platforms 3
 - on OS/2 platforms 3
 - on UNIX platforms 3
 - on Windows platforms 3
 - Workspace 4, 6
- VisualWorks documentation
 - online xv
 - browsing 30–34
 - Database Cookbook* xv, 30
 - Database Quick Start Guides* xv, 30
 - International User's Guide* xv
 - VisualWorks Cookbook* xv, 30, 33
 - printed
 - Cookbook* xiv
 - Database Connect User's Guide* xv
 - Database Tools Tutorial and Cookbook* xv
 - Installation Guide* xiv
 - International User's Guide* xv
 - Object Reference* xv

- Release Notes* xiv
- User's Guide* xiv

W

- warning dialog 145, 146
- widget 41
 - action 129
 - aligning 76
 - as a dependent 150
 - copying and pasting 57
 - creating 72–74
 - data 129, 130
 - defined 198
 - deleting 58
 - displaying properties 59
 - equalizing sizes 75
 - grouping 77–78
 - inspecting default properties 61
 - list 158–163
 - moving selection to next 61
 - painting 55
 - painting multiple copies 58
 - positioning 57
 - programming 151–154
 - quick reference 199–206
 - refining arrangement 74–78
 - resizing 57
 - selecting and deselecting 56
 - selecting multiple 74
 - testing 147–150
 - types and positions 51
- wildcard character 23, 27
- window outline 9
- window-management operations 11
- windows
 - adjusting layout 78
 - collapsing 12
 - creating 53, 79–81

- defined 198
- designing 47, 51–52
- inspecting the prototype 71
- managing 11–13
- painting and setting properties 79–81
- previewing for another platform 82
- primary 44
- resizing 12
- revising 72–79
- secondary 44
- setting properties 63
- Workspace 4, 6
 - see also* VisualWorks main window
- Windows platforms 3, 12, 13, 14, 15, 16
- working image 2
 - creating 13
 - customizing 35
 - defined 198
 - starting 17
- Workspace 4, 6
 - closingwindows
 - closing 11
 - defined 198
 - resizing 12
 - sending messages 18
- World Wide Web xvii

