Dan Shafer
Dean A. Ritz

Practical Smalltalk

Using Smalltalk/V

With 48 Illustrations

Springer-Verlag
New York Berlin Heidelberg London
Paris Tokyo Hong Kong Barcelona

Dan Shafer
Redwood City, CA 94062
USA

Dean A. Ritz
Palo Alto, CA 94301
USA

Library of Congress Cataloging-in-Publication Data
Shafer, Dan
Practical Smalltalk : using Smalltalk/V / Dan Shafer. Dean A. Ritz.
p. cm. Includes
index. ISBN 0-387-
97394-X
1. Smalltalk/V (Computer program language) I. Ritz. Dean A. H.
Title.
QA76.73.S59S53 1991
005.13'3-dc20 91-327

Printed on acid-free paper.

Typeset by Production Services, Redwood City, California.
Printed and bound by R.R. Donnelley & Sons, Harrisonburg, Virginia.
Printed in the United States of America.

9 8 7 6 5 4 3 2 1

ISBN 0-387-97394-X Springer-Verlag New York Berlin Heidelberg
ISBN 3-540-97394-X Springer-Verlag Berlin Heidelberg New York

This book is dedicated

by Dan Shafer

to

Bill Gladstone Super-

Agent and Super-Friend

and

by Dean Ritz

to his Brothers, Keith and Mike

Preface

This is a book about what we believe may well be the Computer Programming Language of the 90s: Smalltalk. No, you're not about to be subjected to yet another sales pitch for Object-Oriented Programming (OOP) and the languages that support that approach to software design and construction. If you've purchased this book, you're already sold on OOP and probably at least to some degree on Smalltalk as well.

As we looked at the available book titles about Smalltalk programming when we started this project, we noticed that those books as well as documentation for Smalltalk products, provided very good introductory information. But the level of detail was such that these materials did not really help the reader to see easily how real-world applications could be constructed in Smalltalk. Yet we realized, too, that once you have a Smalltalk environment in your hands, the natural thing to want to do is to build something useful and interesting. Helping you understand how to build applications is the sole focus of this book.

Who Should Read this Book?

This book is specifically aimed at people who are interested in learning to build real-world applications using Smalltalk/V (and, more specifically, Smalltalk/V 286). To some extent at least, it picks up where the Smalltalk/V tutorial leaves off, although it also includes a bit of refresher material that extends some of the information in that tutorial as well. We assume that you are already sold on OOP and on Smalltalk/V and that you are familiar with the basic programming concepts contained in the Smalltalk/V tutorial. We do not assume that you are a professional programmer.

You will benefit most from this book if you work through the projects it contains. But Smalltalk code tends to be quite readable; if you don't have ready access to a system running Smalltalk/V, you can still gain some insights into the design, coding techniques, and strategies that will make you a better Smalltalk programmer.

What Should You Expect To Learn?

Obviously, we expect you to make a reasonably serious time commitment to learning OOP and Smalltalk/V. Why should you do this? What's in it for you?

In brief, we expect that by the time you finish this book, you will learn how to handle the following programming-related tasks in Smalltalk/V 286 (and, by extension, in other flavors of Smalltalk/V as well):

- how to organize your classes into hierarchies that make them easier to reuse and extend
- where and how to find reusable classes in the Smalltalk/V class library
- how to design and implement projects related to all of the key concepts in Smalltalk/V, including simulations

You will do all of this by following through the book from start to finish as we take you through the process of learning the Smalltalk/V environment, tools, libraries, and techniques, on a practical level.

What Does this Book Cover?

This book is divided into 11 chapters, some of which may be longer than you are used to seeing in computer books. With the exception of Chapter 1, it follows a careful organization of background and theory followed by a specific project for you to build that reinforces and extends the concepts in the preceding chapter.

Chapter 1 is a brief look at the Smalltalk/V environment. It encompasses an overview of the Smalltalk/V tools, some emphasis on the Class Hierarchy Browser (CHB) and the Disk Browser, an introduction to the use of the Debugger, and a discussion of how to customize the Smalltalk/V environment to make it more compatible with your style of programming and design. Some of the material in this chapter reviews information found in the Digitalk tutorial, and some of it is new material.

Chapter 2 focuses on the Smalltalk language itself. The language is, of course, just one of the tools in the Smalltalk/V environment. This chapter looks at basic Smalltalk/V syntax, outlines some of the important classes in the Smalltalk/V class library that we'll be dealing

with in this book, and points out some key programming concepts that are either glossed over or not covered in detail in the Digitalk manuals.

Chapter 3 puts the material in the first two chapters to work in the first of five projects around which the book is centered. In this chapter, you'll build a small project that extends the Smalltalk/V environment by adding a simple but useful application to the system and demo menus. This application is called the *Prioritize^* it helps you to rank and sort through a list of choices you enter. You'll see how to design this application and become comfortable with the idea of modifying the Smalltalk/V environment. Many programmers coming to Smalltalk from other languages and approaches find a difficult psychological barrier in the fact that all of the programming you do in Smalltalk entails changing the environment itself. This chapter will help you over that hurdle and build a useful application in the process.

Chapter 4 turns our attention to the development processes involved in designing and creating Smalltalk/V applications. As you'll see, programming in Smalltalk is a little like peeling an onion. There are several approaches to programming, none of which is the best and all of which may actually be used in creating a single application.

Chapter 5 will then demonstrate the simplest approach to programming in Smalltalk/V as you build a small counter project. While simplistic in execution, this project requires that you understand the basic use of the Model-Pane-Dispatcher (MPD) triad that is at the core of all Smalltalk/V interactive applications.

Chapter 6 then exclusively focuses on this triad and examines the key role played by the MPD interactions. The most useful aspect of this chapter lies in the triage it performs on the relatively complex and extensive set of classes and methods that comprise the MPD architecture. By defining what is important for you to learn and understand, and differentiating it from what you can simply take for granted, this chapter will greatly accelerate your ability to grasp the MPD concepts that are crucial to successful Smalltalk/V programming. We'll take apart the ListPane class to see how the MPD architecture is implemented in Smalltalk/V's class libraries.

Chapter 7 then puts the MPD triad to work by helping you build a new type of pane, specifically a new sub-class of the **ListPane** class that allows the user to select more than one item from the scrolling list it contains. You'll see how this list requires the use of the MPD triad. We'll also describe the interaction between this list pane and a standard one that shows the user's selections as they change.

Chapter 8 turns our attention to the graphic world of Smalltalk/V. This language lives in an inherently graphic environment. Even on systems which do not feature a built-in graphic user interface, Smalltalk/V provides it. In this chapter, we'll look at the basic graphic concepts: forms, drawing primitives, and animation. Along the way we'll take a brief look at some of the mathematics involved in all graphic work.

Chapter 9 puts some of these basic graphic techniques to work as you build your fourth Smalltalk/V project, a graphing tool. This project was designed and constructed specially for this book by Morton Goldberg. You'll see the incremental development approach and understand how the graphic aspects of the program are used both to create the interface for the user and to generate the display of data. This chapter concludes with an enlightening discussion of how to extend this little example project into a more fully featured application.

Chapter 10 does for the text world of Smalltalk/V what Chapter 8 did for the graphic world. It explores Smalltalk/V text editing concepts, including the **TextPane, TextEditor,** and **TextSelection** classes and how they are used.

Chapter 11 provides you a chance to try out your knowledge of the textual aspects of the language and environment by building a flexible extension to Smalltalk/V's built-in methods of prompting users for textual responses. You'll learn how to design, build, and extend a dynamically modifiable blank form generator.

What This Book is Not

This book does not attempt to do any of the following:

- persuade you that OOP is important, efficient, effective, or indispens able. We believe it is all of these things, but other writers have been touting this technology for a long time and many of them are better at such writing than we are.
- teach you basic OOP concepts. You should understand classes, methods, inheritance, encapsulation, polymorphism, and other key OOP ideas, at least minimally, before you attempt to read this book. If you don't, then the Digitalk documentation provides some excellent background in the subject. There are numerous other books dedicated to teaching OOP.
- compare OOP or Smalltalk to other programming methodologies. If you are an experienced programmer, you can draw your own analo gies as you progress through the book. Those conclusions will be more

valid than any we might suggest because they are yours and are based on your experience. If you are not an experienced programmer, then you p. ibably neither care how Smalltalk and OOP compare to these other approaches nor have the background to understand the differences if we took the time and space to list them.

Some Notational Conventions

As with any programming book, this one adopts certain notational and syntactic conventions to make readability easier. Here are the most important of those conventions.

- All class names are printed in **bold** type except in section headings, which are already bold and where they are usually evidently class names from their context.
- All method names are printed in *italic* type, including in section headings.
- We use a special font to reproduce program listings. It looks like this:

  ```
  sample code
  ```

- We try to differentiate between elements of the Smalltalk language and environment that are peculiar to Smalltalk/V and those that are more generic and could be considered applicable regardless of the dialect of the language you choose. We use the term "Smalltalk/V" to mean that the subject being discussed is either unique to Smalltalk/V or implemented differently than in other versions of the language. Where we use the term "Smalltalk," we refer to the broader context of the language of which Smalltalk/V is but one (albeit the most widely used) implementation.

Contacting the Authors

We always enjoy hearing from people who have bought and read our books. Whether you want to tell us how much you enjoyed it and how much you learned, point out weaknesses, complain in general, or just exchange ideas about Smalltalk and OOP, you can reach us in any of several ways. Electronic mail is your best bet; we read and respond to electronic messages practically daily. You can reach Dan Shafer on CompuServe (71246,402),

MCI Mail (DSHAFER), or CONNECT (DSHAFER). If you want to drop us a line, Dan's address is 277 Hillview Ave., Redwood City, CA 94062. That's not too reliable, though; we may even move by the time this book is printed!

Dean Ritz is on the move as this is being printed, so if you have something specifically to say to him, your best bet is to send it through Dan Shafer using one of the above approaches.

Acknowledgments

The authors gratefully acknowledge the assistance of many people whose input, insights, and occasional interrogations helped shape this book. Among the more important people in this regard are:

Adele Goldberg, Stephen Pope, and several other people whom Dan met at ParcPlace Systems and who were responsible for his early Smalltalk education.

Dave Wilson, an OOP fanatic who contributed greatly to Dan's understanding of objects and their proper role in the universe (which Dave believes, of course, is everywhere!).

Timothy Randle, who spent many hours with Dean discussing OOP and Smalltalk and reviewing designs for some of the programs in this book.

Morton Goldberg, who spent a great deal of time reviewing the early chapters of this book and who designed and built the graphing application in Chapter 9.

Richard Szabo, Dean's good friend who first introduced Dean to computer science and demonstrated how something potentially so annoying could be so interesting.

Charles A. Rovira and Martin Shapiro, who reviewed portions of the manuscript, helped Dan over some Smalltalk rough spots and encouraged the project all without having to deal with Dan face to face, thanks to the miracle of CompuServe!

John Sellers, who painstakingly read and commented on a late draft of the manuscript and made several extremely helpful and insightful suggestions.

Dan Goldman, Barbara Noparstak, Mike Anderson, Mike Teng, and the rest of the gang at Digitalk, who put up with delays, repetitive questions, panic and other modes of human behavior that no self-respecting publisher should have to put up with. In addition, both **Jim Anderson and George Bosworth** were positive, encouraging, and enthusiastic in their support for this project from the first discussion about its possibilities.

The Editorial Staff at Springer-Verlag treated the project with care and enthusiasm.

Don and Rae Huntington of Production Services, who typeset the book, tolerated more than a few quirks in the authors and the manuscript, painstakingly proofread the copy, and in general made the book look and read as well as it does.

We've omitted dozens of other people whose support and encouragement, inspiration and help, have made this an enjoyable and, we trust you'll agree, successful project. Thanks to one and all!

Contents

1
The Environment

Introduction

In this chapter, we will look at the various tools that make up the Smalltalk/V programming environment. We will examine the Class Hierarchy Browser, Disk Browser, Inspector, Workspace, and the Debugger. We include some tips that help you make more effective use of the Smalltalk/V programming tools as you develop applications.

The documentation that comes with Smalltalk/V 286 (and the other flavors of Smalltalk/V as well) contains detailed instructions on the use of these tools. We won't attempt to duplicate that information in detail here. Rather, our focus will be on two aspects of each tool: an overview of its use (condensing information contained in the documentation in greater detail) and tips and hints about its use that don't appear in the documentation and that only arise from experience using the environment.

An Overview of the Environment

Any productive object-oriented development tool consists of three interrelated elements:

- tools that facilitate use of the language (editor, browser, debugger, etc.)
- the language itself (compiler)
- the class libraries that give the programmer access to the operating system, user interface, and other elements of program design

This chapter focuses on the first issue: tools. Chapter 2 reviews and provides some insights into the Smalltalk language as embodied in Smalltalk/V. The rest of the book uses these tools and the language in conjunction with the most important and useful classes in the class library to construct some interesting applications.

Most of the tools you'll find useful in the Smalltalk/V 286 environment are windows. All windows have certain things in common, including:

- a title bar containing the title of the window and providing a handle for the direct manipulation of the window
- one or more panes of various types
- pop-up menus associated with each pane
- optional icons that allow you to do things like close, collapse, zoom, or resize the window

Some windows also include buttons on which you can click with the mouse to change the state of things in the window. For example, the Class Hierarchy Browser (at which we'll look closely in a moment) has two buttons labeled "instance" and "class" that let you decide which type of methods you want to examine for the selected class.

Another behavior that all Smalltalk/V windows exhibit may take you some time to get accustomed to. You can only type in a window when that window is the topmost and active window. That's not too surprising if you've had any experience with Microsoft Windows or the Apple Macintosh. Some other windowing environments differ slightly in this behavior (e.g., UNIX X Windows). What *may* come as a surprise, though, is that the cursor must be physically located inside the window for you to edit its contents. If you are accustomed to shoving the mouse pointer out of your way as you type, you'll have to unlearn that behavior; Smalltalk/V 286 will simply beep at you if you attempt to type into a window that does not contain the cursor.

Your Smalltalk/V environment automatically includes one window, the Transcript, when you launch the system. This window cannot be closed or collapsed but it can be zoomed and resized as well as moved. You will quite often instruct your Smalltalk methods — particularly while you are developing and debugging them — to place information into the Transcript because you can be sure it will always be around and because displaying information there is relatively easy.

Using the Class Hierarchy Browser

Most experienced Smalltalk/V programmers always have at least one copy of the Class Hierarchy Browser open (or collapsed but accessible) in their environment. This window is used so frequently in Smalltalk programming that we'll shorten references to it and call it the CHB so you won't get tired of reading (and we won't get tired of typing) Class Hierarchy Browser.

Figure 1-1 shows the Class Hierarchy Browser as it appears when you first open it in Smalltalk/V. This window has five panes. The class list pane in the upper left lists all the classes in your current Smalltalk/V image. The names of classes which have subclasses are followed by ellipses. The method list pane in the upper right lists the methods associated with the selected class.

Notice that when you first open the CHB, no class is selected, so the method list pane is empty. Immediately below the method list pane are two sub-panes that look and act like buttons. One is labeled *instance* and the other is labeled *class*. Selecting one of these panes results in the instance or class methods of the selected class being shown in the method list pane. Finally, there is a larger pane at the bottom of the window that is a text editing pane where the source code of the selected method or class definition will appear. You can edit the text here; in fact, this is the place you will do most of your Smalltalk/V programming.

If you are working with code from two or more classes at once, you will find the most efficient way of setting up your environment will probably be to open a separate CHB for each class with which you are working. Each time you change the class you' ve selected in the upper-left list pane, you will lose the focus on the method you were editing before you made the change. This is quite often inefficient/Since Smalltalk/V lets you have a theoretically unlimited number of CHBs open at one time, multiple browsers will become a frequent part of your Smalltalk/V programming experience.

Remember from your Smalltalk/V tutorial work that you can also open a CHB with a specific focus, that is, a CHB that not only opens on a specific

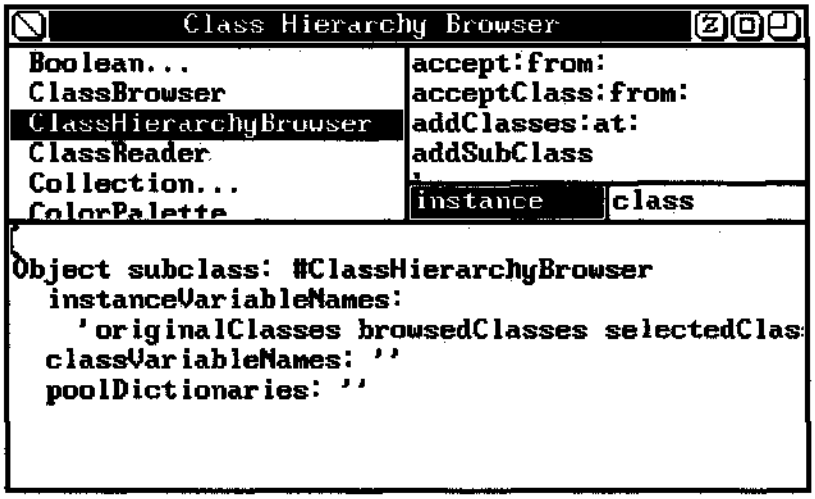

Figure 1-1. Typical Class Hierarchy Browser

class but one that contains a limited subset of the total Smalltalk/V class hierarchy. You do this by evaluating the following expression (probably in your Transcript or your Workspace):

```
ClassHierarchyBrowser new openOn: (Array with: <className> with:
    <className>...)
```

You will, of course, substitute the name(s) of the class(es) you wish to be contained in the CHB for the <*className*> place markers in the above line of code.

Templates in the CHB

Any time you create a new element of the Smalltalk/V environment in the CHB, you will be presented with a template for that definition in the lower, text-editing pane of the CHB' s window. Figure 1 -2 shows you the template for the creation of a new method, which is one of the most common operations you'll perform. Similarly, a relevant template appears if you tell Smalltalk/V through the CHB that you want to create a new class.

These templates are often useful in helping to remind you of the things that are required and expected in defining new Smalltalk/V objects.

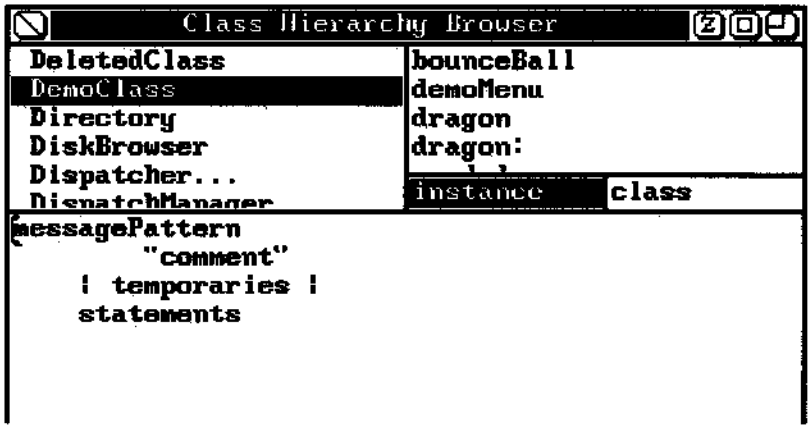

Figure 1-2. New Method Template in CHB

Removing Classes via the CHB

As you work with a Smalltalk/V image (see below), you will occasionally want to "clean it up" by removing classes you created for some experimental purpose but no longer need. Doing so is easy by means of the *CHB .provided there are no instances of the class* lying around the image. We find this is often a source of confusion even to Smalltalk/V programmers with some experience.

If you create an instance of a class — for example, by sending the class the message *new* and assigning the result to a global variable — and then try to delete the class, Smalltalk/V won't let you. Instead, it will present you with a walkback with the message "has instances" (see Figure 1-3),

Smalltalk/V, through its automatic process known as *garbage collection,* removes from the image any object that is not referred to by some other object in the system. So the *has instances* error indicates that you have created an instance of the class in question and cannot delete the class because these instances depend on it.

You can find out what instances Smalltalk/V finds for the class you're trying to delete by sending the *alllnstances* message to the class. For example, if you try to delete a class you created and named **Counter,** you can find out what instances of it exist by typing the following line in the Transcript or Workspace, selecting the text, and selecting *show it* from the pane menu:

```
Counter alllnstances
```

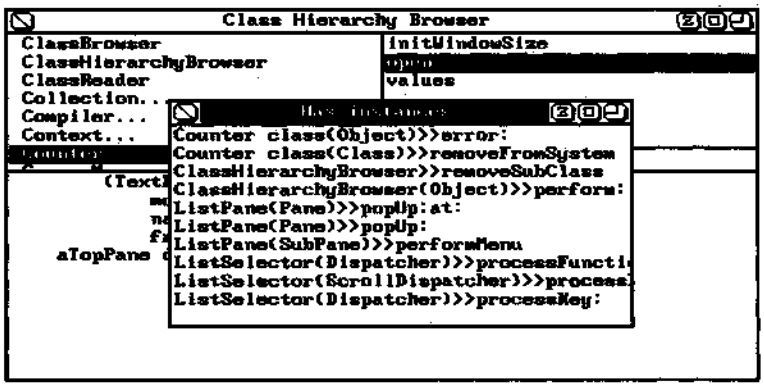

Figure 1-3. Walkback with "Has instances" Error

Then you need to type a line in the Transcript or Workspace removing the class affiliation for each of these references. If, for example, the above line revealed that you had created a new counter at some point called *MyCounterl*, you could do something like the following:

```
MyCounterl := String new
```

After doing this for all of the objects that reference the class you wish to remove, you can then return to the CHB and remove the class.

A more effective way to accomplish this task in one step (if you are sure you want to remove all instances of a particular class so that you can change its definition or remove it) is to evaluate the following expression:

```
Counter alllnstances do:  [reach |
    each become: String new]
```

This expression transfers all the references (pointers) to each instance of Counter to an empty string.

The Smalltalk/V Image

We have mentioned the concept of an "image" in Smalltalk without explaining it. On one level, an image is a simple and basic concept. It can be thought of as the current environment that contains all the compiled methods and all of the instances of active objects.

Whenever you run Smalltalk, you work with an image. You can only work with one image at a time. As you develop applications, you may well have occasion to store more than one image on your disk, particularly as you wish to back up previous work or operate on different versions of the same application or even create other applications. Each separate application, unless they are to be delivered together for some reason, is a separate image.

One of the most helpful and interesting aspects of an image is the Changes Log. This log keeps track of all the changes you make in an image. It should be saved each time you save the image itself. You must keep the change log with its associated image, since they are bound logically but not physically. It is a common mistake to use a change log with the wrong image. Eventually, of course, this portion of the image can become quite large. You can reduce the size of your image by evaluating the following expression:

```
Smalltalk compressChanges
```

When you evaluate this expression, Smalltalk/V clears out all of the contents of the change log except for the latest copy of each new or changed method. It also removes class definitions and writes a new image file automatically.

You can save even more disk space by compressing the entire source file, which is logically also part of the image. Do this by evaluating the following expression:

```
Smalltalk compress Sources
```

Evaluating this expression creates an entirely new source file for all methods in the system, using the most recent copy of the source code for each method. It then empties the change log and automatically writes a new image to the disk. The result, then, is a new base system with which to work.

Using the Disk Browser

You can open a Disk Browser any time you wish simply by selecting the *browse disk* option from the system pop-up menu. You can have multiple instances of Disk Browsers open in your environment at a time (typically, you may have one for each disk that is currently in use).

The Disk Browser (see Figure 1-4) has two list panes in its upper portion. The leftmost one contains the directory of the disk and the subdirectory currently being viewed. The rightmost one contains a list of all the files in that directory or subdirectory.

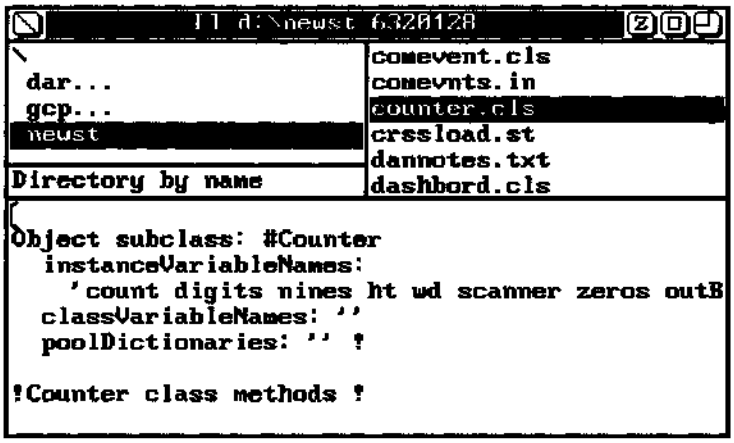

Figure 1-4. Typical Disk Browser

The bottom pane is a text editing pane where the contents of the selected file appear unless the small list pane that acts as a button just below the directory list pane is activated with the right mouse button. In that case, a popup menu appears (see Figure 1-5) and the user selects the order in which he wishes to examine the current directory. The directory, with complete file information, then appears in the text editing pane in place of specific file contents. Clicking on the name of a file in the rightmost list pane displays that file's contents. If a file has more than 10,000 characters in it, Smalltalk/V notifies you of that fact with a line at the top of the text editing pane and shows you the first 2,000 and the last 8,000 characters. (You could, of course, change these limits since the source code for the browser is available to you in the system.)

One excellent use for the Disk Browser, by the way, is note-taking during program development. You can create a new file in the Disk Browser by selecting the *create* option from the popup menu in the rightmost list pane. Then you can edit this file any time you like. You can save the contents of the file by selecting the *save* option from the text editing pane's popup menu.

You can manage most of your DOS file environment from inside a Smalltalk/V Disk Browser. In fact, we know several Smalltalk programmers who do just that, essentially "living" in Smalltalk/V 286 and handling their entire user configuration and file structure from within the programming environment. You can create, remove, print, rename, copy, edit, and even change the system file mode attributes from within the Smalltalk/V

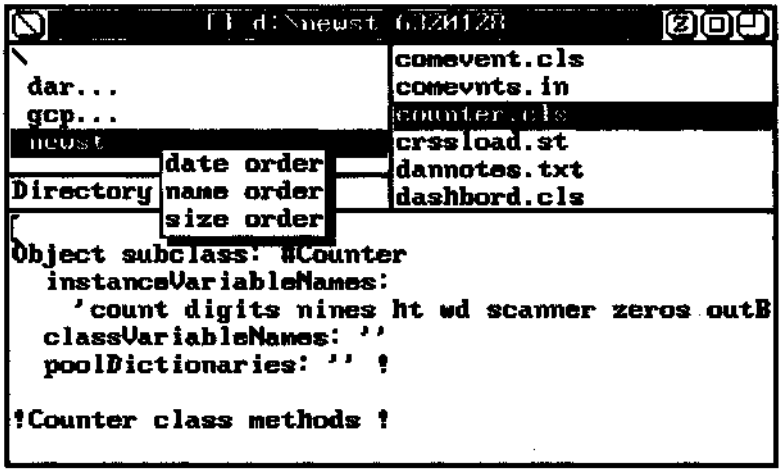

Figure 1-5. Disk Browser With Directory Style Popup

Disk Browser. (Of course, you can't run other DOS applications from inside Smalltalk/V 286, so you can only handle operating system and Smalltalk-related activities this way.)

Using the Other Browsers

There are two other types of browsers in Smalltalk/V for which you may find occasional use. They are most useful when you are "playing detective" and trying to find a particular class or method in the Smalltalk/V hierarchy.

The first browser is a *Class Browser* that lets you browse a particular class. You can create such a browser by sending the message *edit* to any class or by choosing the *browse* option from the class list pane of a CHB after having selected a class.

You can also open a *Method Browser* that lets you examine and edit a particular method that may be implemented in more than one class in the system. You create such a browser by choosing a method name in a browser and selecting either *senders* or *implementors* from the menu of the upper-right pane. In either case, you will be presented with a browser that lists all of the methods that send (or implement) the particular message in which you can view the source code of the methods.

Quite often, you will find that creating a series of these browsers, looking first at classes, then at methods, then at implementors and senders of messages, can give you a very complete picture of how a particular class operates or how a specific method propagates throughout the hierarchy.

Using Inspectors

Next to the CHB, the most useful window in the Smalltalk/V 286 environment is the *Inspector*. You'll use inspectors primarily during the process of debugging your Smalltalk/V applications.

While a CHB (or other browser) lets you look at a class or method as it represents a part of the Smalltalk/V image and hierarchy, an inspector lets you examine the contents of the instance variables of a particular object or instance of a class.

Actually, you can do much more than merely examine contents with an Inspector. You can also change these contents. In addition, you can even compile and execute expressions in the text pane of an Inspector with the *save* menu option in that pane.

To open an inspector, you can either send the *inspect* message to an object or choose *inspect* from the popup menu of the Debugger's list pane that shows the instance variables of a selected object. Figure 1-6 is an example of this usage of the inspector.

Here, we created a walkback by pressing Control-Break. (You should know that you can press 'Control-Break' at any time so that you can inspect what is going on in the system at any place in your development.) By choosing *debug* from the walkback's popup menu, we opened the Debugger. Then, as you can tell from Figure 1-6, we choose the *inspect* option from the middle list pane at the top of the Debugger window after selecting *self in* that pane. Then we were presented with an Inspector window (the topmost window in Figure 1-6) and chose to examine the value of an instance variable called *inPanic.* You'll be happy to know that it contains the value *false,* meaning that we were not in panic when we pressed Control-Break.

You can examine the value of any other instance variable in the inspector. This is often insightful as you debug your programs because the failure to assign a correct or inappropriate value to an instance variable is one of the most common sources of Smalltalk/V programming errors.

(An instance variable is a piece of data associated with each instance of a particular class. If a class defines an instance variable called, for example, *name,* then every time you create an instance of that class, it will have an instance variable called *name.* Like all variables in all programming languages, instance variables have values associated with them. Generally, you can both read and write these values.)

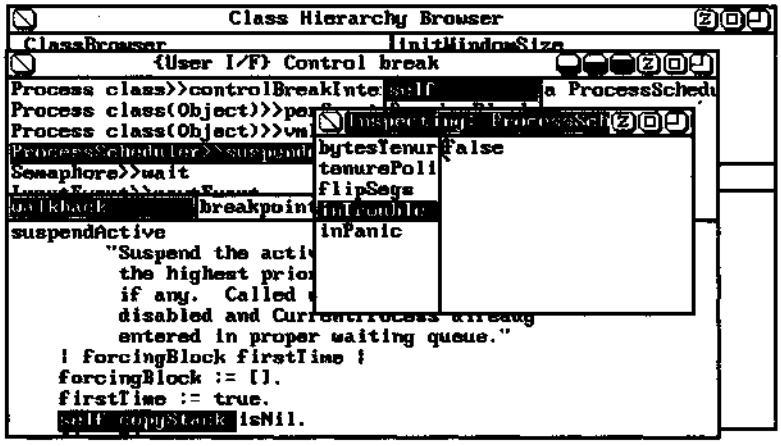

Figure 1-6. Inspector Opened from Debugger

Using Workspaces

We will use workspaces quite frequently in this book, so it's a good idea to get comfortable with how they are created and used right at the outset of your work here.

You can think of a Workspace as a blank slate that you can use as a free-form text editor for any purpose. The Transcript that we discussed earlier is just a special case of a Workspace. You can only have one Transcript window open in your environment at any one time. As we mentioned earlier, the system forces you to have one open. But you can have from zero to a theoretically unlimited number of Workspace windows in your environment.

To create a new Workspace, you just choose *open workspace* from the system popup menu. Smalltalk/V will give you a new, empty Workspace window (see Figure 1-7) with the name "Workspace."

Notice that a Workspace has only one pane, a large text-editing pane, and a title bar with some icons and a label. You can move, resize, collapse, zoom, and close a Workspace. You can freely edit in the pane as well, but remember that this pane is a Smalltalk/V text-editing pane, so word wrap is not available. You'll have to press the carriage return when you want lines to end.

If you close a Workspace that has some text in it, you'll be asked (see Figure 1-8) if you want to discard the changes or not.

We usually give a Workspace a new label after we open it, naming it in such a way that we can determine its purpose at a glance. Then when you save the image next time, this Workspace stays available until and unless you close it after saving its changes or telling Smalltalk/V to discard the modifications.

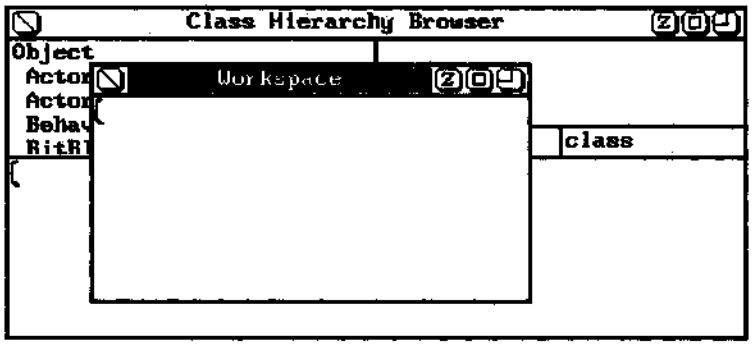

Figure 1-7. A New Workspace

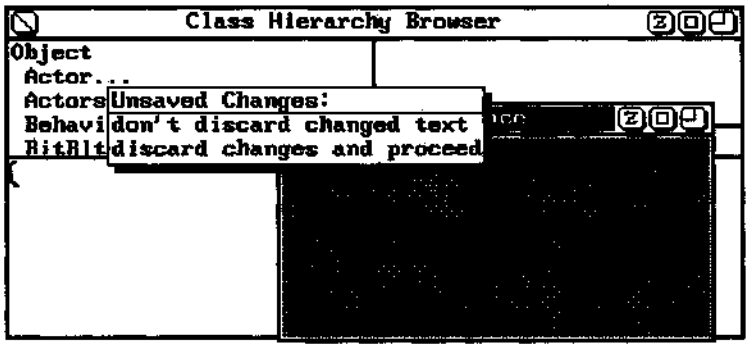

Figure 1 -8. Closing Workspace With Changes

One good use of a Workspace is to create a Workspace, call it something like "Useful Stuff," and place into it code fragments for carrying out useful, frequently needed tasks. For example, when we described earlier how to identify all instances of a particular class you are trying to delete, you could have placed that code in your chosen Workspace. The next time you need to delete a class and get the "has instances" error, you can select the "Useful Stuff Workspace, change the name of the class involved, select the expression and *do it* without having to remember what command to use. If you take this approach to the use of a Workspace window, be sure to comment the code fragments so you' 11 be able to remember later why you' ve stored them so carefully. (You can use the Disk Browser, discussed below, to create a text file that contains these pieces of information. These files are normal DOS files and can be saved, renamed, deleted, etc. This is in many ways a better solution than using a Workspace for this purpose.)

Using the Debugger

The Debugger is one of the richest tools available to you as a Smalltalk/V programmer. It combines the functions of a debugger, browser, and inspector into one immensely useful tool. You will undoubtedly get to know this set of windows much more intimately than you had hoped before you have finished building your first complex application. We'll find frequent uses for the Debugger walkback and main window throughout the book. In this section, we'll take a look at its general use and contents. Then in Chapter 3 we'll take a look at some of its limitations as we see it in use in debugging our first Smalltalk/V project.

There are four ways for the Smalltalk/V Debugger to be invoked:

- by the system encountering an error condition
- by the user pressing Control-Break
- by sending the message *halt* to any object
- by Smalltalk encountering a method for which the programmer has set a breakpoint

We'll look in this chapter at the use of the Debugger in all but the last two situations.

Whenever an error occurs in a Smalltalk/V application, the system creates a Debugger Walkback window. This walkback has two aspects that interest us: its label and its pane contents. The label provides the message that triggered the error condition (in most situations, at least) and the pane shows us the list of messages that have been processed in the current processing chain. The last method executed always appears on the top; as you go down through the list, the messages sent or methods called are less and less recent. (To be precise, this pane contains a nested list of all calls that are currently being executed.) The list represents the most recent portion of the message-passing chain for the currently executing method. It obviously includes many messages that your code does not originate or deal with and leaves off some with which your code did deal.

Let's deliberately create an error that will produce a Debugger Walkback. In the Transcript (or in a Workspace), type the following nonsense code:

```
•a1 +6
```

Now select and *do it*. You'll get a Debugger Walkback like the one shown in Figure 1-9.

When the Debugger Walkback window appears, you can do two things with it: you can close it or you can open the Debug window by selecting "debug" from the pane pop-up menu. You can close it either by clicking its close icon in the upper left corner of the title bar or by choosing *resume* from the pane pop-up. In most cases, you'll probably open a Debug window. Occasionally, the Debugger Walkback's label and the message list are sufficient for you to see quickly what has gone wrong. In that case, of course, you can simply dismiss the Debugger Walkback, fix the offending code, and resume your work.

Generally, the first entry or two on the message list are not very useful. They usually present the messages that deal with the error-handling process itself. As such, they don't provide a lot of insight into what is happening with the code that has gone awry.

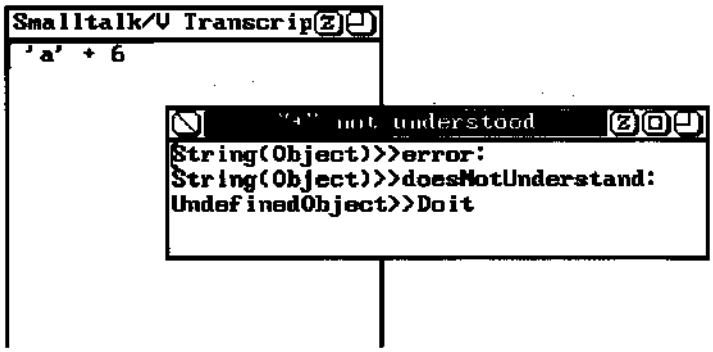

Figure 1 -9. Sample Walkback

Bring up the Debugger window for the Walkback we just created by choosing *debug* from the pane pop-up. Select the second method in the upper left pane. The resulting Debugger window is shown in Figure 1-10. The Debugger window has six panes, two of which look and act like buttons. The top left pane reproduces the message list from the Debugger Walkback window unless the "breakpoints" pane below it is selected, in which

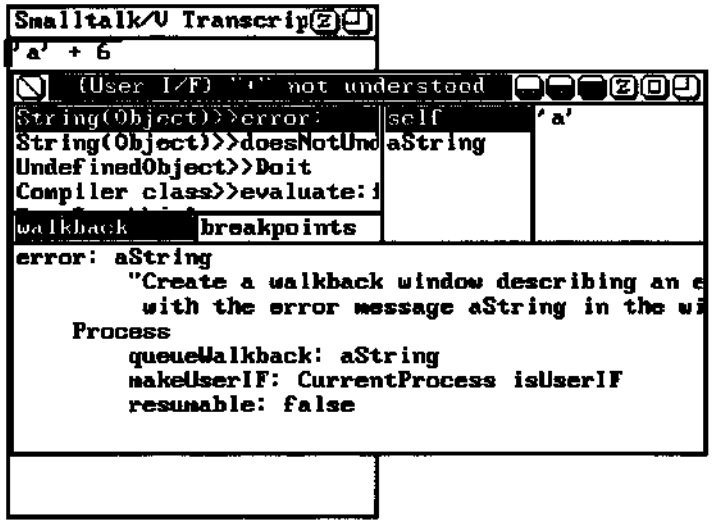

Figure 1-10. Debugger Window

case it lists all of the methods for which breakpoints have been set. (We'll have more to say about breakpoints and Debugger execution shortly.)

When you select a method from the upper-left pane, the next pane to the right displays the receiver of that message as its first line and then all of the temporary (i.e., local) variables associated with the method. Selecting one of these variables, in turn, results in its current value being displayed in the right-most pane. In Figure 1-10, we selected the first method from the method list and Smalltalk/V selected the receiver, *self,* for us. We can see that *self* (which is Smalltalk/V shorthand for the object that is the receiver of the message) has a value of *'a'*, indicating that it is a string. So now we know what object the current message is being sent to. What is the message itself?

Clicking on the second entry in the variable list pane doesn't tell us much. When we do that, the rightmost pane displays the rather cryptic *a Message,* but we can investigate further. Click the right mouse button on the *aMessage* entry in the variable list pane and choose *inspect* from the resulting pop-up. (That's not too hard; *inspect* is the only choice!) This opens an Inspector on *aMessage* so we can see its real contents. (You can also open an Inspector simply by double-clicking on the variable's name.) When the Inspector appears, the leftmost pane has the name of the receiver — again, *self*— as its first entry. Click on the second entry, *selector,* and you'll see the actual message (see Figure 1-11) being sent is the addition operator, +.

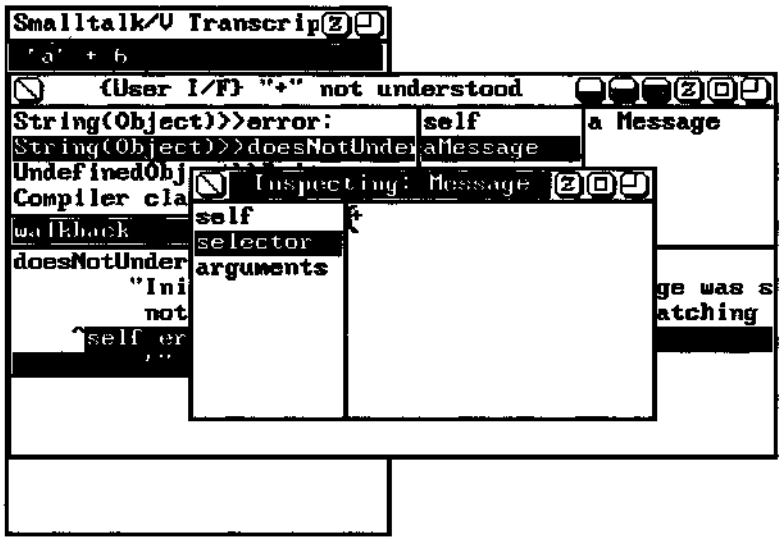

Figure 1-11. Inspector Opened from Debugger Window

Now we 're getting somewhere. We know that the addition message has been sent to a string object called 'a' and that the string object (not surprisingly) doesn't understand the message. If this error had arisen during execution of a method rather than from the manual execution of a message from the Transcript, we could go back to the Debugger window at this point, find the offending code, and fix it.

To summarize the Debugger's basic use, then, you will typically follow a procedure that resembles the following:

1. A Walkback will appear, indicating an error has arisen.
2. Unless the error is obvious from the Walkback itself, you'll select *debug* from the Walkback's pane menu.
3. In the resulting Debugger window, you'll examine potentially mean ingful methods from the leftmost pane, typically starting two or three messages down the list to get past the methods that deal with error-handling.
4. At some point, you'll have a likely suspect for the source of the error. You can then open an Inspector on a specific variable or argument associated with that method.
5. Inspecting the values of the variables will probably ultimately lead to an insight that reveals the error.

There are some kinds of errors that the Debugger just can't handle. We'll take a look at one of these situations in Chapter 3.

2
The Smalltalk/V Language

Introduction

Having examined the programming tools of Smalltalk/V 286 in Chapter 1, this chapter focuses on the other two major components of the environment: the language and the class library. The first part of this chapter will review the basic syntax of the Smalltalk/V programming language. While much of this material duplicates information contained in the Smalltalk/V 286 documentation, it capsulizes the syntax of the language in a way that we hope you' 11 find useful. The second part of this chapter will discuss the major classes in the Smalltalk/V 286 class library. By "major," we mean the classes with which you must be most concerned in a huge percentage of your programming work.

At the end of this chapter, then, you should be more comfortable with the syntax and structure of Smalltalk/V, and be better equipped to program in it.

Review of Basic Smalltalk Syntax

Everything that happens in a Smalltalk/V application happens as a result of a *message* being sent to an *object* which supports a *method* of the same name as the message. Seen from the broadest perspective, Smalltalk/V programming consists of defining objects with methods where some of the methods send messages to other objects.

Syntactically, we can look at Smalltalk/V at two levels: that of the message-passing process and that of the method-execution process.

Message-Passing Syntax

All messages in Smalltalk/V have the same basic syntax. The receiver of the message — i.e., the object to which the message is being sent — appears first, followed by the message. To open an Inspector window on the class **DemoClass,** for example, you send this message:

```
DemoClass inspect.
```

The receiver, **DemoClass,** is passed the message *inspect.* In effect, the class is told, "Inspect yourself." This order of expression is opposite that used in traditional programming languages and is one of the most frequent sources of confusion to Smalltalk/V programmers who have experience in other languages. The temptation is to write the above line as "inspect DemoClass" because you are accustomed to having the commands first and the data structures on which they operate second. This is the natural sequence of verb-object in English declarative sentences as well.

The message you pass to an object can consist of a single word (as in *inspect)* or of a series of words (keyword). Whether there is one or a number of these elements, it is called the *selector.* By convention, a keyword that ends with a colon expects to have an argument associated with it, while one without a colon does not expect an argument. You might have multiple arguments associated with a selector. This results in quite long message names like this one:

```
labelArray:lines:selectors
```

You should recognize this as a single message selector consisting of three keywords, each of which requires an argument. The message name will appear as shown above when you examine it in the CHB or in another browser. Each colon is a signal to you as a programmer that you need to supply an argument, and to the language compiler that it should expect to find one there.

There is one other type of message that requires an argument but has no colon notation. This type of message is called a binary selector. The clearest examples of such a selector are the math operators like the addition sign (+). The following line of Smalltalk/V code sends the message "+" to the object 6 and passes the argument of 4 in the process:

```
6 + 4.
```

You can evaluate expressions from any place where you can edit text, such as a Workspace or the Transcript or from within a method definition. To send a message from a Workspace or Transcript window, you type the message, select it, and then choose either *do it* or *show it* from the pane menu.

The choice of whether to select *do it* or *show it* can seem unimportant but it isn't. For example, if you type the above addition message into the Transcript, select it, and then choose *do it* from the menu, nothing appears to happen. In fact, Smalltalk/V carries out your instruction by adding 6 and 4. But you didn't tell it to do anything with the answer, so it didn't. Select the same message and choose *show it* from the pane menu and the answer appears in the pane.

You will often find it necessary to send a series of successive messages to the same object. In this case, you can avoid repeating the name of the receiver object by using message cascading. Each message sent to the same object is followed by a semicolon until the final message is sent. The entire cascaded message group is terminated with a period like any other Smalltalk message.

Method-Definition Syntax

When you define a new method, whether you do so in the CHB or another browser or in a Workspace or Transcript window, you will follow the syntax shown in the code sample below:

```
messagePattern
 "Comment describing message"
 [primitive number]
 [[temporries]]
expressionSeries
```

The *messagePattern* includes the method selector and the variable names used to refer to arguments in the method. It therefore defines how to phrase a message you wish to send to this object to cause this method to execute.

The comments are optional, but the template for defining a new method in the CHB includes them and they are strongly recommended by experienced Smalltalk/V programmers and designers. Something brief that describes the purpose of the method will suffice. (In fact, it could be argued that if your method requires copious comments to explain it, it is probably too monolithic and large. You should probably consider breaking it into smaller components that are easier to describe because they are single-purpose and focused.)

You will seldom have need to identify *a primitive number* in your methods. Primitives in Smalltalk/V are low-level routines that carry out fundamental computer operations (such as addition, file saving, and logical comparison). You can directly invoke one of these primitives by putting a line like this in your method definition:

```
<primitive: 58>
```

The primitives pre-defined by Smalltalk/V are described in an appendix to the Smalltalk/V documentation.

(Incidentally, you can also create new primitives in Smalltalk/V. In that case, your primitives are not referred to by numbers like the system primitives. Rather they will have a name you have assigned to them and will be invoked using that name.)

If your method uses any temporary, or local, variables, they must be defined at the beginning of the method. Such variables are placed between two vertical bars (created with the shifted backslash on the standard IBM PC keyboard) and separated from one another by spaces if more than one is needed.

The *expressionSeries* portion of the method definition is the part of the code that actually performs some operation when the method is invoked.

Let's take a look at a typical Smalltalk/V method to see how the various pieces (with the exception of the primitive) look and behave. Open a CHB if one is not already opened and choose the class **Magnitude.** If the name is followed by three dots (indicating it has hidden subclasses), double-click on it to open the list of subclasses. Now choose the subclass called **Integer.** Scroll down its method list and pick the *gcd:* method. Examine the code in the text-editing pane.

You can see from the code listing that the *gcd:* method takes an integer as an argument. Since it is a member of the class **Integer,** its receiver must also be an integer value (or a subclass of **Integer).** The comment tells you that this method returns the greatest common denominator of the receiver and the argument (i.e., the largest number that can be divided into both of them without a remainder).

Notice that this method defines three temporaries called *a,* v, and *r.* If you examine the expression series that makes up the executable portion of the method definition, you'll see that all of these variables are used.

Chances are you've never seen this method definition before, but just by reading the source code, you can tell what it does and how to use it. How do you think you could use this method in the Transcript to find the greatest common denominator between the numbers 12 and 18 ? It' s easy, right? Just type the following line, select it, and choose *show it* from the pane menu:

```
12 gcd:  18
```

Smalltalk/V returns the value 6, which is the largest integer number that can divide into both 12 and 18 evenly.

Summary of Syntax

The complete syntax of the Smalltalk/V language is, of course, more complex than what we have just described. But this explanation gives you

enough information to use the language with facility. Nitty-gritty details are discussed in the Smalltalk/V documentation.

The Essential Classes

Smalltalk/V 286 comes equipped with more than 100 classes and nearly 2,000 methods. (Other flavors of Smalltalk/V have different numbers of classes and methods, and all versions come with classes associated with such activities as tutorials that are not technically part of the system.) As you build applications, acquire code from other sources, and work with the environment, the number of classes and methods can grow quite substantially. If you had to be familiar with all of these classes and methods, you might arguably never be confident as a Smalltalk/V programmer.

Fortunately, you can safely ignore many of the classes that exist in the Smalltalk/V class hierarchy because they are either used primarily by the system or have such esoteric functions that the likelihood you'll ever need them is reduced. In this section, we'll identify the classes with which you will want to become most familiar and comfortable as you begin your exploration of Smalltalk/V.

We won't supply a great deal of information about each class in this list. The Smalltalk/V documentation from Digitalk contains detailed descriptions of all classes and methods and we will demonstrate many of these classes in even greater detail when we use them to build our six projects later in the book.

Figure 2-1 shows you the entire Smalltalk/V class hierarchy. Those classes we consider essential to your effective use of Smalltalk/V are shown in boldfaced type in the figure.

It is a tribute to the compactness with which Smalltalk/V's class library is defined that even with the process of elimination we've undertaken, there remain 54 classes with which you will have to have at least a nodding acquaintance to undertake most of the programming work you'll do in this book and in the real world of Smalltalk program development. (In reality, you will probably deal with a subset of these 54 classes for any particular application. But because our intent in this book is to give you broad exposure to Smalltalk programming, we have chosen a broad subset of the total class hierarchy to discuss in varying levels of detail.)

We will look briefly at each of these classes, presenting them in the order in which they appear in Figure 2-1. In each case, we'll describe briefly the purpose of the class and, where appropriate, the types of applications in which it might be used. We will also refer you to the chapter in this book where the class is discussed in greatest detail. (Recognize, however, that sometimes this level of detail is not great because we can focus our attention on a small number of methods in a given class for our purposes.)

Object
- Behavior
  - Class
  - MetaClass
- **BitBit**
  - **CharacterScanner**
  - **Pen**
    - Animation
    - Commander
- Boolean
  - False
  - Truce
- ClassBrowser
- ClassHierarchyBrowser
- ClassReader
- **Collection**
  - **Bag**
  - **IndexedCollection**
    - **FixedSizeCollection**
      - **Array**
        - CompiledMethod
      - Bitmap
      - ByteArray
        - FileHandle
      - Interval
      - **String**
        - **Symbol**
    - **OrderedCollection**
      - Process
      - **SortedCollection**
  - **Set**
    - **Dictionary**
      - IdentityDictionary
        - MethodDictionary
      - SystemDictionary
    - SymbolSet
- Compiler
  - LCompiler
- Context
  - HomeContext
- CursorManager
  - NoMouseCursor
- DeletedClass
- DemoClass
- Directory
- DiskBrowser
- **Dispatcher**
  - **GraphDispatcher**
  - PointDispatcher
  - **ScreenDispatcher**
  - ScrollDispatcher
    - ListSelector
    - **TextEditor**
      - **PromptEditor**
  - **TopDispatcher**
- DispatchManager
- DisplayObject
  - DisplayMedium
    - **Form**
      - BiColorForm
      - ColorForm
      - **DisplayScreen**
        - ColorScreen
- DOS
- File
- Font
- Icon
- InputEvent
- Inspector
  - Debugger
  - DictionaryInspector
- **Magnitude**
  - **Association**
  - **Character**
  - **Date**
  - **Number**
    - **Float**
    - **Fraction**
    - **Integer**
      - LargeNegativeIngeger
      - LargePositiveIneger
      - SmallInteger
  - **Time**
- **Menu**
- Message
- **Pane**
  - **SubPane**
    - **GraphPane**
    - **ListPane**
    - **TextPane**
  - **TopPane**
- Pattern
  - WildPattern
- **Point**
- ProcessScheduler
- **Prompter**
- **Rectangle**
- Semaphore
- **Stream**
  - **ReadStream**
  - **WriteStream**
    - **ReadWriteStream**
      - **FileStream**
      - **TerminalStream**
- **StringModel**
- **TextSelection**
- UndefinedObject

Figure 2-1. Smalltalk/V Class Hierarchy Showing Essential Classes

Object

You will only sub-class this class. You never create an instance of the class **Object,** which is an abstract class that defines behavior common to all objects in the Smalltalk/V class hierarchy. We will not explicitly discuss this class and its contents in the book; rather, you will learn about this class as you sub-class it to create your own special classes for applications and projects.

BitBlt

This class is used in graphics programs and operations. Its purpose is to provide a mechanism by which bits representing an image can be moved from one place to another. We will make use of this class in Chapter 8.

CharacterScanner

The CharacterScanner class plays a key role in text-based applications. Its role is to translate characters from their standard ASCII code representations to bitmapped images representing the appearances of the characters on the screen. We will discuss this class in Chapters 10 and 11.

Pen

This class is a subclass of the class **BitBIt. It** enables you to create a drawing implement that can be instructed to move, draw lines and shapes, and erase its trail as it moves. If you are familiar with the programming language Logo, this class lets you emulate the turtle graphics that made that language so popular and well-known. We will study this class in greater detail in Chapter 8.

The Collection Classes

One of the largest groups of classes with which we will work in this book and which form an important core of classes for your Smalltalk programming experience are the many subclasses of the abstract class **Collection.** Loosely described, a **Collection** is a basic data structure used to store objects in

groups. They may be stored in any of several forms, sorted or unsorted, paired with other key objects or in unary fashion.

(We should point out here that **Collection** is one of several classes in Smalltalk that are described as *abstract classes.* An abstract class is simply a class that is defined as a place to organize and collect common behavior among other classes. You almost never create an instance of an abstract class. Its purpose is not to provide a class definition that is useful to instantiate but rather to provide a convenient place for behavior common to other, more useful classes. You'll see more precisely what we mean as you study several abstract classes later in this chapter.)

The three main subclasses of the class **Collection are:**

- Bag, in which duplicate elements are allowed to gather and the elements of which are stored in no particular order.
- **IndexedCollection,** in which duplicate elements are allowed to gather but the elements of which are either stored in some pre-determined order (i.e., sorted) or are at least accessible by an integer index (from which this subclass derives its name).
- Set, in which no duplicate elements are allowed and the elements of which are stored in no particular order.

The class **IndexedCollection,** in turn, has two main abstract subclasses: **FixedSizeCollection and OrderedCollection.** The first subclass consists of a group of different classes distinguished by the fact that when you create a new instance of one of them, you must give it a size and that size then remains fixed for the life of the object. Instances of class **OrderedCollection,** on the other hand, can shrink or grow dynamically as needed. Among the several subclasses of the class **FixedSizeCollection are:**

- **Array,** which is a collection which can contain a mixture of any variety of object types.
- **String,** which is a group of characters in an indexable sequence. We don't usually think of strings as collections of individual characters from a programming standpoint but Smalltalk/V maintains consis tency even at this detailed level.
- **Symbol,** which is a subclass of the class String consisting of guar anteed unique sequences of characters of which the system makes special use.

The most important subclass of the class **Set** is **Dictionary.** You can think **of a Dictionary as a Set of Associations. An Association** is a pair of objects: a unique key (usually a symbol, but can be any object) and a value (any object). A **Dictionary** has an **Association** for each key. Each key in turn has a value which may or may not be unique. This description corresponds to a

real-world dictionary in which you look up words (keys) to determine their definitions (values). The keys are the words themselves, arranged alphabetically in the dictionary.

We will use various members of the class Collection and its subclasses throughout the book.

Dispatcher

The various members of the class **Dispatcher** are responsible for dealing with user input via the mouse and keyboard. Dispatchers are part of the important model-pane-dispatcher (MPD) triad that is the focus of Chapters 6 and 7. Smalltalk/V defines several subclasses of the abstract class **Dispatcher,** the most important of which for our purposes are the following:

- **GraphDispatcher,** which deals with events in graphic panes.
- **ScreenDispatcher,** which handles events that take place outside any window or pane (i.e., on the background of the Smalltalk/V environ ment or desktop).
- **TextEditor,** which specializes in text-editing input (character input, editing commands, etc.).
- **TopDispatcher,** which is always associated with an instance of the class **TopPane** and is therefore responsible for handling events di rected not to a particular subpane but to the window as a whole.

Form

All drawing in Smalltalk/V graphic applications involves bitmapped displays. The class **Form** holds these bitmaps and contains behavior to initialize and manipulate them. We'll take a close look at this class in Chapters 8 and 9.

DisplayScreen

This sub-class of the class **Form** is a special bitmapped object whose size, shape, and other characteristics are hardware-dependent. You'll often use this class as the target for messages when you wish to show bitmapped images on the Smalltalk/V screen. We learn some of these techniques in Chapters 8 and 9.

Magnitude Classes

Like the class **Collection** which we discussed above, the class **Magnitude** has many subclasses and many important uses in Smalltalk/V. We'll use instances of this class and its subclasses throughout the rest of the book. The class **Magnitude** itself is an abstract class where objects that can be compared, counted, and measured are grouped together. The major subclasses of the class **Magnitude** are as follows:

- **Association,** which can be used to define objects made up of key/value pairs. (See the discussion of the class **Dictionary** in the section on **Collection** classes above.)
- **Character,** which defines the behavior of all characters in the system (i.e., those objects represented by ASCII codes from 0 to 255).
- **Date,** which represents a particular day and which can be used to manipulate months, days, days of the week, years, weekdays, week ends, and other calendar-related activities.
- **Number,** an abstract class where all types of numeric values are managed. This class includes the basic behavior for all numbers of several types, chief among which are **Float** (numbers expressed as IEEE double-precision floating-point values), **Fraction** (numbers expressed as one number divided by another), and **Integer** (whole numbers with no decimal or fraction parts).
- **Time,** whose instances are a particular time and which is used to manipulate time-related objects, in this case objects containing infor mation about hours, minutes, seconds, and fractions of seconds.

Menu

The class Menu contains all methods required to create, display, and respond to the user's interaction with popup menus in your Smalltalk/V applications. All menus other than the System Menu are associated with a pane; as a result, the class **Menu** is discussed when we talk about the MPD triad in Chapters 6 and 7.

The Pane Classes

The class **Pane** is an abstract class which provides a place for windows (referred to in Smalltalk/V as instances of the class **TopPane** or simply as "top panes") and the panes these windows contain (more accurately referred to as sub-panes, all of which are members of the class **SubPane)** to be defined. There are three types of sub-panes, each of which is a sub-class of **SubPane:**

- **GraphPane,** which can be used to display graphic and drawing elements.
- **ListPane,** which displays scrolling lists of one-line entries from which the user can select one element.
- **TextPane,** which is a text-editing pane in which text can be displayed and manipulated.

All Smalltalk/V windows — including those generated by the system as well as those you build in your programs — consist of one and only one instance of the class **TopPane** and one or more instances of the **SubPane** classes.

Point

The class **Point** describes the operations that can be performed on objects that are composed of a pair of x-y coordinate values defining a point on the display. Points are important for our purposes primarily in defining the locations **of SubPane** objects in windows (Chapters 6 and 7) and in graphics (Chapters 8 and 9).

Prompter

You will find frequent use for instances of the class **Prompter** in your Smalltalk/V programming. A prompter is a small window that asks the user a question and waits for a response to be typed. In fact, a prompter is a

window with one instance of the class **TextPane.** The user must supply a response to the question that appears as the label of the prompter before the prompter can be dismissed. Thus a prompter is equivalent to what is referred to in windowing environments as a *modal dialog.* Unlike most modal dialogs in other environments, however, a prompter accepts only one type of input (text).

We will create instances of the class **Prompter** at several places in our programming throughout this book.

Rectangle

An instance of the class **Rectangle** describes a rectangular area either by defining its upper left and lower right corners (i.e., absolute location of both coordinates) or by describing its upper left corner and an extent (length and width) for the rectangle.

Rectangles are used most often in graphics (Chapters 8 and 9) but also have some significance in defining the sizes and relative positions of subpanes in a window (Chapter 6).

Stream

In the world of Smalltalk/V, streams are pathways for information, used to handle input and output. The class **Stream** contains methods that define such things as the current position of the read/write operation in the stream, adding information to itself, or returning some portion of its contents to a requesting object, and relocating the position of the read/write operation.

You can define streams to be read-only (using instances of the class **ReadStream),** write-only (with **WriteStream)** or read-write (with **ReadWriteStream).** This last class, in turn, has subclasses for files **(FileStream)** and the display **(Terminalstream).**

Although we make occasional use of various kinds of I/O streams in other places, the main discussion of streams in Smalltalk/V takes place in Chapter 10.

StringModel

The class **StringModel** plays a key role in text editing as it acts as the holder for the text being edited as well as providing some of the most common editing behavior to be applied to that text. We look at this class and its behaviors in Chapters 10 and 11.

3
The First Project: A Prioritizer

Introduction

In this chapter, we will build our first Smalltalk/V project. Applying the lessons learned in Chapters 1 and 2, we will design and construct a small application, make it accessible from the System Menu and from the Demo Menu, and test it. Then we will see how to make more effective use of the Smalltalk/V environment by consolidating some code. As part of this project, we will also learn to understand more of the capabilities of the Smalltalk/V debugger. We will work with the following classes in this chapter:

- **SortedCollection**
- **Prompter**
- **DemoClass**
- **ScreenDispatcher**

In addition, we'll create our own class called **Prioritizer.**

Project Overview

The project we will build in this chapter is called *Prioritizer*. It is a handy little program that lets you type in a list of items in any order, then helps you through the process of ranking these items from the most important to the least important (or from most expensive to least expensive, or by any other ranking criterion you want to use). In other words, rather than using the conventional "greater than" or "less than" comparisons of mathematics that are built into some of Smalltalk/V's classes, the Prioritizer uses the *user* as the comparison operator,

We encounter decisions in virtually every aspect of our lives. You've probably heard the expression, "Everything is a trade-off." The Prioritizer application will help you focus trade-off decisions between or among two or more alternatives by forcing you to look at each possible alternative's relative value when compared to other choices.

Designing the Project

Viewed at its simplest level, the Prioritizer application converts an instance of the class **Collection** to an instance of the class **SortedCollection.** We have placed some additional design constraints on it so that we end up with a design that requires that the finished application have several components. Specifically, we need to build components to:

- prompt the user for individual items to be prioritized
- ask the user to rank pairs of elements in the list, answering for each pair the question "is this one greater than (more important than) this other one?"
- display the resulting sorted list
- make the Prioritizer application accessible from the System Menu
- make the Prioritizer application accessible from the Demo Menu

Clearly these last two are not mandatory. But we do need some way to keep the Prioritizer application in the environment and accessible to us when we want it. We are going to discuss how to add it to both menus so that you can see what the process of doing so looks like. Then at the end of the chapter, we're going to put the Prioritizer application into a more logical, permanent home, and in the process remove some unnecessarily duplicated code.

Building the Project

We will begin the construction of the Prioritizer project by building some of its key elements in the Workspace and testing them iteratively. When we have these pieces working, we can then copy and paste them into a single text pane as we define the Prioritizer as a method in its own right.

Let's work with the tasks in the order in which we listed them above. This means that we will first build a Smalltalk/V routine to get the user to give us a list of items to prioritize. We'll need a way to ask the user a question,

get the user's response, and add the response to a growing list of items that we can later sort according to the user's instructions.

In this case, the classes we will use derive quite naturally from our description in the previous section of what the program should accomplish. (This is not always the best way to approach a Smalltalk project, as we will see in Chapter 4. In fact, this is a fairly procedural approach. Later projects in this book will take a more object-based approach to design.) We need to find Smalltalk/V classes that let us pose questions to the user and accept the user's responses. Looking through the Smalltalk/V Class Library diagram of Chapter 2, you can probably pick out a likely candidate or two. As it turns out, we will use an instance of the class **Prompter** for the user interaction.

Class Prompter

A Smalltalk/V Prompter is a small window with one **TextPane** that poses a question to the user in its top line, or header, area, and allows the user to type in an answer in the bottom. Figure 3-1 shows a typical Prompter.

Read the brief description of the Class **Prompter** in the manual. You will quickly be able to see that it contains three different class methods for prompting the user for a response:

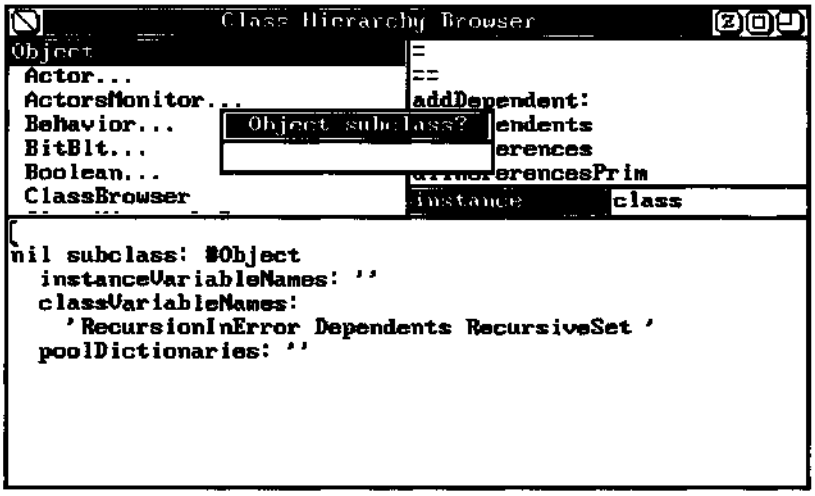

Figure 3-1. A Typical Prompter

- prompt: default:
- prompt:defaultExpression:
- promptWithBlanks:default:

Since we don't want Smalltalk to evaluate (or compile and execute, which is the same thing) the user's answers (we just want the strings), we can eliminate the second of these methods from our list of candidates. The first and third alternatives are nearly identical. A careful reading of the descriptions reveals that the *promptWithBlanks.'default:* method is going to keep any leading and trailing spaces the user inadvertently types into the prompter's window. Since that's unimportant here, we'll use the simpler *prompt .'default:* method instead.

So now we know we'll handle the user interaction via an instance of the class **Prompter** and that we'll use its *prompt .'default:* method to get the user's list of items to prioritize.

There is very little else to the **Prompter** class, so we can leave it for the moment and go on to the next class for building our Prioritizer application.

Creating a Prompter

Open a new Workspace in your environment and type the following code into it:

```
| anotherItem |
anotherItem := Prompter prompt:
    'Something to prioritize? (or leave blank)'
    default: ''.
^anotherItem
```

Select all of this text and select **show it** from the pane menu. The result should look something like Figure 3-2.

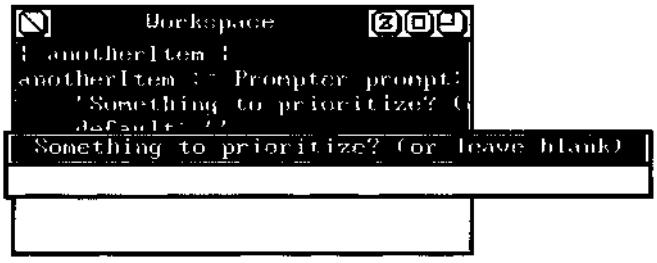

Figure 3-2. Creating a Prompter in the Workspace

When you enter some text into the Prompter and press Enter, Smalltalk/V will display your entered text in the Workspace. (This points out an important aspect of Smalltalk programming. We can execute code fragments to help us understand a problem or choose an approach to its solution.)

Let's take a quick look at what this code fragment does.

The first line sets up a local variable called *anotherltem* in which to store the user's entry. We chose that name because we know that in the final application, the user will be entering more than one item and we wanted to make the code readable.

The next expression uses the *prompt:default:* method of the class **Prompter** as described above to put the question into a Prompter window with no visible default answer.

The last line of code returns the value of the user's entry into the Prompter.

Returning the item entered by the user is obviously not our objective. We need to create a new object in which each of the user's entries can be stored. What kind of object should this be? It must be able to hold more than one object at a time, so it should be some kind of **Collection** (i.e., an instance of **Collection** or one of its subclasses). This class has three direct descendant classes: **Bag, IndexedCollection,** and **Set.** Since the order of the elements is not important during the process of collecting them from the user, we don't need a class that has indexed behavior already associated with it to store the objects.

Deciding whether to use an instance of class **Bag** or an instance of class Set is straightforward. Bags can have duplicated elements, sets cannot. In our case, if the user enters the same value into a list of choices to be prioritized, it is probably unintentional. So we want to make sure that the resulting list has only one entry for each choice the user wants to rank. We will therefore use an instance of class Set.

If we simply define a new local variable to hold the accumulating list of objects and then modify our earlier listing to include the ability to keep adding to this set, we will encounter an obvious problem: how do we get the process to stop?

As you can see, we have already anticipated that we need a way for the user to indicate that the last entry has been made. We chose to let the user simply leave the Prompter blank — i.e., press the Enter key without entering any text — as a signal that the list is complete. Now we need to add the code to handle this situation. This will be a simple conditional test. If the user's entry is blank, then we are done; until the user enters a blank, we keep processing by posing new Prompters and asking the user for yet another item to be prioritized.

Here is the complete code for constructing and displaying a set of choices entered by the user:

```
I  items anotherltem  I
items  := Set new.
[(anotherltem := Prompter prompt:
   'Something to prioritize? (or leave blank)'
```

```
    default: '')  = ''] whileFalse: [items add: anotherItem].
^items
```

Select all of this text and then choose **show it** from the pane menu. The result should look something like Figure 3-3 after you have entered a few answers and pressed the Enter key when you've finished your entries. Note that Smalltalk/V indeed returns a set. You might want to enter a duplicate value just to verify that a set performs as advertised.

Sorting the User's List

Since we will have more than one item in the list of objects to be sorted, we know that there's a good chance we will be dealing with one of the sub-classes of the class **Collection.** A perusal of the Class Library diagram reveals the existence of a class called **SortedCollection** that at least *sounds* like it should do what we want.

A quick reading of the description of this class proves our intuition to be right. Note that any instance of this class has access to a block of code known as the *sortBlock.* This block of code takes two arguments and returns the Boolean *true* if the first argument is higher in the sort order defined by the block than the second, *false* if it is not. Smalltalk/V defines a default sort block that sorts information in ascending alphabetical order.

Since this kind of binary comparison — "Is x > y?" — is exactly the kind of behavior we've determined we want in our Prioritizer, we have a clear-cut choice of class to use for this aspect of our project's behavior.

As it turns out, this is the only aspect of the class **SortedCollection** we need to use in our project. Later in the book, you will see that we make frequent and more extensive use of this class and its other powerful methods.

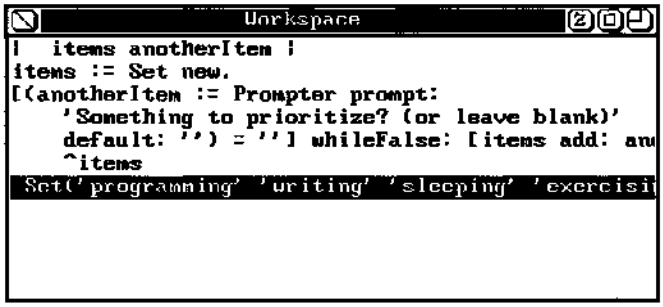

Figure 3-3. Set Returned by Prompter Interaction

Another interesting possibility might occur to you during the course of designing the Prioritizer application. You could simply add a prioritizer method to the class **SortedCollection** rather than making it a separate method in the classes whose menus we wish to use to access the method. But to do so would be inefficient because, as you can see from reading the description of the class **SortedCollection,** elements are sorted as they are added to an instance of this class. This would result in a complete sort each time an element was added to an instance of the class and another forced by the operation *of the prioritize* method. The user would be forced to compare some items over and over again. This is obviously a poor use of the user's time.

Now let's add the sorting capability to our code. To do so, we need to define a block of code called a *sortBlock* that will define how the items are to be sorted. This block of code will then execute for each pair of items in the set called *items* and return an instance of class **SortedCollection** that we '11 call *result.*

For our first attempt at defining this *sortBlock,* we'll use another method of the class **Prompter.** Note that it has a method called *prompt Default-Expression:* which, instead of simply returning the user's response as a string, evaluates it first. Since the *sortBlock* must return a Boolean value (its job, after all, is to compare two values in a collection for sorting purposes), we want Smalltalk to accept the user's input and evaluate it to a Boolean *true or false* expression.

In your Workspace, delete the last line of the previous code (the line that returns the value of *items),* and add a new local variable, *result,* to the list at the beginning of the code fragment. Now add the following code after the block of code following the *whileFalse:* method call:

```
result: =SortedCollection sortBlock:
    [:a :b | Prompter prompt:
        'Does  ', a printString,
        'have a higher priority than ',
        b printString,  '?  (Enter true or false) '
        defaultExpression: 'false'],
result addAll: items. ^ result.
```

(We'll take a look at *fas printString* method in a moment.)

Select the text and **show it.** Enter two or three items to sort and then press the Enter key to signal the end of the list. Now when you are asked if an item is greater than another item (see Figure 3-4), you can either press the Enter key to leave the answer *false* or you can select the default answer and type *true.* (Be sure you use all lower-case letters in either case or an error will result.) When you have supplied answers for all of the pairs of values you entered, Smalltalk/V will return the value of *result* in its sorted order, according to your rankings.

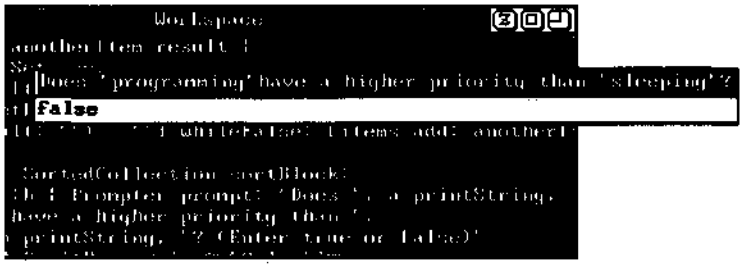

Figure 3-4. Prompter Asking for Ranking

The language used in the prompter is not very natural. We would expect to answer "Yes" or "No" to the question, not "true" or "false." But the *sortBlock* must yield a Boolean result and in Smalltalk/V, a Boolean can have only two values: *true or false.* We could let the user enter other answers and then translate them into the Boolean value to which they correspond, but that would require several additional lines of code. Since we plan to change this user interface later, we won't do that.

(Asking the user to type in an answer of *true or false* is not very elegant. It is also not in keeping with Smalltalk/V's design. What we really want to do is allow the user to click on a response in a different kind of prompting mechanism. Later in the chapter, we'll fix this problem. For now, we simply want to get the Prioritizer working. This is quite often the order in which you will carry out your Smalltalk/V program design and construction.)

Displaying the Result

There are a number of places we can display the results of the execution of Smalltalk/V programs. Perhaps the most common, particularly in a simple application like the Prioritizer, is the Transcript window. We put text and symbols into the system Transcript window by sending strings to the special system global variable called, appropriately enough, *Transcript.*

As a convention, we often use the technique of *message cascading* we discussed in Chapter 2 to send a series of messages to a particular object, in this case the Transcript.

Recall that *result* contains an instance of class **SortedCollection** when we are finished sorting it according to the user's instructions. How do you display the contents of an object of this type? Because Transcript is an instance of the **TextEditor** class, you should look up this class in the Encyclopedia of Classes in the Smalltalk/V documentation and determine how to display elements in an instance of this class. Doing so reveals that the method *nextPutAll:* is the likely candidate. Note, though, that this method

requires that it be passed a string as an argument. We have defined *result* to be an instance of the class **SortedCollection.** How are we going to convert it to a string?

The answer turns out to be simple but not so easy to find by exploring the Smalltalk/V class library. Every member of the class **Object** — which is to say, every object in the environment, since all objects are ultimately descended from the class **Object** — knows how to respond to the system method *printString.* This method is defined in the description of the class **Object** as answering a string that is the ASCII representation of the receiver. Applying this method to *result,* then, returns a string that is the ASCII representation of that collection.

Add the following code to the end of your Workspace:

```
Transcript cr; show: 'Your priorities are:' ; cr;
nextPutAll: result printString.
```

This code fragment displays the value of *result* in a way that is not very acceptable. We can take advantage of the fact that the class **Collection,** of which **SortedCollection** is a sub-class, understands the iterative operator *do:.* So we can have Smalltalk show each element of the sorted list on a separate line by changing the last line of the above code fragment so that the entire Transcript-displaying portion of the code looks like this:

```
Transcript cr; show: 'Your priorities are:' ; cr.
result do: [ :element | Transcript show: element; cr] .
```

The Finished Prioritizer Method

Now that we've built and debugged this new method, we need to give it a name and find it a home. We'll call the method *prioritize.* For the sake of completeness, and to allow you to double-check your code before we proceed, here is the complete listing of the Prioritizer method as it has been defined:

```
prioritize
    "This method collects a bunch of items into an instance
    of Set, then lets the user sort them, placing the result
    into an instance of SortedCollection."
    I anotherItem items result |
    items := Set new.
    [ (anotherItem := Prompter prompt:
        'Something to prioritize? (or leave blank) '
       default: '')='' ] whileFalse: [items add: anotherItem].
    result := SortedCollection sortBlock:
        [:a :b | Prompter prompt: 'Does ', a printString,
```

```
            'have a higher priority than ',
            b printString, '?'
            defaultExpression: 'true'].
    result addAll: items.
    Transcript cr; show: 'Your priorities are: ' ; cr. result
    do: [ :element I Transcript show: element; cr] .
```

Now we just need to have a place or places to store the code. Specifically, we need one or more classes to which to add this method. Since we have determined as part of our design that we want to be able to invoke the prioritizer from the System menu and from the Demo menu, we will add this method to the appropriate classes to make this happen. The next section of this chapter describes how to do this.

Adding Prioritizer to the Menus

Adding the ability to call this application from the System and Demo menus does not require not sub-classing an existing Smalltalk/V class, but rather modifying existing Smalltalk/V code. (Note that, as a rule, directly modifying the library sources in Smalltalk/V is *not* a good programming approach. This is because any changes you make here will have system-wide effects. However, because of the way Smalltalk/V implements the menus with which we want to work, this is the only reasonable means we have of modifying their behavior.)

Be sure to keep your Workspace intact while you complete this chapter. It might be a good idea to save your image at this point. We'd recommend that you then re-label the Workspace (call it "Prioritizer" or something else you'll associate with this program), then collapse it out of your way. During initial program development, many Smalltalk programmers find they keep several Workspace windows around their desktops.

Demo Menu Modification

Let's first add our prioritizer application to the Smalltalk/V 286 Demo menu. This gives us a convenient place from which to show off our work to our friends and it turns out to be a convenient way to learn how to work with existing menus. Then when we move to adding the Prioritizer to the System menu, we'll have very little new skills to master.

To create any menu, we simply create an instance of the class **Menu** and give it at least two arguments: a list of labels, or values, to display; and a corresponding list of selectors, or methods, to execute. These lists have a

one-for-one correspondence. Optionally, we can divide the list of labels into groups with lines by supplying a *lines:* argument.

Open the Class Hierarchy Browser and select the class **DemoClass.** Notice that it has an instance method called *demoMenu.* We're going to modify this menu so that it shows our Prioritizer and runs it when the user picks the new menu option.

Adding the Prioritizer to the menu is simple. Here is what the *demoMenu* method looks like before we begin working on it:

```
demoMenu
    "Answer the menu for the receiver. " ^Menu
labels: (' exit\dragon\mandala\multi mandala\ ',
    ¹ multi pentagon\multi spiral\bouncing ball') withers.
    lines: #(1 4)
    selectors: # (exit dragon mandala multiMandala
        multiPentagon multiSpiral bounceBall)
```

It is important to note that there is a direct correspondence between the order of the labels in the menu and the order of the selectors to which they correspond. The first label, *exit,* invokes the first item in the list of selectors, which happens to have the same name (but it could have had another name). To add our new method to this class, we '11 just modify the contents of the argument of the labels: keyword, adding our label at the end. Then we'll add the name of our method to the array of selectors associated with the menu. We'll also add a dividing line between the *bouncing ball entry* and our *prioritize* entry by adding the number 9 to the *lines:* array. Edit the code in the CHB to look like this:

```
demoMenu
    "Answer the menu for the receiver." ^Menu
labels: (' exit\dragon\mandala\multi mandala\ ᶠ,
    'multi pentagon\multi spiral\bouncingballYprioritize') withers.
    lines: #(1 4 9)
    selectors: # (exit dragon mandala multiMandala
        multiPentagon multiSpiral bounceBall prioritize)
```

Save this code and then run the demo by selecting *run Demo* from the system menu. Notice that the Prioritizer now appears as its own menu item at the end of the Demo menu, separated from the *bouncing ball* option by a line (see Figure 3-5).

To make the Prioritizer available to the class **DemoClass,** you'll need to store its code in the class. To add it to the class, follow these steps:

Figure 3-5. Modified Demo Menu

1. Select the Workspace in which you've stored the code temporarily.
2. Select and copy all of the code in the Workspace.
3. Open the CHB and select the class **DemoClass.**
4. Select *new method* from the method list pane menu.
5. Select all of the text in the editing pane and then choose *paste* from the pane window. (Note that you should not use the Backspace key here because doing so will cause the deleted text to replace the code stored on the Clipboard.)
6. Choose the *save* option from the pane menu.

The prioritizer can now be run directly from the Demo menu.

System Menu Modification

Before we can modify the System Menu to show our Prioritizer project, we must find out how the System menu gets activated. As it turns out, this requires some detective work.

You might first try bringing up the System menu and then pressing Ctrl-Break to force the Debugger to open. This should give you some idea where you are in the system at the moment. But you'll find that this approach yields no useful information (for reasons that will become clear in a moment).

Another way to locate a method that you are having trouble tracking down is to see if you can find something that is called by the method. We already know that the demo menu is activated from the System Menu, so let's open the CHB on the class **DemoClass** and look at the *demoMenu* method. Select

it in the CHB and then choose *senders* from the pane menu. You will notice that there is only one method that sends the *demoMenu* message and that method is *run,* also in the class **DemoClass.** Select the *run* method and again examine its *senders.* The result is shown in Figure 3-6.

As you can see, only one of these senders is of interest. The class **ScreenDispatcher** is the only one that sends the *runDemo* method, so we can now be sure that this is where the System Menu itself is coded. Using the Class Library diagram, locate the class **ScreenDispatcher.** Open the CHB on this class and browse through the instance methods. You won't find anything that sounds particularly useful. Click on the "class" button beneath the pane containing the list of methods and take a look at the class methods. There are two. The first one that sounds promising is *systemMenu.* Select this method and you can see that it's not very helpful. It only has one line and that line refers to something called a *ScreenMenu.*

So let's look at the other class method, *initialize.* (We do this because, after all, the system menu is around from the startup of Smalltalk/V, so it is just possible that it is set up in the process of initialization.) Sure enough, there's some code that looks like it might be helpful. Here's the entire method:

```
initialize
    " Private - Initialize the system menu."
    I aString I
   BackupWindow
    ifTrue: [
        aString := 'dos shell\speed/space\exit
SmalltalkXbrowse disk\open workspace\browse classes\redraw
screen\save image\run demo[1] withers]
```

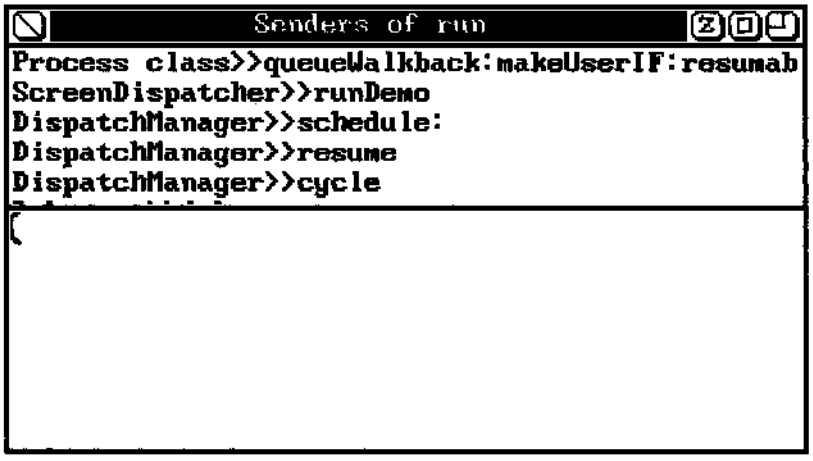

Figure 3-6. Senders of *run* Message

```
    ifFalse: [
        aString := 'dos shell\speed/space\exit
Small talk\browse disk\open workspace\browse classes\redraw
screen\save image\run demo' withers] . ScreenMenu := Menu
labels: aString lines: #(3 7 9)
        selectors: #(dosMenu speedSpace exit openDiskBrowser
openWorkspace openClassBrowser redraw save runDemo)
```

Now the *systemMenu* method makes more sense. It simply returns the *ScreenMenu* created by this *initialize* method.

From our experience with the Demo menu, we can already see what we have to do. Just add the *label prioritize* at the end of the *aString* assignment statements, and after the *runDemo* entry in the *selectors: array.* Save the code.

Now we need to copy the *prioritize* method from the Workspace into the class **ScreenDispatcher** as a new method. Follow the same steps as above when we copied it into the class **DemoClass,** changing the class with which you work. Finally, save the method.

Before you can run this new method of the class **ScreenDispatcher,** though, execute the following line of code from the Transcript or a Workspace:

```
ScreenDispatcher initialize.
```

This causes the **ScreenDispatcher** to re-initialize itself, which is necessary because we need to add the new menu, which is something **ScreenDispatcher** does only when it is initialized since this menu normally doesn't change during processing.

You can now demonstrate the prioritizer from either the System menu or from the Demo menu. The basic functionality of the prioritizer is now set and we could go on to the next chapter at this point. But we promised some final clean-up, so let's take care of that before we move on.

Sprucing Up the Application

We will do two things to clean up the *Prioritizer* application before we consider ourselves done with it for the moment. First, we'll change the somewhat ugly true-false prompter with which we ask the user to rank each pair of choices in the list. Second, we'll remove the necessity of having two copies of the code around in our image by creating a new class for the Prioritizer application and removing the old code which will no longer be needed.

Changing the Prompter

We used an instance of the class **Prompter** to create both of the user interaction components of our prioritizer. The problem is that a prompter can only do one thing: pose a question and let the user type in an answer (or accept a supplied default response). We'd like to allow the user to click on the words "Yes" and "No" as the means of providing information about a given choice that we've placed into the Prioritizer.

Although prompters only allow the user to type a response, there is a user interface component on which the user is accustomed to clicking to provide a response to the system. This object is, of course, the pop-up menu. Such objects are instances of the **class Menu,** which we will work with to create our new Yes-No confirmation interface for the user.

Everything else about the way we deal with asking the user to sort the list of choices remains the same.

We will create a new method for the class **Menu** that pops up a menu with two choices: Yes and No. We will call the method *confirm:*. It will take as an argument the question we wish the user to answer. Note that by not hard-coding this question, we make this a general-purpose method. That, of course, is a key objective of good object-oriented design.

Open a CHB on the class **Menu,** select the *Class* button, then *new method* from the pane menu in the method list pane, and type in the following code:

```
confirm: queryString
   "Returns Boolean true for Yes, false for No." |
label Imp tempMenu response | labelTmp := queryString,
..'\Yes\No*. tempMenu : = Menu
   labels: labelTmp withers
   lines: #(1)
   selectors:  (Array with: true with: false) .
^tempMenu popUpAt: Cursor offset.
```

In line four of this method, we concatenate the question passed to the method when it is called with the labels "Yes" and "No" to form the contents of the pop-up menu. Note that the question itself, contained in *queryString,* appears as part of the menu and, in Smalltalk/V, may be selected by the user. Because of that, we have to supply a selector for the top line of the menu even though it is unlikely that an experienced user of a mouse-and-menu-based system would actually choose the question as a response. The backslashes in the *labels:* argument are item separators that are converted to carriage returns when the string containing them is sent the *withCrs* message.

The *labels:* argument provides a string whose backslashes (\) were converted to carriage returns, and the next line places a dividing line after line 1 of the list of labels (i.e., after "Yes"). The *selectors:* argument provides a value of *true* as a result of the user selecting either the question or the first answer, "Yes," *an& false* as a result of selecting "No." Notice that we set up the selectors explicitly as an array. If we didn't do this, the method would not return the Boolean values *true* or *false,* but the symbols #true and #false. The *sortBlock* would not know what to do with a non-Boolean response and an error would result.

The last line of the method tells our *tempMenu* instance of class **Menu** to pop up where the cursor is located. The caret (^) before the statement makes this the return value of the method.

Now let's make a slight change to the *prioritize* method to use this new approach to the user interface. Move your CHB view to *the prioritize* method in the class **DemoClass.** Change the line that assigns the sorted list to the instance variable *result* so that it reads as follows:

```
result := SortedCollection sortBlock:
    [:a :b | Menu confirm: 'Does ', a printString,
        ' have a higher priority than ', b
        printString, '?'].
```

No other changes are required to the method. Save it and then make the same change to *the prioritize* method in the class **ScreetiDispatcher.** You can now test each of these methods. You'll see that the pop-up menu shown in Figure 3-7 now appears when you ask the user to sort the list of choices. The user simply uses the mouse to click on one of the responses. If for some reason the user clicks on the question, your method assumes the response is the same as "Yes." (You could, of course, either make this selection mean "No" or even have a different method that would execute if the user makes this choice. Our decision to make it an affirmative response is purely arbitrary. By changing the first element of the array in the *selectors:* argument of the setup of the new menu, you can have this selection do whatever you like.)

Consolidating the Code

In creating our prioritizer application, we have replicated this code in two different classes. We did this for the sake of simplicity in building our first project, but we would be remiss if we left it this way. So let's remedy that situation by creating a new class to consolidate this duplicate code. Then we will simply call this method from the classes **DemoClass and Screen-Dispatcher** rather than having the code itself reside in each of those classes.

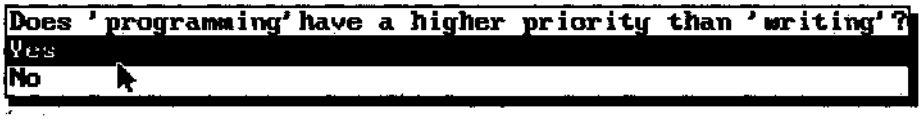

Figure 3-7. Modified Prioritizing Pop-up Menu

First, let's get the code for the *prioritizer* method that we have just created and copy it onto the clipboard. Open the CHB, select either the **DemoClass** or the **ScreenDispatcher** class, and then *the prioritize* method. Select all of the text of the method in the text editor window and choose *copy* from the pane menu. Now select the topmost class, **Object,** and choose *add subclass* from the pane menu.

You will be asked to name the new subclass of the class **Object.** Type in "Prioritizer" and press Enter. Next, you'll see a somewhat obtuse pop-up menu asking you to define the type of sub-class you wish to create. Don't worry now about what it means, just choose the "subclass" option. The class description needs no modification, so with the new class selected in the CHB, select *new method* from the empty method list pane (the upper right pane). A template for a new method appears in the text editing pane. Select all the text of this template and *choose paste* from the pane menu. (Be careful not to hit the Backspace key or you'll lose the data in the copy buffer.)

To be consistent with Smalltalk/V conventions, we're going to rename our method *run* so that it will actually sound like it does what we're going to ask it to do from the other classes from which it is called.

Select the method's name, *prioritize,* and change it to *run.*

Choose *save* from the pane menu. (Be careful not to hit the Backspace key or you'll lose the data in the copy buffer.)

With the method in place in its own class in the hierarchy, we can now modify the *prioritize* method in the two classes from which we call it: **DemoClass and ScreenDispatcher.** The process is the same for both classes.

Move to the class in the CHB, select *theprioritze* method, select all of the code in the text editing pane except the name, (and, optionally, the comment) press the Backspace key to delete it, and type in the following line of code:

```
Prioritizer new run.
```

This line of code creates a new instance of the class **Prioritizer** and then executes the *run* method we just defined.

(All classes know how to make new instances of themselves, so we don't need to define a *new* method unless we want to do something other than what it inherits for *new* behavior.)

Since we haven't reversed any of the changes we made to the menus in the **DemoClass and ScreenDispatcher** classes, we can still invoke the Prioritizer application the same way as before. Test it and prove it for yourself.

Using the Debugger, Part 2

In Chapter 1, we examined the use of Smalltalk/V's debugger. Since you'll be spending more of your time in the debugger than you probably want, we'll take a more adventuresome look at its use here.

You encounter one of the most trying aspects of Smalltalk/V programming and its debugger when you generate a walkback window that refers to a message you didn't send in your methods. Trying to sort through such bugs can be a frustrating experience. But if you understand that there are some kinds of errors with which the Debugger is not going to be terribly helpful, you can save yourself some frustration.

To demonstrate, open a Workspace (you can use the one we created earlier in this chapter or a new one) and type in the following code exactly as shown:

```
I anotherItem items I
items := Set new.
[anotherItem := Prompter prompt:
   'Something to prioritize? (or leave blank) '
   default: ' ' = '' ] whileFalse: [items add: anotherItem] .
^ items
```

Notice that we have omitted a set of parentheses inside the first block. In the *prioritize* method as we implemented it in this chapter, there is an opening parenthesis inside the square bracket and another just before the equal sign near the end of the block. This has the effect of forcing the execution of the *prompt:default:* expression before the comparison of the = operator. Select this code and *show it.* You' 11 get a walkback with the error message:

```
"isEmpty" not understood
```

Where did *that* come from? We didn't use the *isEmpty* method anywhere in our code. The first thing you'll probably do, then, is to choose *debug* from the walkback's pane menu. By stepping back through the stack in the Debugger, you will find the expression *aString isEmpty* highlighted in the text editing pane (see Figure 3-8).

The local variable list pane (the middle of the top panes of the Debugger window) contains an entry for *aList,* so select it. In the rightmost pane, notice that its value is the Boolean value *true.* How in the world did a string variable end up with a Boolean value? This is clearly the source of the problem.

Beyond this clue, though, the Debugger doesn't seem to offer much additional hope for solving the bug. We have to use some of our own logic here. Since the problem is the inappropriate use of a string (by having a Boolean value assigned to a variable designed to contain a string), let's first see where we use strings in our code.

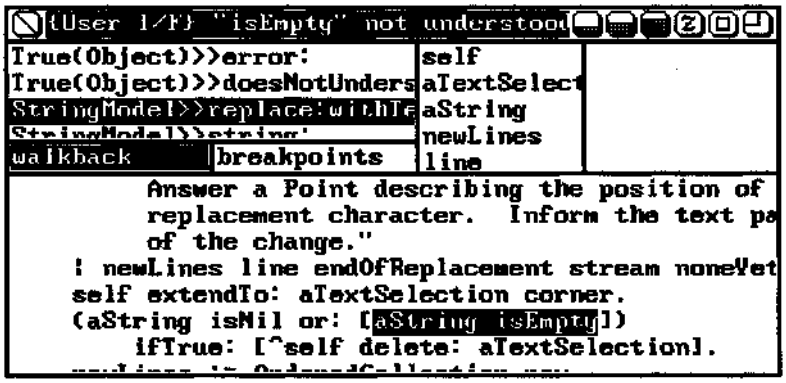

Figure 3-8. Debugger Highlights Offending Code

The prompt:default: method requires two strings, one for the question to be posed and one for the default. Examining our code, the first string appears correct. The null string associated with the *default:* argument also appears correct, but we notice that it *could* be interpreted to mean that this null string is equal to another null string. This would obviously produce the Boolean result *true,* so this part of the code becomes immediately suspect.

In other words, it appears that the key expression may be interpreted as follows:

```
[anotherItern := Prompter prompt: 1
   something to prioritize?' default:
   ("=")]
```

when what we really wanted is:

```
[(anotherItem := Prompter prompt: 1
   something to prioritize?' default:
   ")='']
```

Further examination reveals that we have indeed created a problem here. We don't want to compare the default value of the string with the null string; rather, we want to compare the user's response to the null string. The user's response is stored in the local variable *anotherltem,* so we clearly need to isolate the response from the comparison.

In technical terms, we have created an *order of operations* error. The equality (=) comparison operation was being carried out before the creation of the string to which the comparison was to be made. The result was a nasty little bug. Put the parentheses where they belong and the code fragment executes without error. (You can examine the order of evaluation by putting a self halt message in the beginning and then use the Debugger to single-step through the code.)

4
Programming Techniques

Introduction

In this chapter, we will take a look at high-level programming and design issues that will make your work in Smalltalk easier and more effective. We'll start with a discussion of why object programming in general and Smalltalk/V programming specifically "feel" different from conventional procedural programming. Then we'll discuss the "onion peeling" process that characterizes all computer programming to one degree or another and see how Smalltalk makes that process more robust and easier to manage. Next, we'll examine the two main means by which you will accomplish application development in Smalltalk/V: creating objects and adding methods.

Why Smalltalk/V Feels Different

People who approach Smalltalk programming with a background in other programming languages often express a sense of disorientation and even frustration on their first encounters with this new environment. "It's like another planet," some will say. "I just can't seem to get started" is another often heard complaint.

Our experience indicates that for the most part these comments and feelings reflect the fact that Smalltalk programming — and, in fact, object-oriented programming in general — really is qualitatively different from conventional programming. A short example will serve to focus our thinking on the fundamental reasons for this.

If you have a background in a conventional programming language like C or Pascal, for example, you might approach the process of learning a new implementation of your favorite language by trying to write a straightforward piece of code using the new tool. Let's say that you have a favorite sorting algorithm that helps you both learn the new tool and discover some

of its potential inherent weaknesses. So you write the code and test it. Conceptually, what you really do is you start with an algorithm (the sorting code) and you pour some data into it (the test data which will be sorted).

Now if you try to approach Smalltalk/V with that same approach, you're going to feel like you have indeed traveled to another planet. But if you think not about pouring data into an algorithm but instead focus on creating an object that is inherently sortable and then giving it the ability to sort itself, you'll be able to make the transition.

In other words, the emphasis is not on the separation of data and method but on the very opposite: their tight integration with one another.

Peeling the Onion

Solving any puzzle has at least a faint resemblance to the process of onion-peeling. You strip away outer layers of information, much of which serves only to distract you from your real objective. Then you reach progressively lower and deeper layers of the onion until finally you reach its core. Along the way, you find useful and some not-so-useful parts of the onion.

Programming is puzzle-solving. Regardless of the programming language being used, the hardware on which the program will eventually be run, or the program's ultimate purpose, programming bears a resemblance to onion-peeling. You begin with a broad understanding of what you want the program to accomplish and with a fundamental grasp of the tools available to attain that goal. Then through a series of successive approximations, you gradually hone your tools and your skills with them as you focus more intently on the problem. You identify small pieces that can be dealt with and you master them. (The order in which these processes occur may vary, of course, depending on the programming methodology you adopt. That doesn't make the work any less like onion-peeling, it just results in more or fewer tears in the process.)

Object-oriented programming is even more like onion-peeling than other kinds of programming. This is particularly true when you are working in the Smalltalk/V system. As we saw in Chapter 2, there are families of classes and sub-classes, sometimes extending as much as five layers deep. Associated with each class is a collection of instance and class methods and the code that makes the class and its instances behave as they do. Finally there's the source code, where you can read comments and the source listing itself to determine what the method does and how it does it. Sometimes, you have to take all of these steps to get to the point where you are ready to begin a Smalltalk/V programming assignment.

Beyond this organizational hierarchy, there is a behavioral or message-passing hierarchy that becomes visible any time you are trying to analyze what is going on in a system rather than just in one class at a time. We've seen how the Debugger in Smalltalk/V provides us with a list of currently executing methods. Because applications reuse existing components, messages are passed back and forth between objects you've created and objects that exist in the Smalltalk/V class library. Tracking through that maze can sometimes be quite a challenge. That's why Digitalk supplies so many helpful tools with Smalltalk/V, including the Debugger which is quite revealing when you know how to use it.

Boiled down to its essence, the onion-peeling in Smalltalk programming can be looked at as learning how to decide when to do what.

What we hope to do in this brief chapter is to help you peel the Smalltalk/V onion a little more efficiently.

Where to Begin?

If all programming in Smalltalk consists only of creating objects and associating methods with them, how do you know where to start? How can you determine what objects you need to create? How do you know how to divide up the methods among all the objects available to you? In short, how do you know what to implement?

There are, as you might expect, no easy and universal answers to these questions. But we can offer some general guidance that you will find useful again and again as you create Smalltalk applications.

The process of deciding where to begin with a Smalltalk programming task can be described as posing and answering four basic questions:

- What do I expect my application to do? (Or, expressed in more object-oriented terms, "What behavior do I expect my application to ex hibit?")
- What objects are needed to represent my application's components?
- What is each object's responsibility?
- What must each object "know" about itself?
- How do the objects in my application collaborate with other objects?

Let's take a brief look at each of these questions to glean some ideas for how to answer them.

What Should the Application Do?

Obviously, this question is one you have to ask yourself regardless of which language or programming methodology you're using. It is essential to good software design.

But in an OOP system like Smalltalk, the way you phrase the answer to this question can be useful. Think about describing your application's behavior in an English-language sentence (unless, of course, your native language is not English, in which case you should use a human language you understand!). It may take more than one sentence to describe your application if it is complex. That's fine.

When you have refined your sentence description of your application, you have already gone a long way towards constructing the class diagrams that we'll talk about shortly. The nouns in your sentence are candidates for objects to be created. The verbs are candidates for methods. For example, if you wanted to construct a Smalltalk application that would enable the user to click on some buttons to increment or decrement a counter, you might write a sentence like this:

This application consists of a counter that starts out with a value of zero and then lets the user change its value by clicking on buttons to increment or decrement it by one each time.

In fact, we'll build just such an application in Chapter 5. This sentence has the following nouns:

- application
- counter
- value
- user
- buttons
- time

Not all of these will be converted to objects as it turns out. But all of the objects we create in our application are on this list. Similarly, the verbs in the sentence are:

- consists
- starts
- lets
- change
- clicking
- increment
- decrement

Again, not all of these verbs will translate into methods in the finished application, but all of the methods that appear in that application are either on this list or grow out of it and an understanding of how user interaction is implemented in Smalltalk/V.

Obviously, you can go a long way toward a workable starting point for a design of a Smalltalk application with this simple approach.

Objects and Their Responsibilities

Once you've come up with a list of the candidate objects of which your application may be composed, you need to look at each object and ask yourself what responsibility it will have in the application.

Staying with our counter example, we know we're going to have a counter application object which retains a numeric value and displays that value to the user, updating the display each time the value changes. So we can define an object whose responsibility is to re-display the value each time it changes. The object also obviously needs to know how to add and subtract values from the value.

Other objects might include buttons whose behaviors consist of being able to detect when they have been the subject of the user's mouse-click and sending some information about that fact to another object. You can probably surmise, too, that we'll need a window of some sort to hold all of these various objects.

In this simple application, there is a clear line of demarcation among objects. That is not always the case. When it is not, you will spend some time during the design of your application looking for common or similar behaviors among various objects and combining them into other classes.

What Do Objects Need to Know?

Each object will probably have to have some information about itself so that it can carry out its behavior. The counter object, for example, can't increment or decrement its value unless it knows the value at all times. These things that an object must know about itself provide a starting point for a list of instance variables to be defined for the object.

Sometimes these lists are quite long, but most often an object only needs a few pieces of information about itself to behave as expected.

How Do Objects Collaborate?

You don't build Smalltalk applications in a vacuum. Indeed, it could be argued (and often is) that a key difference between programming in an OOP environment like Smalltalk and developing programs in more traditional systems and languages is the holistic nature of the application. Your objects interact with other objects in the system even when this interaction is not entirely obvious to you.

Any time you use a system method, subclass an existing Smalltalk/V class, or engage in activities that are characteristic of the Smalltalk/V system (such as menu usage or window creation), you are using classes and objects that you did not create and of which you may be only vaguely aware, if you are aware of them at all.

Particularly when it comes time to create a run-time version of your application, you need to be aware of these interactions and collaborations if you are going to be successful in completing your application.

Starting With Class Diagrams

The whole process of programming in Smalltalk/V becomes much easier when you approach it systematically. Our personal favorite methodology for doing this is the *class diagram.* (Class diagrams are the invention of David A. Wilson, who used the technique for several years in teaching OOP before writing the article, "Class Diagrams: A Tool for Design, Documentation, and Testing" in the January/February 1990 issue of the *Journal of Object-Oriented Programming.)* A class diagram (see Figure 4-1 for a sample) defines a class name, the instance variables associated with it, and the methods or behaviors it must have to support its responsibilities.

Creating and modifying the diagram is often facilitated by thinking about the problem you are trying to solve. Attempt to come up with a written or verbal description of the problem. When you have done so, identify all of the nouns in the sentence(s) that describe the problem. These are candidates for objects. Now identify all the verbs; these are candidates for methods.

Not every element needed in every application will lend itself to this approach, of course, and you'll have to experiment with and modify it for yourself, but it may serve to help you get the design process started in a reasonably usable direction.

When you've created a rectangle in the class diagram for each object you've identified, you are ready to start finding common behaviors and creating a class or class hierarchy to support these common threads. When you find you've created two or three or more objects that all seem to need to be able to do the same thing (even if they are using different algorithms), group them together. Consider giving them a common ancestry by creating an abstract class (a process we'll discuss shortly).

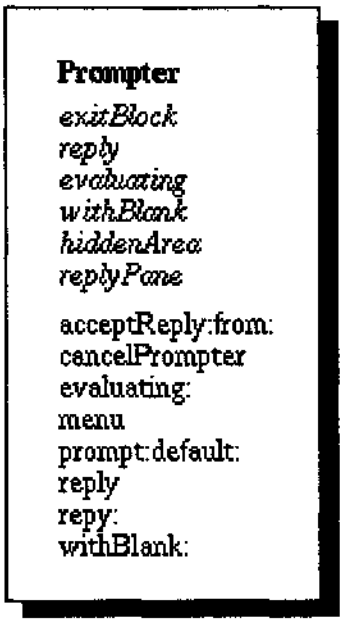

Figure 4-1. Sample Class Diagram

Creating Objects

As we have said, all Smalltalk programming consists of extending an existing Smalltalk class library by creating new classes, and adding methods to these classes.

When you are refining the behavior of a class or extending its capabilities, or adding to its "knowledge" of itself (i.e., instance variables), it might be time to create a new class. When you create a new class, the first thing you have to do is decide which class to subclass, since all new classes you create are going to be subclasses of some Smalltalk class, even if it is the class called **Object.**

Subclassing the Class Object

Your first impulse will probably be to define your new class as a subclass of the top Smalltalk/V class, **Object.** Every class that comes in the Smalltalk/V class library is a descendent of **Object** and since every class has to have a parent, or "superclass", this is the easiest way to approach the need for subclassing.

Classes created as subclasses of **Object** inherit a good bit of behavior, though it is fairly generic behavior. Most of the instance methods of the class **Object** relate to object management (see Table 4-1) or provide basic, generic behavior that your objects will typically override or supplement. (Don't worry if many of the terms and concepts in the table are unfamiliar to you; we will discuss them throughout the book where they are appropriate.)

Table 4-1. Object-Management Methods of Class Object

Instance Method	Purpose
allDependents	Identify all of the objects that are defined as "dependents" of the receiver
allReferences	Identify all references to the receiver in the system.
become: class	Change the identity of an object.
copy	Identify the receiver's class Copy
deepCopy	the receiver
doesNotUnderstand:	Copy the receiver and make copies of all of its instance variables
error:	Used in error handling Used in error
halt	handling Force a Walkback window from
implementedBySubclass	a method
inspect	Initiate a Walkback because a subclass doesn' t implement a message that it should implement
isKindOf:	Open an inspector on the object
isMemberOf:	Test whether the receiver is an instance of a particular class or of any of its sub-classes
isNil	Test whether the receiver is an instance of a particular class
notNil	Test whether the receiver is nil
release	Test whether the receiver has any value other than nil
respondsTo:	Disconnect all dependents of the receiver from the receiver
shallowCopy	Determines if the receiver responds to a particular message (inherited by the re-ceiver or implemented in its class)
species	Copy the receiver without copying its instance variables
yourself	Return a class that is similar to (or the same as) the receiver
	Answer the receiver

Other behavior associated with the class **Object** involves such activities as printing or displaying the object's value, support for the dependency mechanism (see Chapter 5), and forcing the sending of messages to the object.

Subclassing Other Classes

If you can find a class in the Smalltalk/V class library that exhibits some of the behavior you desire or that at least provides some kernel of your desired functionality, you can create a subclass of that class rather than **Object.** Obviously, this means that the newly created subclass inherits all of the behavior of the class you've chosen and all of its superclasses including **Object.**

For example, if you want to create a new class that contains behavior for dealing with a new type of aggregate of elements (such as a new dictionary with special capability), you would probably look first to the class **Collection** or one of its subclasses to see which one seems to offer the most support for the behavior you wish the object to exhibit. Having identified such a class, you can create your own subclass of it using the method outlined below.

The Subclassing Process

The step-by-step process for creating a subclass of an existing Smalltalk/V class, whether the class came with the Smalltalk/V library or is one you added, is quite simple:

1. In the CHB, find the class you wish to subclass and select it.
2. From the class list pane popup menu, choose *add subclass*.
3. Give the new class a name in the prompter that appears.
4. For most classes, you will need to tell the system (see Figure 4-2) whether the subclass is a normal subclass, a variableSubclass or a variableByteSubclass. The first type contains pointers to named instance variables. The second contains pointers to both named and indexed instance variables. The third contains bytes and is seldom used.
5. Make any modifications needed to the class definition template in the text editing pane.
6. Select *save* from the text editing pane popup menu.

The new class is now selected and ready for you to add your first method, following the steps outlined in the preceding discussion. From this point on, the new subclass is treated exactly like any other class in the library.

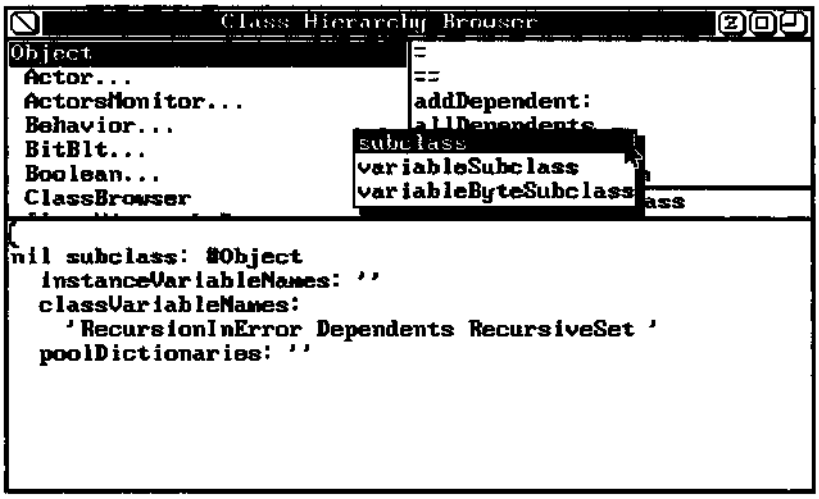

Figure 4-2. Defining Subclass Type for Class Object

Modifying Behavior of Chosen Class

Subclasses usually override one or more pieces of inherited behavior (which is the primary reason they are created). To make this process of subclassing accessible, you need to understand how to select and modify methods. These methods will fall roughly into three categories: those you should ignore, those you should consider modifying, and those you should replace (or override) completely.

- Ignore methods unrelated to the functionality you are adding to the system via your new subclass and that don't need to deal with any instance variables introduced by the subclass.
- Extend methods that are useful as far as they go but which need supplemental behavior added to them, perhaps because of new in stance variables introduced by the subclass.
- Replace methods whose names and behavior you wish to use but whose implementation is not appropriate for your class.

Those methods that fall into the second category are dealt with by the simple expedient of defining a method of the same name in your subclass, carrying out the supplemental activities, and then sending the message to the superclass version of the method with a line like this (assuming the method you're overriding is called *doMethod):*

```
...your new code...
^super doMethod
```

The effect of this line is simply to tell Smalltalk/V, "Use the *doMethod* method found in the superclass of the receiver." In other words, Smalltalk/V will carry out your special processing and then use the inherited version. It is also possible, of course, to execute the superclass method first and then carry out the specialized processing in your method of the same name. But you must understand the implications of this. Including this code ensures a consistent return value, but it may in some circumstances have the effect of entirely undoing the work your method is designed to accomplish. As a basic rule, you should be sure to return a consistent value in keeping with the inherited version of the method you are overriding.

If the behavior embodied in the method in the superclass is just not appropriate to the new class you're defining, simply define a new method to override the behavior exhibited by the ancestor class(es). This includes disabling behavior completely; if there is a message in the superclass that you simply do not want your objects to respond to, you can't just ignore its existence in the superclass. Instead, you must define a method that does nothing (or something harmless and invisible) and give it the same name as the method you wish your class to ignore. Note that if you find yourself with a great many methods that fit this description, you may wish to re-think your design. You may have not chosen the best class to subclass for your project.

Creating and Using Abstract Classes

An "abstract" class in Smalltalk is a class that is not intended to have instances. It exists for the sole purpose of grouping together related behaviors that will be exhibited by objects created from its "concrete" subclasses. But that whole definition is a bit too abstract, so let's take a look at a specific example.

The class *Collection* is an abstract class. You would never create a new object and call it an instance of *Collection.*

An abstract class is a repository for behavior that is shared by all instances of all its subclasses. In the case of the class *Collection,* for example, behavior that all collections need to know how to do is defined. These activities include such things as:

- adding, deleting, and retrieving elements of the collection
- iterating over all of the members of a collection, executing a block of code for each element
- displaying or printing itself in some useful way

Many of these methods as they are defined in the abstract class **Collection** either do nothing or return an error indicating that the method should be implemented by a subclass. For example, the method *add:* in the abstract class **Collection** might do nothing but send the message *implementedBy-Subclass* to the receiver *(self)* and return the result. In other words, the *add:* method is so specific to the type of collection that we cannot generically implement behavior that will apply to all types of collections. (Of course, many methods in abstract classes do implement real behavior.)

The Purpose of Do-Nothing Methods

You might be tempted to ask, "If these methods don't do anything or, worse yet, generate errors if I don't implement them in my subclasses, then why are the methods defined at all?" The answer relates to the general purpose for which abstract classes exist. We've said that abstract classes exist only to gather together behavior that is common to all their subclasses. Another, perhaps more illustrative way of saying this is that abstract classes define "policies" (i.e., what all subclasses and their instances must know how to do if they are going to be members in good standing of the abstract class) while concrete classes (i.e., all those that are not abstract) define the "mechanism" for carrying out the policy.

Seen in this light, the *add:* method in the class **Collection** tells you as a Smalltalk/V designer that if you create a new subclass of this class, you must be sure that it knows how to respond to the *add:* message. It sets a policy: all members of this class know how to deal with the message *add:*. It is up to your class to implement this method any way it wants (including, of course, ignoring it by defining an *add:* method that does nothing). You cannot ignore it without generating a Walkback.

Identifying the Right Class

A difficult part of programming in Smalltalk by these last two approaches we've examined is identifying the class to subclass — whether that class is abstract or concrete. How do you go about this?

There are no hard and fast answers. This is part of Smalltalk programming that lends itself to a great many stylistic differences among programmers. We know Smalltalk developers who apply the following techniques:

- examine abstract classes for desired behavior, then critically analyze a candidate abstract class' subclasses

- use the Method Index in the Smalltalk/V documentation to find a method that sounds like it does something they'd expect the class for which they are looking to know how to do and then examining that class' definition in the Encyclopedia of Classes in the same manual
- make it a subclass of object and, after implementing some of the application, move it as appropriate
- ask other programmers in their group or circle of colleagues

Ultimately, there is no substitute for a solid working knowledge of what's in the Smalltalk/V class library. One of the main purposes of this book is to familiarize you with the most often used and important elements of that library so that you have a base from which to begin your own explorations.

You can rarely expect to find a class that is coded with all of the behavior you want. (If this does happen, think how easy your job has become!) Look for a class that has a reasonable amount of behavior dealing with things you expect your class to understand. When you find such a class, dig in and start programming. If your search doesn't yield perfect results, you can build your own class and later make it a subclass of a class you later identify as useful.

In Chapter 5, we'll take a close look at a practical example of how to approach this problem.

Adding Methods

The simplest approach to programming in Smalltalk is to find a class that makes a good "home" for the behavior you're trying to add to the system and simply define a new method for that class. You saw this method used in Chapter 3 when we created *the prioritize* method and added it to the classes **DemoClass and ScreenDispatcher.** This process is the easiest because:

- You don't have to know very much at all about the class involved and its overall behavior. In fact, you could even just grab an arbitrary class and stick the new method into it if it was a stand-alone method (i.e., one that didn't depend on that object's other behaviors). We should note that this would *not* be good programming practice, but it illus trates the point that you need not always have an in-depth understand ing of a class before adding a new method to it.
- Adding a new method is a relatively simple process. We'll provide a step-by-step instruction list shortly, just for reference.
- It is easy to "undo." Thus you can use it to experiment with a new method approach without committing yourself to careful tracking of potential changes to the image.

The Process of Adding a Method

Here is the process of programming in Smalltalk/V by adding a method to an existing class:

1. Identify the class to which the new method should be added. Ideally, you'll find a class which contains at least some related functionality and which may be in a "good" position in the hierarchy (i.e., its subclasses would make good use of the method as well).
2. Find the class in the CHB and move to the method pane.
3. In the method list pane, choose *new method* from the popup menu.
4. In the text editing pane, enter the source code for the new method. You will generally do this, of course, simply by editing the template Smalltalk/V provides for you. But on occasion you will have previ ously typed the method code into the Transcript or a Workspace window. In that case, you can simply copy it from that window and then select the entire method template in the text editing window of the CHB and paste the copied method over the template.
5. Select *save* from the text editing pane's popup menu.

That's all there is to the process. Your new method can now be invoked by a message being sent to the class or an instance of the class.

Avoid Adding Methods to System Classes

Adding a method to an existing class is so simple and straightforward, you might wonder why you just don't handle all of your Smalltalk/V programming this way.

If you are adding a new method similar to one which already exists, select the existing method, edit its name to the new method name, and then edit and save the code. When we made our first pass at implementing *the prioritize* method in Chapter 3, for example, we simply added it to two classes. It did not affect other objects in those classes in any significant way, only altering their menus (which is a relatively unintrusive change). Similarly, it did not depend heavily on those classes and their other methods.

It is also wise to consider creating a new method by adding a method to an existing class only if the new method is going to be added only one place. In Chapter 3, when we had implemented *the prioritize* method in two different places, we concluded that we should consolidate the code. To do so, we abandoned the strategy of simply adding the method as a functional piece

of code to two different classes, opting instead to create a new class to hold the method and then sending the *prioritize* message to that class from the other places in the system where we wanted the functionality available. However, this was feasible because we did not need to inherit anything from these two classes. Inheritance sometimes forces the duplication of code. In addition, if a new method needs to be added in several places, it may be an indication that it is being inserted at the wrong position in the class hierarchy.

Finally, you should probably not decide to undertake programming by adding methods to existing classes if you are going to have to add several new methods to one or more existing classes to make your design work. That's probably an indication that you'll be able to build a more specialized object, so you should create a new class to hold all of the related pieces of behavior. This is not, however, to say you should never build an application by adding many methods throughout the system. That approach is sometimes appropriate; it just isn't often the right technique from the standpoint of reusability.

In other words, before you decide to create a new bit of functionality for your Smalltalk environment by adding a new method to an existing class, make sure it is singular and isolatable. If that's the case, you're probably safe handling the task by addition of a method. Otherwise, this is not the most effective or robust means of Smalltalk programming.

5
The Second Project: A Simple Counter

Introduction

In our first project, the Prioritizer of Chapter 3, we added methods to existing classes as our primary programming method. At the conclusion of that process, we saw how to create a new class with a single method and then use that class as we would any other supplied in the class library. It would, of course, be somewhat unusual for us to have a Smalltalk/V application that consisted of one class with one method.

This chapter extends our work with Smalltalk/V as we build another application, this time beginning by creating a new class. The application is purposely kept small so that we can concentrate on the details of design and programming methods that you can use on all of your Smalltalk programming projects rather than on the details of how the application itself works. Though small, the project is functional and it forms the kernel of something that could be far more useful.

In this chapter, you'll also learn how to use an Inspector to observe what is happening in a running application.

Project Overview

This project involves creating a counter. Recall from Chapter 4 that we described this project in one sentence that served there as the starting point for our design. We said the project would consist of a counter that starts out with a value of zero and then lets the user change its value by clicking on buttons to increment or decrement it by one each time. Figure 5-1 is a sketch showing how the finished application should look. If the user clicks on the portion of the pane that looks like a button labeled "increment," the counter will increase the value by 1. If the user clicks on the "decrement" button, the counter will decrease the value by 1.

That's all there is to the project. Let's dive into seeing how we might design it.

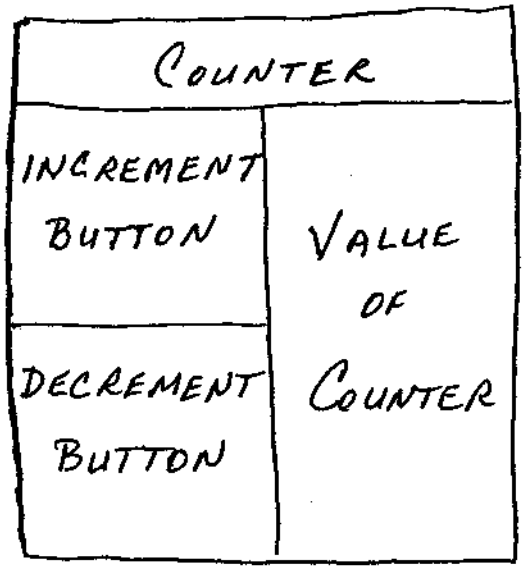

Figure 5-1. Sketch of Counter Project

A Quick Overview of Model-Pane-Dispatcher

The heart of all interactive Smalltalk/V applications lies in understanding the interactions of windows and their elements *(panes* and *subpanes* in Smalltalk/V parlance) through dispatchers and models. This trio is collectively referred to as the model-pane-dispatcher paradigm, or MPD for short. In this chapter, we will examine the concepts involved in MPD in just enough detail to build our application. Chapter 6 provides an in-depth look at this important topic.

We begin with a brief overview of MPD. If you are already comfortable with the basic concepts of MPD from your work with the Smalltalk/V tutorial, you should feel free to skip this section.

A windowed application in Smalltalk/V typically involves the use of three classes (or groups of classes):

- the model class, which defines the application object and which you can almost always think of as the application class
- **the Pane** class
- **the Dispatcher** class

Generally speaking, you can focus your concerns on creating the model class and with supporting the **Pane** class protocol, which is the secret to your application's interactivity.

The Model

The purpose of the model is to represent the underlying information depended upon by the other objects in the application. Normally, this consists primarily of the graphic user interface associated with the application.

The model class is also responsible for the activities listed below. You can see why we said earlier that the application and the model are almost always the same object. These are tasks you normally associate with an application:

- saving the current state of the window and the application, generally by tracking instance variables
- creating new panes
- initializing the contents of panes
- carrying out communications between panes and synchronizing the activities of all the panes in a window
- defining optional menus for the panes

One significant issue that appears when we talk about the model class is the concept of dependencies. Various elements of your Smalltalk/V application are dependent on other elements. For example, in our counter application, what is displayed as the value of the counter is dependent on what the user has instructed the application to do to that value. Because of this dependency relationship, we establish a connection between the buttons that the user clicks on to change the value and the pane in which the value is displayed.

The Pane

You will make extensive use of the behavior of the Pane class in your Smalltalk/V applications, so you must understand how panes and subpanes work and interact. Smalltalk/V provides a complete set of pane-building blocks and you generally use them just as they are supplied. The class **Pane** has two immediate subclasses: **TopPane and SubPane.** An application window always has one and only one instance of **TopPane.** Its purpose is

to control the actions of all of its subpanes in relation to the window. (If an application has more than one window, as we will see in Chapter 6, each window has an instance of **TopPane.)** There are three types of subpanes available in Smalltalk/V:

- text subpanes that support the display of and allow the user interaction with text
- list subpanes that display (optionally) scrolling lists of one or more items
- graphics subpanes (GraphPanes) which best support graphical infor mation

Unless you are defining a new type of pane (which we will demonstrate in Chapter 6), you will generally not need to do anything with these classes other than to create instances of them with specific characteristics peculiar to your program.

In our counter application, we will see that we have the requisite **TopPane** and two subpanes, one to handle the button-like behavior with which the user interacts and one to display the current value of the counter.

The Dispatcher

Every pane and subpane has a unique instance of a dispatcher associated with it. You don't need to do anything to create this instance and for all practical purposes you can ignore it. Its only purpose is to gather input events (i.e., keyboard and mouse actions) from the user and pass them on to the pane that should process them.

Generally speaking, the only interaction you have is with the **Dispatcher** associated with the **TopPane** in your window and even then all you do is send it a message to tell it to start working. Since this is "canned" code, you don't even need to understand what is happening at this point unless you are simply insatiably curious about it.

What's Really Going on Here?

This whole MPD interaction discussion may seem a bit mysterious to you. In fact, you may be tempted to accuse us of arm-waving, of saying, in essence, "This is top hard, so just ignore it and trust us."

This behind-the-scenes activity is desirable and supportive, but until you get used to the idea and learn to trust Smalltalk to do exactly what it says it will do, you can get a bit paranoid about using it. As we hope you'll see

before this chapter is over, writing interactive Smalltalk/V programs is easy once you gain a little trust in the system.

Designing the Project

This is a single-window application. That window consists of three components:

- a button-like object for incrementing the counter's value
- a button-like object for decrementing the counter's value
- a place to display the current value of the counter

Before we can begin writing the code for this project, we need to answer the following questions:

- How can we divide up the responsibilities of this application among the model, pane, and dispatcher portions of the design?
- What type of user interface objects support our desired button-like behavior?

Dividing the Responsibilities

From the sketch we made of the application window (see Figure 5-1), we know that the subpanes and their automatically associated dispatchers are going to be responsible for detecting the user's click on a button-like object, and for displaying the value of the counter after any change. The model, then, is responsible only for two basic tasks:

- It must keep track of the value. For this purpose, it is relatively apparent we should use an instance variable. All classes are capable of storing values in instance variables.
- It must coordinate the activity between the button-like objects and the pane where the value is displayed. Since MPD behavior is defined in the class **Object,** all classes know how to handle these tasks.

Essentially, then, we've seen that we don't require any particularly specialized behavior of the class we are going to create for our counter application. Because of this, we do not need to look for a class to subclass. Instead, we can create our new class as a sublcass of **Object.**

Defining the Glass

We begin, as we generally will, by defining a new class with which to associate the behavior and the application. To do this, activate or open the CHB, select the top-level class, **Object,** and then select *add subclass* from the pane's popup menu. When you are asked for the name of the class, answer **Counter.**

Now edit the class definition appearing in the text pane in the CHB. Merely add the instance variable *value* to the definition. Save the class definition.

You are now ready to begin adding methods to your new class, starting with the method that will define the main window for the application.

Defining the Main Window

Our Counter application will have a single window with two subpanes. The main window, as we have already seen, will be defined as an instance of **TopPane** and will be given certain characteristics in the process. Here is the basic code involved:

```
aTopPane := TopPane new
    model: self; "self will be a Counter because code is in Counter"
    label: 'Counter'; rightIcons: #().
```

In almost every case that you'll experience as a Smalltalk/V programmer, the model associated with a pane—whether a **TopPane** as here or a SubPane as we' 11 see in the next section—is the application object that also initializes the application. Using Smalltalk/V notation, we refer to this object using the Smalltalk special variable *self.* This simply means in this case that the Counter application is the model with which the pane will interact.

It is optional to put a label on a **TopPane.** On the other hand, subpanes do not have labels.

You can request that Smalltalk/V 286 put icons to the left or right of the window label with the methods called *leftlcons:* and *rightlcons:*. By default, Smalltalk/V 286 puts a close icon in the upper-left corner of the window and three icons — one for zooming, one for collapsing, and one for resizing — in the upper-right corner of the window. In our case, we've decided not to deal with the issues of redrawing the window if the user resizes or zooms it, so we've supplied an empty array as an argument to the *rightlcons:* message. Because we haven't specifically supplied any *leftlcons:* message, the window will, by default, have a close box icon.

As we'll see in Chapter 6, there are other things we might frequently want to do as part of the process of defining the **TopPane** for our application model window. But in this case* what we have described will suffice.

Defining the Subpanes

We need two subpanes, as we've already seen. One subpane has the responsibility for displaying the current value of the counter. The other subpane allows the user to interactively change the value of the counter by adding or subtracting 1 from the counter's current value. This latter subpane could consist of a single pane containing both *increment* and *decrement* commands, or buttons, or it could consist of two separate panes, each of which would handle one of the commands.

Let's work on the first subpane. It has a simple assignment, so it shouldn't be too hard to implement. Of the three types of subpanes we have available, it should be clear that we don't need a **ListPane** since there is only one value for the counter at any given point in time. That leaves us with a decision between a **TextPane and a GraphPane.** We could actually choose either one as it turns out. There are advantages and disadvantages to each.

A TextPane will be simpler to work with primarily because placing information into such a subpane is much more straightforward than using graphic-related commands associated with the **GraphPane.** On the other hand, using a **TextPane** means we can't stop the user from editing the contents of the window, though we could still protect the value of the counter if the user did make a direct change to this subpane's contents. This would not be good user interface design in a real-world application but since creating such an application is not our intent here, that is not a significant factor in the decision.

On the other hand, using a **GraphPane** makes the value displayed uneditable. But displaying information in such a pane requires more careful understanding of graphic procedures, which are inherently more complex than methods for displaying text.

In the interest of simplicity, then, we'll create the part of the window that displays the counter's value as a **TextPane.**

Here is the code we'll use to create the **TextPane** that will hold the value of the counter. An explanation of each line follows the listing.

```
aTopPane addSubpane: (TextPane new model: self;
    name: ttvalues; framingRatio:  (3/4 @ 0 corner:
    1 @ 1) ) .
```

We have previously created the instance of **TopPane** called *aTopPane.* Here, we send this instance the message *addSubPane:* along with an argument that describes the subpane to be added. This description consists of four lines. The first creates a new instance of **TextPane** and the remaining three lines are cascaded messages sent to this instance. The second line defines the pane's model to be the Counter application by using the keyword

self as described earlier. The third line gives this new pane a name, *values.* (The pound sign before the name causes Smalltalk to treat the word "values" as a symbol rather than as a variable. Its omission would result in an error.) This name gives us a label by which to refer to this pane later when we want to update it. It also acts as the message selector the pane will send to the model to get the data to display in this subpane. The final line tells **aTopPane** what portion of its total area to allocate to this new **TextPane.** We are allocating 1/4 of the total area by starting this window's horizontal position 3/4 of the way across the width of the **aTopPane.** (Don't worry at this point about how this calculation is carried out and what it means. We'll spend a good bit of time in Chapter 6 discussing this issue.)

In the other subpane we want to create we will put the commands from which the user will select what to do with the value of the counter. As we said earlier, we could implement these button-like objects as two separate subpanes or as one subpane containing both commands. In the interest of keeping the design of the window as simple as possible, we'll use a single pane to hold the button-like objects labeled *increment* and *decrement.* Since this subpane will hold multiple objects and since we want the user to be able to select either (but not both) of them, this subpane should be a **ListPane.**

Another reason for selecting **ListPane** as the type of subpane to carry out this responsibility is that this type of pane exhibits the behavior we need. Look at the **ListPane** class in the *Dictionary of Classes* in the Smalltalk/V documentation; it is described as having button-like behavior. The class behaves in such a way that, "when one of the strings in the collection is selected, either the selected string or its index in the list is passed to the application model for further processing." Sounds like what we need, doesn't it?

Here is the code that we'll use to create this new subpane. An explanation follows the listing.

```
aTopPane addSubpane: (ListPane new model: self;
    name: #cmds; change: #cmds:; framingRatio:  (0
    @ 0 extent:  3/4 @ 1) ) .
```

The only part of this subpane definition that is different from the **TextPane** subpane we saw earlier is the addition of the fifth line, which defines a *change:* method for this subpane. We obviously need such a method, since we want the user's interaction with this subpane to modify the contents of the **TextPane.** This links the subpane and the model more closely together and requires that we define a method that we can use to notify the text pane of changes in this subpane. The argument to the *change:* message is the selector for the message that the **ListPane** will send to the model when the **ListPane** is clicked.

Displaying the Window

Now that we've defined both of the subpanes that make up the window as well as its required **TopPane** instance, we're ready to ask Smalltalk/V to display the window. This requires a standard line of code, which looks like this:

```
aTopPane dispatcher open scheduleWindow
```

The *dispatcher* message is sent to the aTopPane. This returns the identity of the pane's dispatcher. This dispatcher is then sent the *open* message, which opens and activates the window but does not yet display it. It can then receive and process the *scheduleWindow* message, which shows the window on the display and makes it active. So in this one line of code, we open the window on the display and make it the currently active window. (Again, we'll go into the specific functions in this line of code in Chapter 6.)

Creating a Single Method for Window Definition

Generally, your Smalltalk/V applications will have a single method that takes care of defining and creating their windows. By convention, this method is called the *open* method. (Optionally, it may be called *openOn:* when you wish to pass an argument to the method when you open a new window.) We can assemble all of the code we've written so far into this method, which will then appear as follows:

```
open
    "Create a two-pane window for the Counter application" I
    aTopPane I
   aTopPane := TopPane new
   model: self; label:
   'Counter'; right Icons:
   #(). aTopPane
   addSubpane: (ListPane
   new model: self; name:
   #cmds; change: #cmds:;
   framingRatio: (0 @ 0 extent: 3/4 @ 1)) . aTopPane
   addSubpane: (TextPane new model: self; name:
   #values;
           framingRatio: (3/4 @ 0 corner: 1 @ 1)).
   aTopPane dispatcher open scheduleWindow
```

(You may have noticed that we reversed the order of the **ListPane and TextPane** portions of this method. In this particular case, the order in which they appear is insignificant, but the code is somewhat more readable if the panes are defined in the order they appear in the window, left to right and top to bottom.)

Writing the Methods for SubPane Interaction

As a result of the code we have written to create the Counter's application window, we can see that we need to write several other methods:

- *cmds,* where the list of commands that will make up the contents of the ListPane will be defined
- *cmds:,* which will contain the code for the method that notifies the **TextPane** when the contents of the **ListPane** have changed (i.e., when the user has made a selection in the **ListPane)**
- *values,* where the **TextPane** can get text to display

Just by their descriptions, the code that makes up these methods is fairly easy to envision.

The *cmds* method must simply define what the command options will be. Here is the code for this method:

```
cmds
    "Answer the list of valid commands" ^#
    (increment decrement)
```

This method simply returns an array that contains two symbols that are the labels to appear in the **ListPane.** If you ever wanted to change the labels in the counter, then, all you'd have to do is change this method.

Similarly, the *cmds:* method will translate the user's selection (which we'll pass to the method as an argument) into the appropriate method to be invoked. Here's the code for this method:

```
cmds: userSelection
    "Perform the command selected by the user"
    Self perform: userSelection
```

If the user selects the word *increment* in the **ListPane,** then the message *cmds: increment* will be sent to the Counter application. It will then invoke this method and discover that it should send the *increment* message.

Finally, the *values* method simply handles the display of the current value of the counter. Since we are using a **TextPane** for this purpose, we need to tell the counter's value to print itself in a way that the **TextPane** can understand. A reading of the description of the class **TextPane** in the *Encyclopedia of Classes* reveals that it deals with an indexed collection of strings. The value contains an integer value, so we have to find some way to display it as a string.

Fortunately, Smalltalk/V's **Object** class contains a method called *printString* which all objects understand. This method returns a string representation of the receiver. So the method to display the current value of the counter is simple:

```
values
    "Answer the value of the counter in the TextPane"
    ^value printString
```

We still don't have any real interaction between the two subpanes. The *cmds:* method sends either the *increment* or the *decrement* message, where the actual updating of the value of the counter takes place. These methods should, in turn, notify the Counter application (or model) that the counter's value has changed. Let's look at these two methods at one time:

```
increment
    "Add 1 to the value of the counter"
    value : = value + 1. self changed:
    lvalues

decrement
    "Subtract 1 from the value of the counter"
    value  : = value - 1. self changed: #values
```

The final line in each of these methods takes care of "broadcasting" the fact that the value of the counter has changed. It sends to the model (represented by the keyword *self)* the message *changed:* with the argument lvalues. This message, inherited from the class **Object,** tells the model to notify all dependents (e.g., the **ListPane)** that something has changed. Subpanes inherit behavior that tells them to refresh themselves from the model if the argument is their name. As we have seen, we associated this name with the **TextPane,** so that pane responds to that change. The dependents are notified by sending them an *update:* message with the argument *lvalues*.

Methods to Create a Counter

We have now written methods to create a two-pane window in which the value of a counter is displayed, and in which the user can click on button-like objects (actually, entries in a **ListPane)** to add or subtract 1 from that value. We've also coded the interaction between the two panes.

The only thing left to do now is to provide a way for the user to create and initialize a new Counter object. This will require two methods: one which we'll call *new* that will allow the user to create a new Counter object and one called *initialize* that will give that Counter object its initial value of zero.

In the *Method Index* in the Smalltalk/V 286 documentation, you can tell at a glance that the method called *new* is supported by every class in the Smalltalk/V system. Examine the places where it is implemented and you'll find in every case it is a class method rather than an instance method. Think about that for a moment and the logic will be apparent. You can't send the *new* message to an object and ask it to create itself because the object doesn' t exist until you create it. You see the problem. The reason for this is that a class is really a factory for creating instances. The *new* method requests that a class create an instance of itself. So we'll define *new* as a class method (also referred to in some Smalltalk literature as a *meta-method)* in the class **Counter.**

To do so, just click on the *class* button in the CHB and then select *new method* from the method pane popup menu. Then enter the following code in the text pane:

```
new
    "Create a new counter"
        ^ super new initialize
```

We call the *new* method of the class **Object** by sending the message *new* to the object *super,* which always holds the name of the receiver class' superclass. We do this because we are redefining the *new* method but we need the basic behavior of the *new* method to do so. Then we send the *initialize* method to this newly created object. We haven't yet coded the *initialize* instance method, of course, so let's do so now. Here's the code:

```
initialize
   "Initialize the value of the counter to 0"
   value := 0
```

The first time the **TextPane** is displayed on the screen, it queries the model for text to display. Because the model (our application) is initialized to 0, that is what the **TextPane** displays. (Note that if we didn't initialize *value,* we would not be able to add 1 to it later — a fairly common mistake.)

It is interesting to note that we could have accomplished our purpose here without writing a *new* class method if we had made the *open* method a class method instead of an instance method. Then we could have avoided having to send two messages to create a new **Counter** object. But the approach used in the text makes the creation, manipulation, and initialization **of a Counter** object clearer than combining some or all of these functions into one class method. As you gain more experience as a Smalltalk programmer, you may wish to circumvent this clarity in the interest of efficiency of coding.

Testing the Counter

It seems like we've defined everything we need to create, initialize, and manipulate a new counter object, so let's test it. To do so, we're going to use another technique that you'll find useful as you build Smalltalk applications. We could simply test this application by typing the following line into the Transcript of a Workspace:

```
Counter new open
```

But then every time we want to test the behavior of a Counter, we'd have to type the same text again. Furthermore, when someone else comes along and wants to create a new Counter, they'd have to rummage around through our methods and, by inspection, figure out how to invoke our application. Good programming technique dictates that we find a better way.

We recommend that for all applications you build, you create a class method called *example* that will create and initialize an instance of the object. Go ahead and do this now for the class **Counter.** (If you can't remember exactly how to create a class method, go back and review how we did it when we built the *new* method, above.)

Here's the code for this method:

```
example
    "Demonstrate the creation of a new Counter"
    Counter new open
```

Now go to the Transcript and type:

```
Counter example
```

The result should probably look something like Figure 5-2. If you click on the labels *increment* and *decrement* a few times, you'll see that the value of the counter displayed in the **TextPane** changes accordingly.

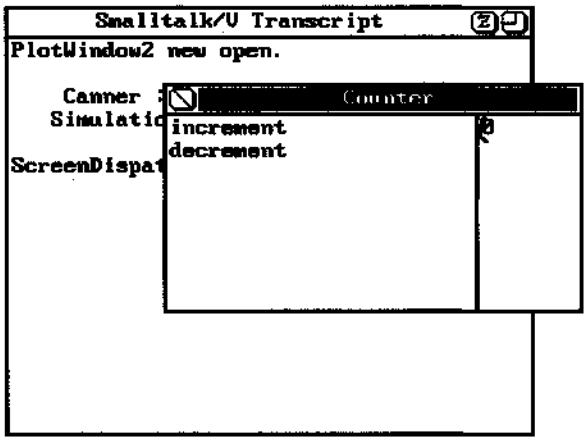

Figure 5-2. A New Counter Object

If you will follow this practice for all of your applications and other classes (other than abstract classes), you'll find that your projects become self-documenting to some degree. Anyone who wants to know how to create a particular object can simply find the name of the object and send it the *example* message.

Making the Window Smaller

There is only one problem with this application: the window is far too large for the application. This is because we're letting Smalltalk/V use its default initial window size for the window rather than taking control of that process. Let's do a little investigating and see if we can figure out how to make the window initialize itself to a more reasonable size.

Since the **TopPane** is responsible for the display and management of the entire window, let's take a look at its methods in the CHB.

Select **TopPane** in the CHB (you may have to open the **Pane** class to do so) and scroll through the method names. Nothing sounds like its job would be to set up an initial window size, does it? Where else could this value be set? How about the class **TopDispatcher** since all instances of class **TopPane** will have such an object associated with them? Select **TopDispatcher** in the CHB and scroll through the method names. Sure enough, there's a method called *initWindowSize*. Select it and note the code:

```
initWindowSize
    "Private - Answer default initial window extent"
    ^Display extent * 2 // 5
```

Without going into details on what the *extent* of a window is, suffice it to say that this method opens a window that occupies approximately 2/5 of the screen's total area.

While you're looking at this code, copy it and then go back to the class Counter and create a new instance method. Select all of the text in the text pane and choose *paste* from the popup menu. Now experiment with the fraction (2/5) until you get a size you like. We found that 1/5 was a good size with which to work, but you should feel free to refine that. At 1/5, the window looks like Figure 5-3.

It may not be clear to you why this process of creating an *initWindowSize* method worked. The **TopDispatcher** uses the default *initWindowSize* method when it encounters a pane that does not respond to that message itself. Since **Counter** had no such method, its **TopDispatcher** was simply using the default window size supplied by that method. By overriding that behavior with a custom version of the method, we overcame the limitation of that method and gave our application a window size with which we were happy.

This concludes the creation of our Counter project. Considering that this application creates a multi-pane window with which the user can interact, overrides the default system size for a newly created window, updates its contents on demand, and goes away when asked, it didn't take much code to create it, did it?

Now let's look at the promised new debugging aid before we wrap up our work in this chapter.

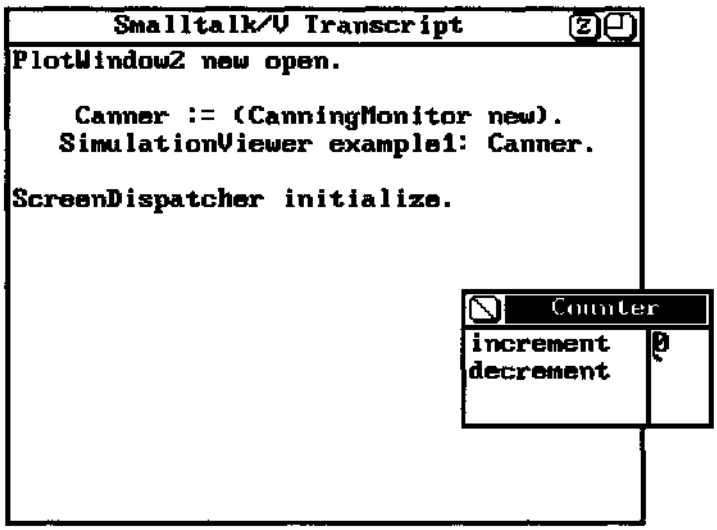

Figure 5-3. Smaller Initial Window for Counter

Inspecting a Running Counter

There will be many times in your experience as a Smalltalk/V programmer when you' 11 want to run an application while you watch what happens to its instance variables. Sometimes, this will enable you to spot an insidious bug faster than you can without such a capability. Other times, it will simply give you insight into code perhaps written by another person.

To do this, you need to open an Inspector on the application. The problem is, you can't usually open an Inspector except from the Debugger or by typing a command into the Transcript or a Workspace. While a program is running, these options are not readily available. So we trick Smalltalk/V into creating a Walkback by inserting a *halt* method at an appropriate point. To see this in action, follow these steps:

1. In the CHB, select the class **Counter** and then select its *initialize* method.
2. Change the *initialize* method to read as follows (note the period after the first line):
   ```
   value := 0.
   self halt "for demo debugging purposes only"
   ```
3. Now save this modified form of the *initialize* method and go back to the Transcript.
4. Type *Counter example* in the Transcript or a Workspace. When the counter begins to appear, you'll see a Walkback labeled "halt encoun tered."
5. Select *debug* from the popup menu in the Walkback. You'll be shown the usual Debugger window.
6. Select the label *self in* the receiver pane (the middle pane across the top of the window) and then select *inspect* from that pane's popup menu.
7. An Inspector opens on the Counter object that is presently executing. Position and size it conveniently.
8. Go back to the Debugger window and select *resume* from the frame popup menu.
9. The Counter window appears, with the value of 0 in the **TextPane** as expected.

You can now use the *increment* and *decrement* selections in the **ListPane** to manipulate the value of the counter. Any time you wish to inspect the state of the program, just select the Inspector window and choose any of the instance variables in the left **ListPane.** Its value will then appear in the right pane.

With the **Counter** object, this is not terribly revealing or exciting, but the technique is one you'll undoubtedly want to remember.

Be sure when you are finished with your inspection experiment that you remove the *halt* from the *initialize* method. Removing the period at the end of the remaining line is optional but good form.

Removing the Counter Class

Hopefully, you've found the process of creating, testing, modifying, and debugging the Counter class helpful and enlightening. However, unless you intend to build something more interesting using it as a base, you probably want to remove the class from your image before you save the image again. Instructions for removing a class are in Chapter 1.

If you wish to save your work somewhere for later review but don't want it cluttering up your Smalltalk/V image, you should file it out. Simply select the class name Counter in the CHB's class hierarchy list pane, then choose *the file out* option from the pane menu. Smalltalk/V creates a text file called COUNTER.CLS that you can later read, print, or even edit and file back in if you wish.

The Complete Counter Project Listing

In every chapter where we present a full program for you to create and execute, we'll reproduce the entire listing of that application at the end of the chapter. This gives you a central place to look at the code, compare it to your work, and retype it if you come back later after having removed the class.

```
Object subclass: ^Counter
    instanceVariableNames: 'value'
    classVariableNames: ''
    poolDictionaries: ''

"Class method"
new
    "Create a new counter"
    ^ super new initialize

[11] Instance methods"
cmds
    "Answer the list of valid commands"
    ^# (increment decrement)
```

```
cmds: userSelection
   "Perform the command selected by the user"
   self perform: userSelection

decrement
   "Subtract 1 from the value of the counter"
   value : = value - 1. self changed: #values

increment
   "Add 1 to the value of the counter"
   value : = value + 1. self changed:
   lvalues

initialize
   "Initialize the value of the counter to 0"
   value := 0

ini tWindowS i z e
   "Private - Answer default initial window extent"
   ^Display extent * 1 // 5 "Your value may vary here. "

open
    "Create a two-pane window for the Counter
    application" I aTopPane I
   aTopPane := TopPane new
   model: self ; label: '
   Counter'; rightlcons: # ()
   . aTopPane addSubpane:
   (List Pane new model:
   self; name: #cmds;
   change: #cmds:;
   framingRatio: (0 @ 0 extent: 3/4 @ 1) ) .
   aTopPane addSubpane: (TextPane new model: self;
   name: #values;
           framingRatio: (3/4 @ 0 corner: 1 @ 1) ) .
   aTopPane dispatcher open scheduleWindow

values
   "Answer the value of the counter in the TextPane"
   ^value printString
```

6
The World of MPD

Introduction

As we saw in Chapter 5, the model-pane-dispatcher (MPD) triad is central to Smalltalk/V programming. In fact, it is safe to say that you will probably never write a Smalltalk/V application that doesn't include the use of MPD. This chapter takes a closer look at MPD in an attempt to answer the following questions:

- Of all of the classes and methods involved in the classes that make up MPD interaction, which ones are essential to understand and use, and which can you safely ignore?
- How can you identify and use the protocols of MPD to promote pluggability?

Our goal, then, is to perform a sort of triage on the MPD world so that you can concentrate your focus on the important points and set aside what otherwise looks like a hopelessly complex portion of Smalltalk/V programming. Our experience has been that many Smalltalk/V programmers stumble over MPD as they attempt to implement their first serious application.

This chapter will take a look at the various methods you'll need to understand before you can apply MPD design and interaction to your Smalltalk/V applications. It provides the groundwork for understanding MPD in general and the work we'll do in Chapter 7 in particular.

There's So Much Going on Here!

Smalltalk/V window interaction involves instances of three families of classes:

- your own application class (the model)
- **Pane**
- **Dispatcher**

A quick perusal of the *Encyclopedia of Classes* in the Smalltalk/V 286 documentation reveals that just these last two classes involve 15 classes with a combined total of nearly 175 instance methods, to say nothing of dozens of instance variables and class methods.

Let's see if we can cut this forest down to a manageable size. We'll focus, by the way, on non-graphics applications (i.e., those that don't draw anything on the screen other than windows and their subpanes). In Chapter 8, we'll take a look at how to construct a graphic application in Smalltalk/V and then we' 11 add a bit more complexity to the amount of MPD with which you should become familiar. But since many applications are fundamentally non-graphic once the windows and menus are accounted for, you may find yourself never needing the graphic-related MPD classes and methods.

TopPane Methods You'll Need

Of the more than 20 methods included in the class **TopPane,** you need to be aware of only seven. In fact, you will often use only three of them. The three most often used are:

- *new*
- *label:*
- *addSubpane:*

The other four important **TopPane** methods are:

- *minimumSize:*
- *initWindowSize*
- *rightlcons:*
- *leftlcons:*

Let's take a look at each of these methods in turn.

The *new* Method

The *new* method in the class **TopPane** is actually inherited from the class **Pane.** While most of the methods we'll examine in this chapter are instance methods, *new* is a class method. In essence, it says to the class, "make a new instance of yourself."

Only very rarely will you do anything to this method. It will be adequate for your needs; all you have to do is to send the *new* message from your application to the **TopPane** class. This is how you create a new window for your application.

You will almost always assign the return value *of new* to a temporary variable declared in the method where the *new* message was sent to **TopPane.** This gives you a way to send to the instance the messages that add subpanes and handle other window processing. In most cases, you'll probably name the method that contains the **TopPane** *new* expression either *open* or *openOn:* depending on whether it has any arguments associated with it.

The *label:* Method

Every window in Smalltalk/V may have a label associated with it. If you define a label for the **TopPane** of the window, the label will appear centered in the title bar of the window. If the title is too long for the title bar, it will be truncated on the right and as much of it as possible will be shown in the title bar. The argument passed to the *label:* method should be a string (i.e., a series of characters enclosed in single quotation marks). Here is an example of its use:

```
label: 'Counter'
```

Given that the label is optional, you may wonder why we placed it among the most common methods for you to learn. It is quite unusual to create a Smalltalk window without a label. There are occasions when you wish to do so, but you should generally provide a label for the window in the absence of a very strong design reason for not doing so. The user may want to know the purpose of the window long after it's been opened. An unlabeled window is clearly less identifiable than one that is clearly labeled as to its purpose.

The *addSubpane:* Method

A TopPane without any subpanes can do only what an instance of **TopPane** is capable of doing. In other words, it does nothing of particular interest. Even a single-pane window must have at least one subpane. You add subpanes to a **TopPane** with the *addSubPane:* method. Since you generally create these subpanes as you insert them into the **TopPane,** you should not be surprised to find out that the general way this process works is exemplified by this code fragment:

```
a topPane addSubpane: (ListPane new. . . arguments)
```

Of course, you'll use whatever temporary variable you've used to hold the name of the TopPane instance in place of our *aTopPane* and you'll substitute the kind of pane you want to create where we've used the name of the ListPane class. But the format is the same regardless of those two differences. A quick perusal of the various windows defined in the Smalltalk/V environment reveals that every use of the *addSubPane:* method appears more complex than we've just described. This is because the newly added SubPane is generally sent a series of cascaded messages to give it the desired characteristics. We'll get to some of the most important of those messages in the next section when we discuss the important implementation aspects of subpanes.

The *minimumSize:* Method

Any window that can be resized by the user should define a minimum size to which the user will be allowed to shrink it without collapsing it completely. If you don't define this method in your TopPane description, Smalltalk/V will use the default, which defines a window's minimum size as 24 by 32 pixels.

You will often want to define a window's minimum size in terms of the font(s) being used in its pane(s) while keeping in mind the minimal amount of information that the user will find useful as he interacts with the window. For an example of this, select the class ClassHierarchyBrowser in the CHB itself and examine its *openOn:* method. Use the text pane popup menu to choose *next menu* and then *search* and find the reference to the *minimumSize:* method. You should see something like this:

```
minimumSize: 20 * SysFontWidth @ (10 * SysFontHeight)
```

Here Smalltalk/V defines the minimum size of the CHB's window to be no narrower than 20 times as wide as the width of a character in the font being used to display its contents, and no shorter than 10 times that font's height. If you try to reduce the CHB to as small a size as possible, you'll end up with a size something like that shown in Figure 6-1. If you count the characters and allow for the borders of the subpanes, you'll find out that the CHB is indeed 20 characters wide, and 10 lines high, at its smallest possible size.

You can also define a window's minimum size in terms of other objects that describe rectangles in Smalltalk/V. We will have more to say about this subject later in this chapter when we discuss how to describe the sizes of the subpanes within a TopPane instance.

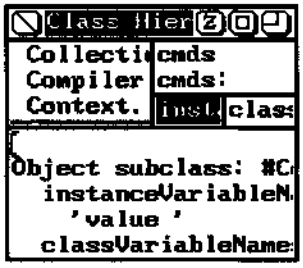

Figure 6-1. Minimum Size of the CHB Window

In the **Counter** class we defined in Chapter 5, we did not allow the window to be resized, so we could safely ignore this method. If you create windows that are not resizable, or if the minimum size of the window is immaterial to your design, you can omit calling this method when you initialize your **TopPane** instance.

The *initWindowSize* Method

You'll recall from Chapter 5 when we constructed our **Counter** class that we found Smalltalk/V's default initial window size to be too large for our purposes. To change that default size, we wrote an instance method called *initWindowSize* that returned a rectangle that was smaller than the one defined by Smalltalk/V as the default.

Most Smalltalk applications you construct will have an initial window size which is optimal. As a result, you will quite often write your own version of this method to establish that initial size appropriately.

This is one of the more interesting and obtuse of the important MPD methods we'll examine. At first glance, it would seem to be one of those methods that calls for an argument; in other words, you'd expect its name to be *initWindowSize:* with the initial window size supplied as a argument. This would seem at first glance to be more Smalltalk-like.

But it turns out that this method is implemented not in the **TopPane** class (which your instance of **TopPane** would inherit and where a direct setting of the value would therefore be logical) but rather in the class **TopDispatcher,** as we saw in Chapter 5. Remember that in Chapter 5 we opened a CHB on the class **TopDispatcher** and examined its *initWindowSize* method. We discovered that it sets up a default size that was too large for our purposes, but that was as far as we investigated the situation.

Since the comment in *initWindowSize* clearly says that it returns a value rather than setting it, the question is, "How does the window's initial size actually get set from the information returned by this method?" The answer,

obscurely enough, lies in the class **Dispatcher's** *open* method. Use the CHB to examine this method and notice the following lines:

```
size := (pane topPane model respondsTo: MnitWindowSize) if
    True: [pane topPane model initWindowSize] ifFalse: [self
    topDispatcher initWindowSize]
```

This code makes use of the *respondsTo:* method defined in the class Object and therefore understood by all objects in the environment, to determine whether you have implemented an *initWindowSize* method in your application. If you have, then Smalltalk/V uses that method to define a value. If you haven't, it uses the value returned by the *initWindowSize* method of your application's TopDispatcher. That's how the default is used whenever you fail to override it with your own definition of this method.

The *rightIcons:* and *leftIcons:* Methods

One of the strong suits of the MPD capability in Smalltalk/V is the relative ease with which you can give your windows behavior that would take significant amounts of code to program if they didn't exist in the library. Icons represent a substantial part of that capability. Smalltalk/V defines icons with the names and functions shown in Table 6-1.

Note that you can use these icons in ways other than those defined by Smalltalk/V if you want to do so, by identifying and overriding the methods that respond to the user's click on the icons.

Table 6-1. Predefined Icons In Smalltalk/V

Icon Name	Icon	Purpose
closeIt		Allows the user to close the window. Generally appears in upper left corner of title bar.
collapse		Allows the user to collapse the window to a title-bar-only size with only the collapse icon visible in the title bar.
hop		In the Debugger, used to execute the next expression in a process that is being debugged.
jump		In the Debugger, used to execute all instructions between the present one and the next breakpoint.
resize		Allows the user to resize the window using the current upper left corner as an anchor point.
skip		In the Debugger, used to execute the next expression in the selected method or up to the next breakpoint, whichever it encounters first.
zoom		Allows the user to zoom a window's text pane contents so they occupy the entire screen.

If you don't supply either a *rightIcons:* or a *leftIcons:* method in your **TopPane** definition, then Smalltalk/V will populate the title bar with a close icon on the left side of the title and icons for zoom, resize, and collapse on the right side of the title. You can eliminate all of the icons on the right side of the title bar as we did in the **Counter** class in Chapter 5 by sending our **TopPane** instance the message:

```
rightIcons: #()
```

The argument is an array of symbols which specify the names of icons to be included.

SubPane Methods You'll Need

Among the main subpanes — specifically the classes **SubPane, ListPane, and Text-Pane** — there are some 60 methods. We'll focus attention on seven of these, six of which are applicable to all types of subpanes and one of which applies only to **TextPane** objects. The seven methods you'll need to know are:

- model:
- name:
- change:
- framingBlock:
- framingRatio:
- menu:
- contents

The last, *contents,* is the one that is applicable only to instances of the class **TextPane.** All the others are relevant to all types of subpanes, including instances of **GraphPane,** whose details we delay until Chapter 8. Let's look at each of these methods in turn.

The *model:* Method

The *model:* method is required for using any **SubPane.** It is commonly used in this form:

```
aSubPane model: self
```

where *self* refers to the application object.

This is because it is rare to find a case where the model for a given subpane is different from the application model itself. Since *self* refers to the application throughout its execution, and since each pane is generally dependent on the application for its contents and behavior, *self is* almost always the right argument to the *model:* message.

The *name:* Method

The *name:* method is required and always takes as an argument a symbol that is the name of a method which the pane uses to obtain its contents from the model. This symbol is also used as the name of the **SubPane** instance. We saw this in Chapter 5 in the **Counter** class where the **ListPane** instance received the following message:

```
name: #cmds
```

and our model included a method called *cmds* which contained the following code:

```
cmds
    "Answer the list of valid commands"
    ^#(increment decrement)
```

For our Counter application, the contents of the **ListPane** are determined by sending the model the *cmds* message. In this case, a simple two-element array is returned. In other examples we'll see throughout the book, more complex methods may be invoked by this same process to give panes highly complex and dynamic behavior.

The *change:* Method

Whenever a change in a subpane's contents are global in scope—i.e., affect at least one other pane in the window — you must define a *change:* method. This method takes a single argument, a symbol that must be the name of a message that the pane will send to the model to indicate that its contents are changed. (As we'll see shortly, the model must then use its *changed:* method to deal with the change.)

Again, in the **ListPane** of our **Counter** class from Chapter 5, we find the following pair of Smalltalk/V program elements:

```
change: #cmds:

cmds: userSelection
   "Perform the command selected by the user"
   userSelection == #increment ifTrue: [self
   increment. ^self]. userSelection == ttdecrement
   ifTrue: [self decrement]
```

The *increment* and *decrement* methods, in turn, adjusted the value of the counter and then used the *changed:* message to cause the model to broadcast that change to the **TextPane** by using that pane's name, *values.*

There are several variations on the *changed:* message in Smalltalk/V. Each of these variations makes it possible to add arguments to the message to be more detailed or explicit about the *changed:* process. For example, you can use *changed:with:* to add a second argument. In that case, you will use a corresponding *update:with:* method. Similarly, there *artchanged:with:with:* and *changed:with:with:with:* messages with corresponding *update:with: with:* and *update:with:with:with:* messages

Figure 6-2 depicts the entire interactive process that transpires when a user (or, for that matter, the programitself) changes the contentsofapaneinawaythathasaglobal effect.

The *framingBlock'.* and *fromingRatio:* Methods

Perhaps the trickiest part of defining a subpane in a Smalltalk/V application is calculating its relative size and position with respect to the total window. The *framingBlock:* and *framingRatio:* methods have this responsibility. Both of these methods take an argument which evaluates to a rectangle. The difference between them may be summarized as follows:

- *framingBlock:* returns an absolute rectangle, leaving your application in control of its size and location.
- *framingRatio:* returns a rectangle proportional to the enclosing **TopPane** and whose proportions will be maintained when the window is resized. Its size is therefore relative to the size of the enclosing **TopPane** and is not within the direct control of your application, which has essentially delegated responsibility for the window size to another element of the application.

Every subpane you define must use one of these methods (but not both) to describe its size and location within the window.

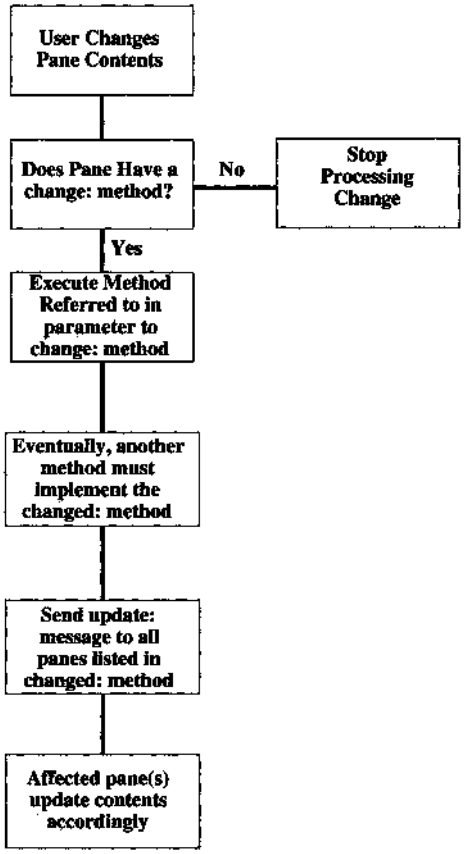

Figure 6-2. MPD Interaction Summarized

Creating Rectangles

Simply saying that each subpane occupies a rectangular area defined by either *aframingBlock:* message or *aframingRatio:* message doesn't tell you how to define a rectangle. As it turns out there are numerous ways of accomplishing this, depending on what basis you want to use for determining the size of the subpane.

A rectangle is defined by two related instances of the class **Point.** Instances of the class **Point,** on the other hand, have two instance variables, an *x* (or column) coordinate and a *y* (or row) coordinate. Smalltalk refers to points with a notation that looks like this:

```
15 @ 23
```

Conceptually and to maintain consistency of the language, Smalltalk defines this construct as sending the @ message to the first integer (in this case, 15) with the second integer argument (here, 23) as an argument.

To create a rectangle, we generally send either the *extent:* or the *corner:* message to a point, passing along another point as an argument. Smalltalk/V then creates the appropriate rectangle. Thus a statement like this:

```
15 @ 23 extent:  30 @ 45
```

creates a new rectangle with an upper-left corner 15 pixels to the right and 23 pixels down from the upper-left corner of the screen, with a width of 30 pixels and a height of 45 pixels. This same rectangle could be created by this statement:

```
15 @ 23 corner:  45 @ 68
```

Rectangles and Subpanes

It would be unwise to use these absolute-address techniques for creating subpanes in a resizable window because they wouldn't be able to adjust to the window's change in size by the user. Their positions would be fixed on the screen in absolute row-column coordinate terms and they would not understand intuitively how to move. In other words, they would not act like subpanes of the window's **TopPane** object.

That's where *framingBlock: md framingRatio:* come in. They allow you to define rectangles relative not to the entire display but to the window in which their subpanes exist. Thus when the user moves the window, the subpanes' understanding of where they are located on the screen's bitmap adjusts automatically. By this simple expedient, Smalltalk solves for you the otherwise very complex issue of keeping track of a window and all of its elements as it is dragged about on the display by the user.

Recall from Chapter 5 that when we defined the two subpanes in our **Counter** class window, we used *a framingRatio:* approach that looked like this for the **ListPane** where the commands *increment* and *decrement* were displayed:

```
framingRatio:  (0 @ 0 extent:  3/4 @ 1) )
```

and like this for the **TextPane** where the counter displays its value:

```
framingRatio:  (3/4 @ 0 corner:  1 @ 1) ) .
```

The first *of* these lines created a rectangle that started in the upper-left corner of the **TopPane** (at a point that is the "Oth" position across and the "Oth" in height) and extended 3/4 of the way across the width of the **TopPane** as well as the entire height (as indicated by the whole number 1 rather than a fraction for the second part of the argument to the *extent:* message). The second line then specifies a rectangle that starts 3/4 of the way across the **TopPane**'s width and at the top of the pane (thus the 0 for the second part of its starting point) and extended the rest of the way across and down theareaoftheTopPane(wWchisthemeaningoftheargument7 *@lmaframingRatio:* message).

The size and location **of a SubPane** within a **TopPane** are always relative, that is, they are recalculated each time the window is resized. The *framingRatio:* message provides a rectangle by which the system calculates the coordinates. You just sit back and let the system do all the work. If you use *framingBlock:* instead, you supply the code for the calculation. This code takes the form of a block (thus the name of the method) that takes a single argument. This argument is the rectangle of the window (excluding the title bar). Your code can do whatever it likes to calculate a rectangle for any **SubPane** that *usesframingBlock:* so long as your other **SubPanes** take this method of resizing into account. Your block of code must return a rectangle.

Pane Menus

If you want a pane to have a special popup menu, you define this menu with the *menu:* method as part of the subpane's definition. Note that a **TextPane** has a default popup menu that allows for menu-driven editing of the text contents of the subpane. You can add to this menu, completely replace it, or simply default to it.

The argument to the *menu:* message is a symbol that is the name of a method of the model that returns or contains the menu definition. The following code fragment from the CHB will clarify what we mean. Here is a portion of the code that defines the selector **ListPane** in the CHB (the right-most subpane across the top of the window):

```
menu: #selectorMenu
```

You would therefore expect to find a method named *selectorMenu* among the CHB's instance methods. It is indeed there and it looks like this:

```
selectorMenu
    "Private - Answer the selector pane menu"
^Menu
    labels: 'remove\newmethod\senders\implementors ' withers
    lines: Array new
    selectors: # (removeSelector newMethod senders implement or s)
```

In creating almost any Smalltalk/V menu, you'll use the same technique, namely sending the class **Menu** the *message labels: lines .'selectors:* as shown above. These three elements of the selector have the following roles:

- *labels:* takes as an argument a string that lists the names of the menu items as they will appear when the user pops up the menu. Note the combined use of backslashes to separate items from one another in the string and the *withCrs* message, which replaces each occurrence of a backslash with a line-feed character.

- The argument to *lines:* is an array of the line numbers of those menu items that should be separated from the next item by a horizontal line. In the case of the selector pane menu above, all four items appear as part of one, uninterrupted menu, so an empty array is supplied as the argument. (This is required since you cannot simply omit *lines:* from the message.) To see how this works, edit the *selectorMenu* method in the class **ClassHierarchyBrowser** so that the *lines:* argument line looks like this:

  ```
  lines: # ( 1 2 3 )
  ```

  Now select a method in the selector pane and pop up the menu. Notice the horizontal lines after each entry. Be sure to return the CHB to its original condition before you save your image.

- The final element in the selector takes an array as an argument as well. This array is a list of message selectors (or message names), each corresponding to an entry in the menu itself. If the user selects the first entry in the menu, the first selector in this array will be executed (i.e., that message will be sent to the model). You can now see how you could easily modify the behavior of a popup menu. Just change the selector associated with that menu's entry in the **Menu** definition and Smalltalk/V would send your new message to the model instead of the original whenever the user selected that menu option.

Obtaining a TextPane's Contents

The last of the subpane methods we'll look at is *contents.* As we said earlier, it applies only to instances of **TextPane.** Its use is quite straightforward. It always returns the entire contents of the **TextPane.** You can then do whatever you like with the contents. (For example, you might write them to a file, change some characteristic, etc.)

The Only Model Method You'll Need

Your application is the model, of course, and that means that it uses a great many methods. But in terms of its interaction with the MPD world, your model really only needs to deal with one method: *changed:*.

There are actually several variations on this method's theme. They are as follows:

- *changed*
- *changed:*
- *changed:with:*
- *changed:with:with:*

The only difference among these four versions of this method is the number of arguments they take. You'll choose which of them to use based on how much information you need to pass to the various panes when a change occurs that you are interested in broadcasting to the model's dependents.

If you use the *changed:with:* method, it takes as an argument the name of the method to be executed. If you use the *changed:with:with:* method, it takes as its first argument the name of the method to be executed and as its second argument an argument that this called method requires. In other words, the *changed:with:* method is used when the method being called requires no arguments and the *changed:with:with:* method is used when the method being called requires an argument.

In the **Counter** class example of Chapter 5, recall that the *increment* and *decrement methods* both sent the model the *changed:* message when the user selected one of the entries in the **ListPane.** In each case, the *changed:* message was adequate because the only argument needed was the name of the subpane (in the example, the subpane called *values)* that had to be notified of the change.

Dispatcher Methods You'll Need

We could argue that you really don't need to understand any of the messages implemented by the class **Dispatcher** and its subclasses because as a rule you'll simply use these messages without needing to know much about what they do. However, both for the sake of completeness and because they are interesting, we'll look at the two **Dispatcher** methods you will use in virtually every Smalltalk/V application you build. We'll also look at the structure of the statement in which these messages are usually sent to describe briefly how you might want to change it for some specific purpose. The two methods we'll examine are *scheduleWindow* and *open.*

The *scheduleWindow* Method

This method is sent to a dispatcher to make it a recognized part of the system as an active window. The system scheduler is notified of the window's existence and treats it accordingly. As you know, only one window can be active at a time in Smalltalk/V, and whichever window is active is the one into which the user's input will be entered for processing (provided the cursor is in the window's boundaries). When you create a new window, it is usually because you are ready for the user to do something to interact with your program, so you will want your new window to be the active one.

Warning

If you attempt to open an unscheduled window, a fatal error may result.

The *open* Method

The *open* message is sent to a window to cause it to initialize and prepare to appear on the display and to become the active window using its default size.

Standard Use of Dispatcher Methods

As you may recall from our discussion in Chapter 5, we generally end our *open* (or *openOn:)* methods with a single line that looks like this:

```
topPane dispatcher open scheduleWindow
```

This has the desired effect of causing our new window to appear on the desktop and become the active window. There are times when you need to divide this line into two operations so that after you've opened the **TopPaneDispatcher,** you can effect some other change in the window before making it the active window, at which point you lose control over some aspects of the window until the user interacts with it somehow. For example, if you wanted to position the selection (i.e., the text cursor) at a given point in the **TextPane** of a window before you made it active, you would do so in three steps:

1. Open the dispatcher with the line:

   ```
   topPane dispatcher open
   ```

2. Carry out your special subpane processing. For example, you might have a line like this:

   ```
   inputPane selectAfter: startPoint
   ```

   where you've previously named the **TextPane** as *inputPane* and defined the variable *startPoint* to contain a point within the appropriate pane.

3. Activate the window with the line:

   ```
   topPane dispatcher scheduleWindow
   ```

7
The Third Project: Creating a New Pane Type

^ Introduction

In this chapter, we extend our understanding of the MPD architecture described in detail in Chapter 6 by using our knowledge of the class **ListPane** as the basis for creating a new subclass called **MListPane,** which allows the selection and de-selection of multiple items from a list.

We begin by explaining a new approach to the design of Smalltalk applications. We will use this approach for the rest of the book, so it is important that you understand what this methodology means and how to apply it to your own designs. Once we have explained the new design approach, we' 11 work through the construction of a small application whose sole purpose is to help us solidify our understanding of the **ListPane.**

Finally, we'll create the subclass **MListPane** and demonstrate its use in a small application.

At the conclusion of the chapter, we present a postscript that describes an alternate approach to this problem that is much more direct and concise but which lacks one of the capabilities of the project we build in this chapter.

^ Designing the Project

The **ListPane** class supplied with the Smalltalk/V class library allows only a single element in a list to be selected at a time. Selecting a new element automatically de-selects any currently selected element. For some applications, we may want to allow the user to make multiple selections from the list and to invoke a single action that applies to all of them. For example, in a Disk Browser, we might want to allow the user to select several files to be printed or copied. There may also be times when we would like to know which of a list of several selected items the user selected most recently. None of this behavior is supported by the class **ListPane,** so we will add it to our application by creating a new subclass.

In Chapter 3, when we created our first example application, the Prioritizer, you'll recall that we took a strongly procedural approach to the design, consisting of the following steps:

1. Get the data from the user.
2. Sort the data elements by querying the user about each pair of alternatives.
3. Format and print the results of this processing in the Transcript.

Essentially, then, we designed the Prioritizer by building a mental model of the way the program would process the information. We focused on the process or flow of the operations. This was a perfectly acceptable way to approach the design of such a simple application but in the real world of OOP, we want to take a far more object-based approach to the design.

In the design of this application, then, we will adopt the following methodology:

1. Identify the problems to be addressed by the application. What kinds of behavior do we want it to exhibit? What kinds of objects does the application need to represent and/or manipulate?
2. Assign responsibilities to objects. What responsibilities should each object in the application have? What kinds of actions should they represent or embody? What kind of knowledge about themselves and their teammate objects must these objects have to carry out their responsibilities?

We will first identify the problems addressed by the current **ListPane** class and determine how it behaves. Then we will do the same for our **MListPane** class. As is often the case in building a Smalltalk application, the very activity of refining the responsibilities of the objects that make up the application will clarify the process by which the various elements interact.

Problems Addressed by *ListPane*

We need to understand a number of things about the way the **ListPane** class works (i.e., the problems it solves) before we can design and construct a reasonable subclass to modify this behavior. These problems include:

- how to highlight a selection
- how to interpret user input to the pane as a signal to select an element
- how to handle scrolling
- how to interact with the model so that use can be made of the selection

Responsibilities of *ListPane*

Figure 7-1 lists the responsibilities of an instance of class **ListPane** with which we are concerned.

The elements of the list that the **ListPane** displays are furnished by its model.

Problems to be Addressed by *MListPane*

In addition to the problems that **ListPane** addresses, we need to pose some additional assignments for the new class **MListPane** that we will design and build. Specifically, we need to know:

- how to interpret user input with regards to multiple selections
- how to display multiple selections (i.e., indicate that more than one item is selected and allow the user to see which element was the most recently selected)

Present all elements of list in scrollable pane

Interpret user input for selecting an item

Keep track of selected item

Inform model of changes in selection

Scroll to selected item

Highlight and unhighlight selection

Figure 7-1. ListPane Responsibilities

Responsibilities of *MListPane*

Figure 7-2 summarizes the responsibilities that the class **MListPane** will have to be able to handle. Some of these responsibilities are slight modifications of those shown in Figure 7-1 for **ListPane** while others are new.

The last of the responsibilities in Figure 7-2 requires some explanation. We need a way for our **MListPane** class to indicate selections that are currently in the list of active elements (i.e., those that the user has selected). We have chosen in our implementation to flag all currently active selections in the list by preceding them with an asterisk. The most recent selection will be highlighted, which is the behavior we inherit from **ListPane.** Figure 7-3, then, shows what a list will look like if the elements called 102,103, and 105 have been selected, with 105 the most recently activated selection.

Interpret user input to select and de-select item(s)

Keep track of multiple selected items

Inform model of changes in selections

Highlight and unhighlight most recent selection

Format and unformat past active selections

Figure 7-2. MListPane Responsibilities

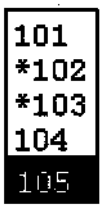

Figure 7-3. Flagging Multiple Selections

A Note About Responsibilities

The link between the pane and the model is limited to a minimum number of channels. The only things a **ListPane** can do with its model are query it for a list of items to be displayed and inform the model when the selection changes. (The **ListPane** determines when it will ask the model for a list to display, though this action is often triggered by the model broadcasting a *changed* message.) This clear delineation of responsibility is a key aspect of object-oriented design and can result in objects that are easily reusable.

Building the Test Application

As we said at the beginning of this chapter, we delay the implementation of our **MListPane** class to build a small application which confirms our understanding **of ListPane** and provides us with a test case to use as we build and test our **MListPane** class.

This test application will create a window with two subpanes, one a **ListPane** and the other an **MListPane.** (Actually, we'll start with two **ListPanes** since we haven't yet built the **MListPane** class. But to keep the discussion focused and simple, we'll still refer to the second pane as an **MListPane.** Later, when we convert it so that it actually is an instance of our newly created class, we won't have to change all our terminology.) The **MListPane** will hold a list of items and, before we're finished, will enable the selection of multiple items in the pane. The **ListPane** will be responsible for displaying a list of the items selected in the **MListPane.** If the user selects an item that is displayed in the **ListPane,** we'll simply display it in the Transcript.

We're ready to begin building the test application itself. Since it is an application that doesn't depend on any particular inherited behavior, we'll make it a subclass of the class **Object.** Open the CHB, select **Object,** and choose *new class* from the pane pop-up menu. Name the new class **ListApp.**

Defining and Initializing Instance Variables

Since we've already described the behavior we expect of this application, we can determine that this class needs two instance variables: one to hold the items to be displayed in the **MListPane** and one to keep track of the items displayed in the **ListPane.** We support this self-knowledge by defining two instance variables, *allItems* and *selectedItems.* The former keeps track of all of the items from which the user can choose in the **MListPane;** the latter

holds the list of currently selected items to be displayed in the **ListPane.** That means that the object's definition, as you will edit it in the CHB' s text editing pane, looks like this:

```
Object subclass: #ListApp
    instanceVariableNames:
        'allItems selectedItems'
    classVariableNames: " pool-
    Dictionaries: ''
```

From what we already know about **ListPane** (based on our discussion in Chapter 6 and earlier in this chapter), we can determine that the model for a **ListPane** must support two key methods, one to return an **IndexedCollection** of strings to be displayed in the **ListPane** and another that takes as its argument a new selection made in the **ListPane.** The names of these methods are included in the **ListPane,** which sends these selectors as messages to the model when it needs to invoke a corresponding action.

Since our **ListApp** must serve as the model to two different panes, it needs to support these essential methods for both of them. The four methods it must provide will be responsible for implementing the following behaviors:

- providing a list for the **ListPane** to display (we'll call this method *selectedItems)*
- receiving a change in selection for the **ListPane** *(singleSelection:)*
- providing a list for the **MListPane** to display *(allItems)*
- receiving a change in selection for the **MListPane** *(multipleSelection:*)

The selectors for these methods are stored in their respective panes and automatically sent to the model as determined by the panes.

> Note that most Smalltalk programmers use the same name for the accessing method of an instance variable as they do the instance variable itself. If this confuses you, feel free to modify the method names to something like *getAUItems* or *getSelectedItems* to sharpen the difference between symbols used as methods and instance variables.

Here are the listings for the four methods listed above. Note that, because we don't yet have an **MListPane** class, we simply have the *multipleSelection:* method do nothing. This does not, however, mean that this

method has no value or can be omitted. It provides documentation for the behavior we are planning to implement in our class.

```
selectedltems
   "Return the IndexedCol lection that represents those items
   selected in the MListPane of our application."
   ^selectedltems

singleSelection: index
   "Just to show that the model and view are communicating we
   print out some simple information in the Transcript. "
   Transcript show: 'The single item ' , index printString ,
        [1] was selected[1] ; cr.
   [A]self

alii tems
   "Returns an IndexedCol lection of items
   for use in the MListPane" ^al II tems

multipleSelection: theSelections
   "For now, this does nothing."
```

Since we need something for the *allltems* method to return, we should provide code to initialize this instance variable to some useful value. We'll define an *initialize* method that will handle this process. As you can see from the Smalltalk/V 286 documentation, **ListPane** deals with data in the form of an indexed collection of strings. Since **IndexedCollection** is an abstract class, we'll use the concrete class **OrderedCollection** to hold the strings for **ListPane.** While we're at it, we should also provide an initially empty **OrderedCollection** for the instance variable *selectedltems. Initialize* provides a place for both of these tasks:

```
initialize
   allltems := OrderedCollection new.
   100 to: 120 do: [ ritem I allltems add: item printString] .
   selectedltems := OrderedCollection new.
```

The initial value for *allltems* will be a list of numbers from 1 00 to 1 20, which is adequate for testing purposes and has the further virtue of allowing us to see at a glance which line of the list has been selected. This may come in handy during debugging.

Opening the Application Window

Now that we've got a method to initialize the instance variables for the **ListApp,** it's time to focus on the *open* method that will create a window **(TopPane),** add two subpanes (both of which are, for the moment at least, **ListPanes),** and schedule the window. We'll discuss this code in several segments, and provide a listing of the method in its entirety at the end of the discussion.

The first chunk of code, below, creates an instance **of TopPane,** provides a window label, and defines the window as having a minimum size of 20 characters in width and eight lines in height. Note that *SysFontWidth* and *SysFontHeight* are global variables in the Smalltalk/V 286 system.

```
aTopPane := TopPane new.
aTopPane label:  ' My TestApp';
    minimumSize: SysFontWidth * 20 @ (SysFontHeight * 8) .
```

Now we need to create an instance **of ListPane** (which will eventually become a new instance of **MListPane).** Here is the code that handles this assignment:

```
theMListPane := ListPane new.
theMListPane model: self;
    name: #allIterns;
    change: ttmultipleSelection:;
    framingRatio:  ((1/2 @ 0)  extent:  1/2 @ 1) ;
    returnIndex: false.
```

In addition to creating a new instance of **ListPane** called *theMListPane,* this code also sets up some important arguments for the pane.

The *model:* argument defines our instance **of ListApp** to be the pane's model.

The *name:* argument is the name of a message selector that will be sent to the model to invoke a method that returns the **OrderedCollection** of strings the pane is to display. This method also gives the pane a name by which it can identify itself, which is important because it enables the pane to respond to a *changed:* message, as we'll see later.

The *change:* argument provides a selector that will be sent to the model along with an argument to inform the model when the selection changes.

We have *seen framingRatio:* in detail before. It simply determines the size of the pane in coordinates relative to the **TopPane.**

The *returnIndex:* argument informs the pane to send the selection as a string rather than the index of the selection when we inform the model of a change in the selection. The default value for this argument *is false,* but we take this step solely to make the code more readable.

The next chunk of code is nearly identical to the last one and requires no further explanation. It simply adds the other **ListPane** (the one that stays a **ListPane** even in our final application):

```
theListPane := ListPane new.
theListPane model: self;
    name: #selectedItems;
    change: ttsingleSelection:;
    framingRatio: ((0 @ 0) extent: 1/2 @ 1).
```

The final chunk of code in the *open* method simply adds the subpanes to the **TopPane** and schedules the window:

```
aTopPane addSubpane: theListPane.
aTopPane addSubpane: theMListPane.
aTopPane dispatcher open scheduleWindow.
```

We can now test our **ListApp** by executing the following code in a Workspace or in the Transcript:

```
ListApp new initialize open.
```

When we execute this code, a window with our two panes appears. One is empty, the other contains our list of numbers (see Figure 7-4). We can select an item from the right pane, but nothing appears in the other pane. This shouldn't surprise you, since our definition *ofmultipleSelection:* doesn't do anything yet. This also means we can't yet test selecting an item in the left pane (the **ListPane)** either. But we'll get to those steps soon enough.

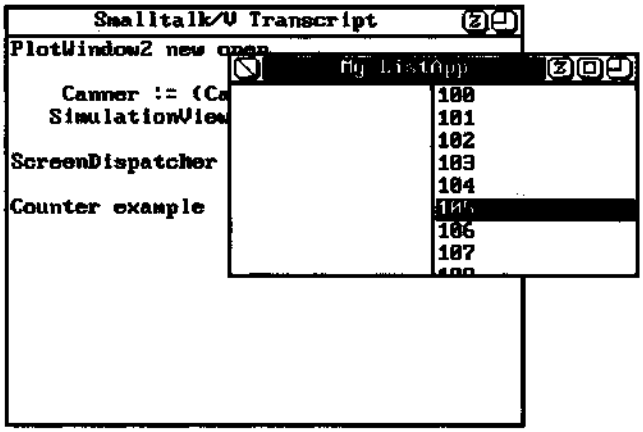

Figure 7-4. Non-Functional ListApp Window

As we did in Chapter 5, we'll create a class method called *example* that will facilitate our future testing of this application. Select the *class* button beneath the method list pane in the CHB, choose *new method* from the pane pop-up menu, and enter the following code into the text editing pane:

```
example
    "ListApp example"
    self new initialize open.
```

We can now examine the listing of the entire *open* method so that you can see it all in one place:

```
open
    "Opens our test application."
    I aTopPane theListPane theMListPane I
    aTopPane : = TopPane new.
    aTopPane label: 'My ListApp[1] ;
        minimumSize: SysFontWidth * 20 @ (SysFontHeight* 8) .
    theMListPane := ListPane new. theMListPane model: self;
        name: #alIIterns;
        change: tmultipleSelection:;
        framingRatio: ((1/2 @ 0) extent: 1/2 @ 1) ;
        returnlndex: false.
    theListPane := ListPane new.
    theListPane model: self;
        name: #selectedltems;
        change: ttsingleSelection:;
        framingRatio: ( (0 @ 0) extent: 1/2 @ 1) .
    aTopPane addSubpane: theListPane. aTopPane
    addSubpane: theMListPane. aTopPane
    dispatcher open scheduleWindow.
```

Connecting the Two Panes

Now that we have the application developed to the point where it opens a two-pane window and displays the initial list in the right pane, let's connect the two panes so that a selection in the right pane will appear in the left. (The method *singleSelection:* which we have already defined will take care of the user selecting an element of the left pane's list.) We've already established the necessary model relationships between the panes, so all we really need to do is flesh out our *multipleSelection:* method so that it handles the current capability of the **ListPane.** In other words, it needs to receive a single string

and place this string into an **OrderedCollection** so that the left pane can display it. Here's the code to handle this assignment:

```
multipleSelection: theSelections
    "Respond to the change in theMListPane selection, and
    broadcast to its dependents who respond to the aspect of
    #selectedlterns to do what they need to do." selectedItems :=
    OrderedCollection with: theSelections. self changed:
    #selectedItems.
```

The first line of code in the above listing assigns to the instance variable *selectedItems* the value of an instance of **OrderedCollection** of only one element, namely the argument passed with the *multipleSelection:* message. (The *with:* method is inherited by **OrderedCollection** from the class **Collection.)**

The first thing we need to do is to provide an instance **of OrderedCollection** for the **ListPane** to display (since, as you'll recall, we defined the *selectedItems* method for this pane to return an **OrderedCollection).** For the moment, since we are not yet dealing with multiple selections, this collection will have only one element in it. How do we create a collection with one element? We examine the hierarchy of the class library starting with **OrderedCollection** until we find a method in a superclass capable of handling this task. When we get to the class **Collection,** we find the *with:* class method that answers a collection of a single element. This is the behavior we want; since it is inherited, we simply use it.

Next, we add the *changed:* message. This method is inherited by **ListApp** from the class **Object.** It broadcasts to all dependents of the instance that a change has occurred. Its argument is a argument that lets a dependent determine whether and how it should respond to the change.

(Recall from our discussion in Chapter 6 that the very fact that we have defined the **ListApp** as the model for this pane, the pane is automatically defined as dependent on the model. We don't need to do any other explicit programming to create this dependency.)

The argument of the *changed:* message is the symbol *ftselectedItems.* This is significant since it corresponds to the name of the **ListPane.** Therefore, the argument triggers the update to the **ListPane** that we expect.

After you change *multipleSelection:* as described above and save it, try testing the **ListApp** one more time by typing into the Transcript or a Workspace the following line and executing it with *do it:*

```
ListApp example.
```

Now the application is beginning to take shape. When you select an item in the right pane, it displays in the left pane. If you select the item in the left pane, a sentence prints in the Transcript (see Figure 7-5).

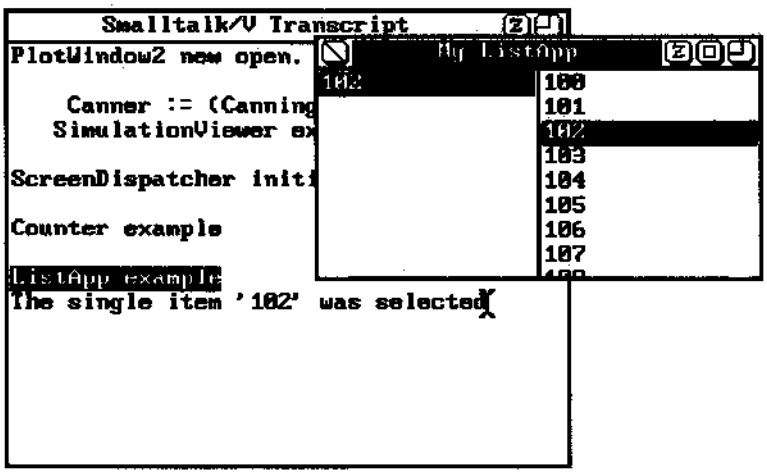

Figure 7-5. Interim Working Version of ListApp

Creating and Constructing *MListPone*

Our test case works. We have identified the behavior in ListPane that we need to understand and have implemented it in our own test application. Switching over to a new subclass of ListPane that does not override any of the currently used methods shouldn't upset anything in the design, so we'll create the MListPane class and then take another incremental step in proving to ourselves that things are working as expected.

First, create the new class MListPane as a subclass of ListPane with the following steps:

1. Open a CHB and select the class ListPane.
2. Select *add subclass* from the pop-up menu in the pane that lists the classes.
3. Name the new class MListPane.
4. Select *subclass* from the next pop-up menu.
5. In the template that appears in the text editing pane, modify the code to look like this:

```
ListPane subclass: #MListPane
instanceVariableNames:
    'selections '
classVariableNames: ''
poolDictionaries: ''
```

6. Save the change.

We have defined one new instance variable called *selections*. It will hold the potentially several selections that the user has made in the **MListPane** of our application. We also need an instance variable to keep track of the last item selected, but we can see by looking at the class definition for **ListPane** that it already has such an instance variable, called *selection*. We will simply use that inherited variable rather than upsetting the apple cart by defining our own new instance variable to handle the same task. (Another reason for doing this is that **ListPane** will take care of finding, highlighting, and other housekeeping functions as a result of this decision and its inherent behavior.)

Now let's test our application to be sure it still works with this minimal change. Find the line in the *open* method in the class **ListApp** that looks like this:

theMListPane := ListPane new.

and change it to read as follows:

```
theMListPane := MListPane new.
```

Try out the application by typing into the Transcript or Workspace the line of code that invokes the *example* method. Everything should work as before. We have proven that we can create a new instance of **MListPane** and use it in our application transparently.

Building *MListPane*

Now we start the more challenging task of building the **MListPane** class by adding to it the behavior that we need in order for it to carry out its previously defined responsibilities. We will identify the unique or new responsibilities it has, find the methods in its parent class **ListPane** which deal with those tasks, and override them in **MListPane.**

Relevant *MListPane* Responsibilities

We can identify the following responsibilities that our **MListPane** class must handle that are different from those handled by **ListPane:**

- setting the instance variable *selections* to empty any time *selection* is set to empty
- formatting and unformatting string representations in the **MListPane** as the user makes and un-makes selections
- adding elements to and removing elements from *selections*

preserving the original list of items as provided by the model (explained in detail later)

interpreting user input for selecting and de-selecting items

providing the model with the selections the user has made as a collection of indexes or strings

Let's look at each of these responsibilities in turn and see how we implement them in **MListPane.**

Clearing *selection* and *selections* as Needed

When the instance variable *selection* is set to empty, we must set our new instance variable *selections* to be an empty instance **of OrderedCollection.** Examining the methods in **ListPane** to find out where *selection* is set to *nil,* we find numerous places where this takes place. Rather than overriding all of these methods and having to deal with the two instance variables separately, we've decided to write a new method called *clearSelections* which will set both of these instance variables to appropriate empty values. Another thing we notice as we go through **ListPane** looking at the places where *selection* is set to empty is that in every case the code also sets the instance variable *currentLine* to nil. So we add this to our method as well. This leads to the following new method in our class **MListPane:**

```
clearSelections
    selection := currentLine := nil.
    selections := OrderedCollection new.
```

Now we can override the **ListPane** methods where this processing takes place and replace the line in the **ListPane** methods that reads:

```
selection := currentLine := nil.
```

with:

```
self clearSelections.
```

This affects the *close, restore,* and *update* methods, which will now look like this in their implementations in **MListPane:**

```
close
    "Close the pane."
    super close.
    self clearSelections.
```

```
restore
    self clearSelections.
    super restore.

update
    self clearSelections.
    super update.
```

Note that if you examine the *initialize* method in the **ListPane** class, you'll find that it does not set up *selection* to be *nil.* This was probably considered unnecessary in that class since an untouched variable is bound to *nil* by default. In our case, though, we must initialize *selections,* so to keep things consistent and to make the code somewhat more readable, we'll override *initialize* with a version that handles our selections:

```
initialize I
    result I
    result := super initialize,
    self clearSelections.
    ^result.
```

> **Note** that the placement of the *super* expression in relation to our *clearSelections* message is significant. Normally, we place our message send after the call to the *super* method because we want the new method to return the same information as was returned by the version of the method provided in our superclass. In this case, though, we want to empty the *selection* and *selections* instance variables before the pane is refreshed, so we place our message sends first. We do this because it is important to return the expected or same value as that returned by the inherited method. This also explains why we go to the effort of saving the value in *result.*

It is also worth noting, as a design lesson for your own work in Smalltalk, that if the original implementors of Smalltalk had handled this process of selection identification by putting the line:

```
selection := currentLine := nil.
```

in a separate method of its own, we would not have had to override it in three other places. You should see this not so much as a criticism of the originators of Smalltalk as you should take heart from the fact that even the people who invented this language didn't do everything right the first time!

Save these new methods and run the **ListApp** example again. Everything should work precisely as before. (This may sound like paranoia; after all, we

didn't make any significant behavioral change to the application in this discussion. But sometimes even the tiniest changes have a surprising effect on existing programs. We prefer frequent testing to protracted debugging sessions later!)

Formatting and Unformatting Selections

When the user selects an entry in the **MListPane,** we need to add an asterisk to the beginning of the string entry if it wasn't previously selected and remove the asterisk if the selection was previously made. (We have previously decided that the user can de-select a list entry by clicking on an entry the user had already made.)

MListPane inherits from **ListPane** an instance variable called *list* that contains the **IndexedCollection** of the list. An instance of **IndexedCollection** can reference its individual components by means of an index (which is, of course, where the class derives its name). Using simple concatenation, we can get the string that is located at the index where the selection has been made and then add an asterisk to the beginning of that string. Here is the method that handles this responsibility:

```
formatIndexedStringAsSelected: anIndex "Append an
    asterisk [*] to the string in the list instance
    variable inherited by MListPane from ListPane.[11]
    list at: anIndex put: '*' , (list at: anIndex).
```

(Note the relatively long name of this method that describes with some precision what it does. We recommend that meaning take precedence over length to make your code more readable.)

The removal of the asterisk requires some logic testing. If there is no asterisk in the first position of the string, we don't have to do anything because *ourformatIndexedStringAsSelected:* method will add the asterisk. Here is the method that removes the asterisk if necessary:

```
unformatIndexedStringAsSelected: anIndex
    "Remove the asterisk [*] from the string
    in the list instance variable inherited
    by MListPane from ListPane." I theString
    theStringLength I theString := list at:
    anIndex. (theString at: 1) == $*
        ifTrue: [theStringLength := theString size.
        theString := theString copyFrom: 2 to: theStringLength.
        list at: anIndex put: theString.].
```

Adding and Removing Elements of *selections*

When the user selects an element of the list in the **MListPane,** we must either add it to the **OrderedCollection** *selections* if it is one that was not previously on the list or we must remove it from that collection if it was previously there.

To add an element to an **OrderedCollection,** we have a number of methods from which to choose. A quick review of their operations, however, reveals that the simplest of these — the method called, simply, *add:* — is sufficient for our purposes. We also want to handle the formatting of the display list using the methods outlined in the previous section. Here is the code we'll use to add a newly selected element to the Collection being displayed in the **ListPane** in our application:

```
addToSelections: anIndex
    "Adds the index represented by anIndex to the
    collection of selections, and formats the string
    representation of the selection."
    selections add: anIndex.
    self formatIndexedStringAsSelected: anIndex.
    ^anIndex.
```

The code for removing an element from the list is similar. Examining the methods available in the class **OrderedCollection,** we find that all of them amount to over-kill. We need a simple removal of an indexed element, while all of those defined in the class either work only with specific elements (as do *removeFirst* and *removeLast,* for example) or are more complicated than we need. In the superclass **Collection,** though, we find a simple *remove:* method that appears to do what we want, so we'll use it. Here's where:

```
removeFromSelections: anIndex
    "Removes the index represented by anIndex from the
    collection of selections/ and unformats the string
    representation of the selection."
    selections remove: anIndex.
    self unformatIndexedStringAsSelected: anIndex.
    ^anIndex.
```

Preserving the Original List

With all of the methods in the previous section, we alter the contents of the list involved in the application. This destructive change to the list may not be what we want, since this list is shared by the pane and the model. The

model probably considers this rude, and other methods outside **MListPane** may depend on the model's state. Furthermore, it is bad manners to alter directly the contents of an instance variable belonging to another class.

We examine the methods of the class **ListPane** to find out where the instance variable called *list* is used and determine if the methods make a copy of the list before using it or if they return the original list. We discover that *list* is bound in several **ListPane** methods and that in no case does the system make a copy of the instance variable. (Note that if the **ListPane** class had been defined with a specialized method to handle this process, our application would be simpler to write. We would then merely have to override that method in **MListPane.** As it is, our new class will be much easier to subclass in this respect than the original **ListPane** class from which we are working.)

To provide this behavior to our **ListApp** application, we override a number of methods of **ListPane in MListPane.**

There are two main copying methods defined in the Smalltalk/V system: *deepCopy* and *shallowCopy.* The former copies a variable and all of its elements while the latter copies only the receiver and not its elements. In this case, we need to copy the strings as well as the list, so we use *deepCopy.*

In keeping with good design and the goal of reusable code, we define a method *setListFromModel* which we can then use in various other methods to take care of the copying process wherever it's needed:

```
setListFromModel
    "Gets the list from the model and stores a copy
    of it in the list instance variable. " list :=
    (model perform: name) deepCopy.
```

The perform: method is Smalltalk/V's way of sending a message to an object and returning the result of the message sent. It is a behavior exhibited by all objects in the Smalltalk/V system since it is inherited from the class **Object.** The above example sends the message *name* to the model and then sends a *deepCopy* message to the result, assigning the output to the variable *list.*

The *restore* method in the class **ListPane** also modifies the list, so we need to override it in **MListPane.** But we have already created an overriding version of the method earlier. As we examine the method in **ListPane,** though, we notice that it performs other processing that is dependent on the value of *list,* so we have to create a more complex version of the overriding method:

```
restore
    "Refresh the list from the model
    and maintain the position in the list
    without selecting it."
```

```
    topCorner == nil
    ifTrue:  [topCorner := 1@1] .
    self setListFroinModel.
    topCorner y > list size
    ifTrue:  [topCorner y:  (list size max:  1)].
    self clearSelections.
    self refreshAll
```

Notice that this new method is a hybrid of our earlier version in **MListPane** and code taken directly from ListPane.

There are two methods closely related to *restore* that we must similarly override in **MListPane.** The new code for these methods is shown here; it is self-explanatory in light of the discussion above.

```
restoreWithRefresh: aString
    "Refresh the list from the model
    and keep the line equal
    to aString showing and selected."
    self setListFromModel.
    self restoreSelected: aString

restoreSelected
    "Refresh the list from the model
    and keep the old selection."
    self setListFromModel.
    self refreshAll.
    self boldLine: selection
```

The *open* method for the **ListPane** also sets the *list* and must therefore be overridden in **MListPane** as follows:

```
open
    "Private - Open the pane."
    self setListFromModel.
```

Similarly, the *update* method in **ListPane** must be overridden because of its use of *list* to refresh the list from the model:

```
update
   "Refresh the list from the
   model and display it." self
   setListFromModel. self
   clearSelections. topCorner :=
   1@1. self refreshAll
```

We must override one remaining method. The *searchForLineToShow:* method also makes use of the *list* instance variable. Be careful in working with this method. Select the method name in the CHB and then select *senders* from the pane pop-up menu, you' 11 see that this message is sent by *restoreSelected:*. A careful examination of the code for the *restoreSelected:* method shows that it relies on the returned value of this method for its correct operation. Thus if we simply override the *searchFor-LineToShow:* method we could "break" the *restoreSelected:* method.

Further examination of the *restoreSelected:* code reveals that although this method sets the value of the list, it accesses only the length of the list, not its contents. So we can safely use *super to* handle this processing within our overridden version of the *searchForLineToShow:* method, rather than rewriting the method as we were forced to do in the methods above. Here is the new code:

```
searchForLineToShow: anObject
    "Override method inherited from List Pane."
    I result I
   result := super searchForLineToShow: anObject.
   self setListFromModel.
   ^result.
```

Interpreting User Input for Selecting and De-Selecting Elements

In ListPane, all of the interpretation of the user's input and the display of the results of the user's interaction with the application are handled in *selectAtCursor.* This method is complex as it stands; to override it with another monolithic version capable of dealing with multiple selections would require making extensive additions to a copy of the method. The result would be code that would be difficult to read and maintain. So we take advantage of the fact that we have to override this method anyway to make the code simpler to deal with. We override *selectAtCursor* with a method that unhighlights the current selection, then calls a new method to interpret the user's input, and finally redisplays the pane's contents. Here is the new version of *selectAtCursor* for **MListPane:**

```
selectAtCursor
   "Handles interpreting and displaying the user selections."
   "Set currentLine instance variable to the user's selection"
   self findCurrentLine.
```

```
self topPane textModifled "Safety feature"
    ifTrue: [^self].
currentLine isNil "Safety feature"
    ifTrue: [^self] ifFalse: [
        currentLine > list size
        ifTrue: [^self] ] . self
hideSelection. self
interpretSelection. self
refreshPane. model perform:
changeSelector with:
(returnlndex
        ifTrue: [self allSelectionsAsIndexes]
        ifFalse: [self allSelectionsAsStrings] ).
```

If you compare this method with the method of the same name in the **ListPane** superclass, you'll see that this version is much more straightforward and policy-oriented. That doesn't mean the original *selectAtCursor* method was badly designed, only that as we found it necessary to add complexity, it turned out to be easier to simplify the method and have it call other methods than to attempt to squeeze all of the new functionality into a single new version.

Notice that the overridden version of the *selectAtCursor* method calls three new methods that are not part of the **ListPane** class: *interpretSelection, allS electionsAslndexes,* and *allSelectionsAsStrings.* Here are the listings for those three methods:

```
interpretSelection
    "Performs the interpretation of the user selection.
    By having this as a separate method we can simplify the
    nesting of ifFalse: and ifTrue because we can simply test
    for a case, and return from the method when finished."
    "The first selection made in the MListPane"
    selection == nil
        ifTrue: [^selection := currentLine].
    "If the only item selected is selected again, it becomes
    unselected." ( (currentLine == selection) and: [selections
    isEmpty] )
        ifTrue: [^self clearSelections] .
    "If we have made it here, selections must not be empty so. . ."
    (currentLine == selection)
        ifTrue: [selection := selections last.
            ^self removeFromS elect ions: (selections last) ] . "The
    new selection has nothing to do with the old selection, so
    check on what it has to do with previous selections."
    (selections includes: currentLine)
```

```
        "Remove the selection from selections."
            ifTrue: [self removeFromSe lections: currentLine] "A completely new
        selection. Select iti"
            ifFalse: [self addToSelections: selection. . ^selection :=
        currentLine].

allSelect ionsAsIndexes
    "Return an OrderedCollection of indexes
    representing selections in the pane."
    I result I
    result := selections shallowCopy.
    result add: selection.
    result

allSelectionsAsStrings
    "Return an OrderedCollection of strings
            representing the unformatted selections in the pane."
    I result cleanList I
    cleanList := self getListFromModel.
    result :== OrderedCol lection new.
    selections notEmpty
        ifTrue: [selections do: [ :anIndex |
    result add: (cleanList at: anindex) ] ] . selection notNil
        ifTrue: [result add: (list at: selection) ] . ^result        — •
```

The *allSelectionsAsStrings method, in turn, calls an as-yet-undefined method called getListFromModel, which sounds like it ought to return a copy of the list from the model. In fact, that's just what we'll define that method to do:*

```
getListFromModel
    "Gets the list from the model and stores a copy of it in the list instance
    variable." A (model perform: name) deepCopy.
```

Now that we've written a method that returns a copy of the list, we can rewrite *setListFromModel to use this new method, thus further reducing the number of links between our pane and the model. This is always a desirable design goal.*

```
setListFromModel
    "Gets the list from the model and stores a copy of it in the list instance
    variable." list := self getListFromModel.
```

^ Providing the Model With User's Selections

The last responsibility we've assigned our new class is that of providing the model with the selections made by the user as collections of indexes or of strings. This is determined in the *selectAtCursor* method when it references the *retumlndex* variable, as we explained when we began building our test application, **ListApp.**

The *allSelectionsAslndexes* and *allSelectionsAsStrings* methods described earlier carry out the bulk of the work here. They examine the pane contents and return an appropriate value. We have also modified *selectAtCursor* to deal with these new methods.

To demonstrate the use of this new approach to returning the r"^ information to the model, change the *multipleSelection:* method in the class ListApp so that it gets an OrderedCollection rather than a String as it does in **ListPane.** Here is the modified code for this method:

```
multipleSelection: theSelectionStrings
    "Respond to the change in theMListPane selection, and broadcast that its
    dependents who respond to the aspect of #selectedItems to do what they need
    to do." selectedItems := theSelectionStrings. self changed: ftselectedItems.
```

With all these changes made, try out the modified **ListApp** and experiment with the results. Figure 7-6 shows an example of the experimental use of this application.

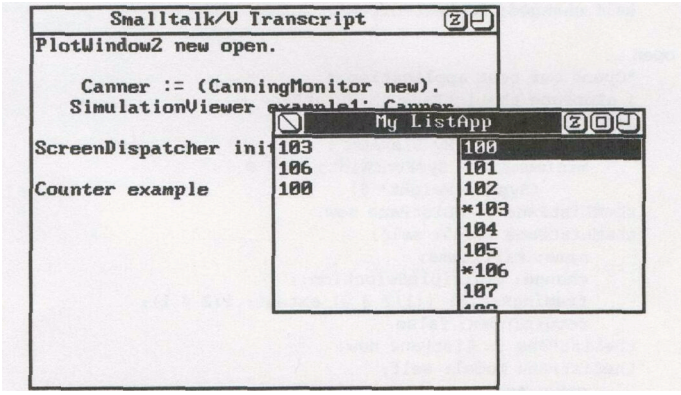

Figure 7-6. Completed ListApp Application

^ The Complete Listing

Here is the complete listing of the code in this chapter, in file-in format.

```
Object: subclass: #ListApp instanceVariableNames :
'allItems selectedItems ' classVariableNames: ' '
poolDictionaries: '' !

!ListApp class methods !

example
        "ListApp example" self- new Initial i ze open!


!ListApp methods !

allltems
   "Returns an IndexedCollection of items for
   use in the MListPane"
   ^allltems!

initialize
   allltems := OrderedCollection new.
   100 to: 120 do: [ :item I allltems add; item printString].
   selectedltems := OrderedCollection new.
   "self!

multipleSelection: theSelectionStrings
   , "Respond to the change in theMListPane selection, and
   broadcast that its dependents who respond to the aspect of
   ftselectedltems to do what they need to do." selectedltems
   := theSelectionStrings. self changed: #selectedltems!

open
    "Opens our test application."
    I aTopPane theListPane theMListPane I
    aTopPane := TopPane new.
   aTopPane label: 'My ListApp';
       minimumSize: SysFontWidth * 20 @
       (SysFontHeight* 8) .
    theMListPane := MListPane new.
    theMListPane model: self/name:
             ttallltems;
       change: ftmultipleSelection:;
       framingRatio: ((1/2 @ 0) extent: 1/2 @ 1) ;
       returnlndex: false.
    theListPane := ListPane new.
   theListPane model: self;
       name: #selectedltems;
       change: ftsingleSelection:;
       framingRatio: ((0 @ 0) extent: 1/2 @ 1).
    aTopPane addSubpane: theListPane.
    aTopPane addSubpane: theMListPane.
    aTopPane dispatcher open scheduleWindow!

selectedltems
   "Return the IndexedCollection that represents those items
   selected in the MListPane of our application."
```

```
   ^selectedltems!

singleSelection: index
   "To show that the model and view are communicating
   we print out some simple information in the
   Transcript. " Transcript show: 'The single item ' ,
   index printString
   ' was selected' ; cr.
   ^elf! !
```

ListPane subclass: ftMListPane
instanceVariableNames:
'selections ' classVariableNames: "
poolDictionaries: ' ' 1

IMListPane class methods ! !

```
!MListPane methods !

addToSelections: anindex
   "Adds the index represented by anindex to the
   collection of selections, and formats the
   string representation of the selection."
   selections add: anindex.
  self format indexedStringAsSelected: an-
  Index. ' ^anindexI

allSelectionsAsIndexes
   "Return an OrderedCollection of indexes
   representing selections in the pane."
   I result I
   result := selections shallowCopy.
   result add: selection.
  result!

allSelectionsAsStrings
   "Return an OrderedCollection of strings
  representing the unformatted selections in the pane.
   I result cleanList I
  cleanList := self getListFromModel.
  result := OrderedCollection new.
  selections notEmpty
      if True: [selections do: [ :anindex I
  result add: (cleanList at: anindex)]].
  selection notNil
       ifTrue: ::-si;", .dd: n^st at:
   selection)]. ^result!

clearSelections
  selection := currentLine := nil.
  selections := OrderedCollection
  new!

close
   "Close the pane."
  super close.
  self clearSelections!

  formatIndexedStringAsSelected:
  anindex "Append an asterisk [*] to
  the string in the list instance
  variable inherited by MListPane from
  ListPane." list at: anindex
      put: '*' , (list at: anindex)!
```

```
getListFromModel
    "Gets the list from the model and stores a copy of
    it in the list instance variable." ^ (model
    perform: name) deepCopy!

initialize
    I result I
    result := super initialize.
    self clearSelections.
    •^result!

interpretSelection
    "Performs the interpretation of the user selection.
    By having this as a separate method we can simplify the
    nesting of ifFalse: and ifTrue because we can simply test
    for a case, and return from the method when finished."
    "The first selection made in the MListPane"
    selection == nil
         ifTrue: [^selection := currentLine].
    "If the only item selected is selected again, it becomes
    unselected." ((currentLine == selection) and: [selections
    isEmpty])
         ifTrue: [^self clearSelections].
    "If we have made it here, selections must not be empty so...
    (currentLine == selection)
         ifTrue: [selection := selections last.
              '"'self removeFromSelections: (selections last)] .
    "The new selection has nothing to do with the old selection,
    so check on what it has to do with previous selections."
    (selections includes: currentLine) "Remove the selection
    from selections."
         ifTrue: [self removeFromSelections: currentLine]
    "A completely new selection. Select it!"
         ifFalse: [self addToSelections: selection,
              ^selection := currentLine]!

open
    "Private - Open the pane."
    self setListFromModel!

refreshPane
    "Refresh the display without changing the value of list,
    as is done with the restore* methods."
    self refreshAll.
    self boldLine: selection!

removeFromSelections: anIndex
    "Removes the index represented by anIndex from the
    collection of selections, and unformats the string
    representation of the selection."
    selections remove: anIndex.
    self unformatIndexedStringAsSelected: anIndex.
    ^anIndex!

restore
    "Refresh the list from the model and
    maintain the position in the list
    without selecting it." topCorner == nil
         ifTrue: [topCorner := 1@1].
    self setListFromModel. topCorner
    y > list size
         ifTrue: [topCorner y: (list size max: 1)].
    self clearSelections.
```

```
    self refreshAll!

restoreSelected
    "Refresh the list from the model
    and keep the old selection."
    self setListFromModel.
    self refreshAll.
    self boldLine: selection!

restoreWithRefresh: aString
    "Refresh the list from the model
    and keep the line equal
    to aString showing and selected. "
    self setListFromModel.
    self restoreSelected: aString!

searchForLineToshow: anObject
    "Override method inherited from ListPane."
    I result I
    result := super searchForLineToshow: anObject.
    self setListFromModel.
    ^result !

selectAtCursor
    "Handles interpreting and displaying the user selections."
    "Set currentLine instance variable to the user's selection"
    self findCur rent Line. self topPane textModified "Safety
    feature"
         ifTrue: ["self] .
    currentLine isNil "Safety feature"
         ifTrue: ["self] ifFalse: [
              currentLine > list size
              ifTrue: ["self] ] . self
    hideSelection. self
    interpretselect ion. self
    refreshPane. model perform:
    changeSelector with:
    (returnlndex
              ifTrue: [self allSelectionsAsIndexes]
              ifFalse: [self allSelectionsAsStrings] ) !

selections
    "Returns the current selections in the pane."
    I allSelections I
    allSelections := selections shallowCopy.
    allSelections add: selection.
    "allSelections!

setListFromModel
    "Gets the list from the model and stores a copy
    of it in the list instance variable." list :=
    self getListFromModel!

unformatlndexedStringAsSelected: anlndex
    "Remove the asterisk [*] from the string
    in the list instance variable inherited
    by MListPane from ListPane." I theString
    theStringLength I theString := list at:
    anlndex. (theString at: 1) == $*
         ifTrue: [theStringLength := theString size.
    theString := theString copyFrom: 2 to: theStringLength.
    list at: anlndex put: theString.] !
```

```
update
    "Refresh the list from the
    model and display it." self
    setListFromModel. self
    clearSelections. topCorner
    := 1@1. self refreshAll! !
```

An Alternative Approach

When we had finished the work on this chapter, Mike Anderson of Digitalk, who handled most of the technical review on the manuscript, suggested that we consider an alternate approach which, while eliminating one bit of functionality from **MListPane,** was a much simpler approach to the single problem of multiple selections. His solution, which does not allow us to deselect items or to know by examination which item was selected last, is nonetheless highly instructive.

He modified the **ListPane** method *boldLine:* so that it took a collection as an argument rather than an integer that indexes the location of a single entry in the pane. Along with other modifications shown in the listing that follows, this approach results in fewer modifications than our approach. You will benefit from studying it and its behavior in comparison with the MListPane project we built together in this chapter.

Here's the listing for Anderson's **MultipleSelectListPane** project.

```
Project : MLP
Date    : Nov 1, 1990
Time    : 00:16:07

Classes :
    MLPTest MultiSelectListPane

Methods :



Object subclass: ttMLPTest
  instanceVariableNames:
    'multiSelectListPane '
  classVariableNames: ''
  poolDictionaries: ''!

ListPane subclass: #MultiSelectListPane
  instanceVariableNames: ' '
  classVariableNames: ''
  poolDictionaries: ''!


!MLPTest class methods ! !
```

```
IMLPTest methods !

ClassCorranent
    A'this class is creates a simple window
      to test MultiSelectListPanes ' !

list
    ^#(one two three) !



        labels: 'show selected1
        lines: #()
        selectors: # (showSelected) !

open
     (TopPane new
        addSubpane: (multiSelectListPane := MultiSelectListPane new
        model: self; name: #list; change : # select : ; menu: ttmenu) )
        dispatcher open scheduleWindow!

select: aString!

showSelected I
     stream I
    stream := String new asStream.
    multiSelectListPane selection do: [ -.line I
    stream
    nextPutAll: line printString; nextPutAll:
    ' , ' ] . Menu message: stream contents!
    !


!MultiSelectListPane class methods ! !


!MultiSelectListPane methods !

boldLine: aCollection
    "Private - Reverse the set of lines indexed by
    aCollectoin in the receiver pane." aCollection do: [
    :line I paneScanner reverse:
             (self lineToRect: line)]!

graySelection
    "Private - Change the visual cue of the selection to
    reflect a deactivated window." selection do: [ :line
    I paneScanner gray:
             (self lineToRect: line)]!

hideSelection
         "Private - Turn off the reverse of the
      selected items."
    selection do: [ :line I
    paneScanner recover:
             (self lineToRect: line)]!

initialize
```

```
    super initialize.
    selection := OrderedCollection new!

selectAtCursor
    "Private - Set currentLine to the line at the
    cursor position." self findCurrentLine. self
    topPane textModified
    ifTrue: [^self].
    currentLine isNil
    ifTrue: [^self]
    ifFalse: [
            currentLine > list size
                ifTrue: ["self]].
    super boldLine: currentLine.
    (selection includes: currentLine)
    ifTrue: [selection remove: currentLine]
    ifFalse: [selection add: currentLine]. model
        perform: changeSelector
        with:
             (returnlndex
                 ifTrue: [currentLine]
                  ifFalse: [list at: currentLine])!

"construct application"
((Smalltalk at: #Application ifAbsent: [])
    isKindOf: Class) ifTrue: [
        ((Smalltalk at: #Application) forr'MLP[1])
            addClass: MLPTest;
            addClass: MultiSelectListPane;
            comments: nil;
            initCode: nil;
            finalizeCode: nil;
            startUpCode: nil]!
```

8
The Graphic World

Introduction

Smalltalk is a completely graphic environment. Its graphic nature has set it apart from other PC languages. It should come as no surprise, then, that you can control its graphic nature and create interesting and useful graphic patterns. In fact, if you've ever run the demos that come with Smalltalk/V 286, you've seen some intriguing graphic effects available on a desktop computer in a program which is not primarily designed to create graphics.

In this chapter, we're going to examine Smalltalk/V's graphic world in some depth. We'll take a look at the basic concepts of graphic programming as Smalltalk/V implements them. Then we'll examine the use of forms and drawing primitives in the language, pausing in the process to look at the basic mathematics involved in calculating and drawing graphic objects. We'll also take a brief look at some of the ways Smalltalk/V 286 implements and supports animation.

We'll accomplish our usual triage on the graphic classes, helping you focus on the classes and methods that are important for you to understand as you add graphics to your own Smalltalk applications.

All of this will prepare us for Chapter 9, where we will undertake the design and construction of a highly graphic application.

Basic Graphic Concepts

Everything displayed on the Smalltalk screen — windows, lines, cursors, even text characters — is composed of a series of dots (more technically referred to as pixels, an acronym for picture elements) arranged in a pattern. A line appears as a continuous vector of such dots whose color is black (on a monochrome display) or any color contrasting with the background (on a color display).

On a monochrome display, a bitmap defines a rectangular area of bits where each bit has a value of 1 if the pixel represented by that bit is black and 0 if it is white. Because of this method of representing graphic patterns, Smalltalk is said to use bitmapped graphics.

An individual pixel within a bitmap is addressed as a point with an x coordinate and a y coordinate. The x coordinate defines the point's column (horizontal position) and the y coordinate defines its row (vertical position). The coordinates may be stated relative to the entire area of the display or relative to a specific pane, depending on the circumstances and the need.

A bitmap in Smalltalk/V is always contained in an instance of the class **Form** or in one of its subclasses — **BiColorForm or Color Form.**

Rectangular collections of bits in a bitmap can be described as instances of the class Rectangle, where the origin (upper-left corner) and either the bottom-right corner or the size (height and width) of the rectangle are known, with the latter two being expressed relative to the upper-left corner.

It is important to understand that instances of the class **Point or Rectangle** are not displayable objects but simply represent positions and areas. The object that actually holds the graphical image of points and rectangles is the **Form.**

The basic classes of graphic data representation and display, then, are **Point, Rectangle, and Form.** Let's take a brief look at each of these classes. (This subject is covered in some detail in the *Smalltalk^286 Manual,* so we do not attempt a complete treatment here. If you want more information on any of these subjects, refer to the manual.)

The Class *Point*

An instance of the class **Point** has two instance variables of interest: *x* and y. As we have said, these represent, respectively, the horizontal and vertical position of the point.

To create a new point, you send the @ message to an integer. (We saw some of this in Chapter 6 when we studied how to define the size and relative location of a subpane in a window.) You can create a point and then interrogate it for its coordinates in a process depicted in Figure 8-1. Notice that the point we create is never displayed; we have not given the point an instance of class **Form** within which it can display itself.

You can perform a number of manipulations on instances of the class **Point.** For example, you can add, subtract, or multiply two points or a point and an integer value. If you use a point as the argument to any of these messages, then the x coordinate of the point used as a argument is applied to the x coordinate of the receiver. The y coordinates are treated similarly. For example, evaluate the following expressions separately in a Workspace or Transcript window with *show it:*

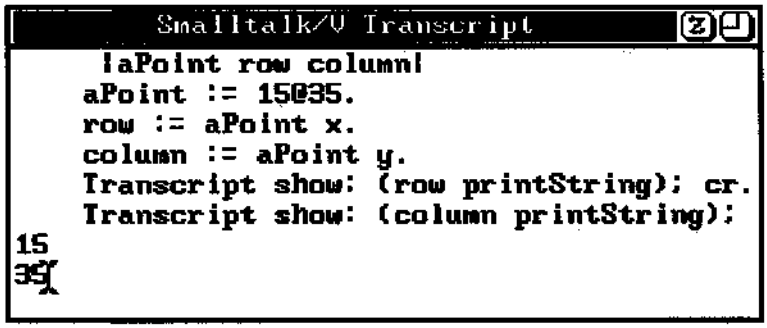

Figure 8-1. Creating and Referencing a Point

```
(15 @ 30 +  (-1 @ -2))  x
((15 @ 30)  +  (-1 @ -2)) y
```

The first expression should return the result 14 and the second should return the result 28. Now try:

```
15 @ 30 + 13
```

You should get the point 28@43 as an answer. Since you supplied only a single value rather than a point, Smalltalk/V adds the value to both the x and y coordinates.

The Class *Rectangle*

One of the most useful things you can do to a **Point** is to create a **Rectangle** with it. Any rectangle is defined by two points: its upper-left corner and its lower-right corner. Thus, we can make a point into a rectangle by supplying it with another point. There are two methods to handle this:

- *corner:,* which returns a rectangle whose upper-left corner is the receiver of the message and whose lower-right corner is the coordi nates of the argument, itself a point.
- *extent:,* which returns a rectangle whose upper-left corner is the receiver of the message, and whose lower-right corner is calculated as the sum of the receiver and the argument.

For example, the following two rectangles, taken from the *Smalltalk/V 286 Manual,* are identical:

```
1 @ 1 corner: 100 @ 100
I @ 1 extent: 99 @ 99
```

You can change (or define) the size and/or location of a **Rectangle** with either of two methods: *origin:corner:* or *origin:extent:*. As you would expect, these two methods operate quite similarly to *corner:* and *extent:* in the **Point** class.

You can also shrink or expand a **Rectangle** in relative terms with the *insetBy:* method. (Actually this method creates anew rectangle with the new size.) This method takes a rectangle, a point, or a number as an argument and reduces the coordinates of both coordinates in both directions accordingly. Evaluate the following expression in a Workspace to see the *insetBy:* method in action:

```
(1 @ 1 extent: (99 @ 99)) insetBy: 10
```

The result should be the rectangle defined as:

```
II @ 11 corner: 90 @ 90
```

Notice that the *insetBy:* method added 10 to the x and y coordinates of the upper-left corner and subtracted 10 from the coordinates of the lower-right corner. The effect, then, is a **Rectangle** that is shrunk from the original, or inset on both corners by a specific amount.

There are numerous other operations you can perform on a **Rectangle.** These are largely self-explanatory and described in the Smalltalk/V 286 documentation, so we won't go into them in detail here.

The Class *Form*

So far, the graphics classes we've looked at have been somewhat abstract (not in the sense that they are formally abstract Smalltalk/V classes, but in the sense that we can't *see* them doing anything). Now we're ready to take a look at the class **Form,** which you can think of as the canvas on which all Smalltalk drawing takes place.

A Form is a subclass of **DisplayMedium,** which in turn is a subclass of **DisplayObject.** Figure 8-2 depicts the hierarchical relationship of these three classes and describes their functions briefly. As you can see, both of the superclasses of **Form** are abstract classes. This may seem strange, but this design makes it possible for Smalltalk/V applications to be quite portable across machines with different types of displays.

DisplayObject	Provides common protocol for transferring rectangular blocks on a DisplayMedium.
DisplayMedium	Provides methods for coloring and drawing borders around rectangular areas
Form	Contains the bitmap that is displayed on the screen

Figure 8-2. Hierarchy of Class Form

To create a new **Form,** you can use the *width:height:* class method. For example, evaluate the following line of code in a Workspace or the Transcript. The result should look something like Figure 8-3.

```
(Form width: 100 height: 50) displayAt: 150 @ 150
```

Get rid of the rectangle by choosing *redraw screen* from the system pop-up menu. Now change the color of the rectangle to black by sending the new **Form** the *reverse* message:

```
(Form width: 100 height: 50) reverse displayAt: 150 @ 150
```

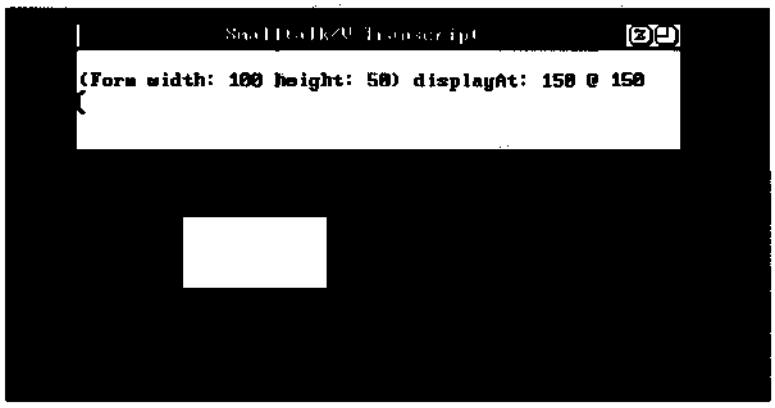

Figure 8-3. Sample Form Displayed

The result of executing this will look something like Figure 8-4. You should restore your display after you've proven that this works.

You can display only part of a Form by using the *displayAt:clippingBox:* message. This message takes a **Point and a Rectangle** as arguments. To see the impact of this, enter the following expression in the Workspace and then *show it:*

```
| aRect |
Form1 := (Form width: 100 height: 50) reverse.
aRect := (150 @ 150) extent: 100 @ 50.
Form1 displayAt: aRect origin clippingBox: (aRect insetBy: 20 @ 10)
```

A smaller black rectangle will be displayed at approximately the same point on the screen as before. Its center will be at the same point, but its four corners will have been inset from their original positions.

You should note, too, that both *displayAt:* and *displayAt:clippingBox:* are inherited from **DisplayObject** which, as you'll recall, has responsibility for moving rectangles of bits around within the display environment.

Drawing in Smalltalk/V

Now that you are familiar with the three basic graphic classes in Smalltalk/V, let's look at the question of how drawing takes place in the system. To do this, we need to look at three new classes:

- **BitBlt,** which is the fundamental class of all drawing operations.
- Pen, a subclass of **BitBlt** which creates visible lines in a **Form.**
- **GraphPane,** a subclass of **Pane,** that is associated with a **Form.**

Let's take a brief look at each of these classes and their important instance variables and methods.

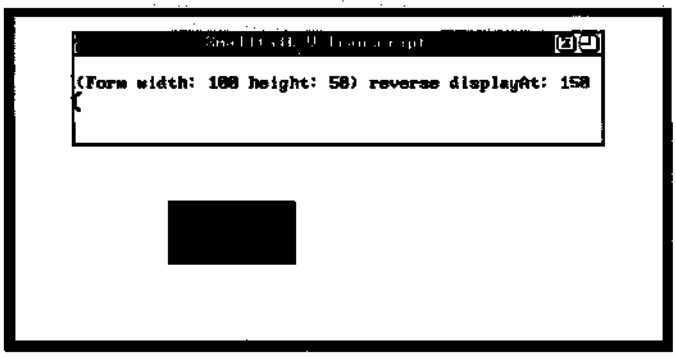

Figure 8-4. Sample Form Displayed in Black

The Class *BitBIt*

This class takes its strange-sounding name from the concept of a bit block transfer and is usually pronounced "bit-blit" by experienced Smalltalk programmers. As its full name implies, this class has as its primary responsibility the transferring of blocks of bits from one area to another. This simple-sounding process turns out to be quite complex because the process of moving these blocks of bits around involves three important ideas: rules, masks and clipping rectangles. As we will see, the message called *copy Bits* actually carries out the transfer of the bits.

The Mask Form

The movement of a block of bits from one place to another involves the careful combination of the characteristics of three different forms:

- the *source form,* where the bits to be moved are located before the process begins
- the *destination form,* where the bits are to be moved
- the *mask form,* which contains information **BitBIt** uses to determine the appearance of the bit block as it is transferred to (i.e. drawn in) its destination form

The message that initializes all of the necessary instance variables in an instance of the class **BitBIt** is *destForm: ordestForm:sourceForm:*. (There is one other initialization message that sets up six other instance variables but we won't be using it here and you will probably find little practical use for it unless you are creating quite complex graphic applications.) Once a **BitBIt** has been initialized, you define its mask form with the *mask:* message.

Some of the interesting power and complexity of **BitBIt** manipulation lie in the mask form. This form defines a "halftone," or pattern that produces a gray or multi-color effect as the bits are transferred. On a monochrome display, the mask has one of five pre-defined values:

- black
- darkGray
- gray
- lightGray
- white

As **a BitBIt** transfer takes place, the bits in the source form are first logically ANDed with the bits in the mask form. A mask form is restricted to have a height and width of 16 pixels. When the source form with which the mask

form is being ANDed is larger than 16 by 16 pixels, as is usually the case, then **BitBlt** repeatedly tiles the mask form across the width and down the height of the source form, performing a logical AND operation with each 16x16 area of the source form.

Type the following code into the Transcript and then *do it* to see what we are talking about when we discuss **BitBlt** and its key instance variables:

```
| aForm |
aForm := (Form width: 100 height: 100).
(BitBlt destForm: Display sourceForm: aForm)
   mask: Form lightGray;
   copyBits.
```

Figure 8-5 shows what the screen should look like after you execute this code. Note the light-gray rectangle in the upper-left corner of the display.

Clipping Rectangles

To avoid the possibility of drawing outside a window or other boundary, the class **BitBlt** allows you to define a clipping rectangle that defines the area in the *destForm:* within which your drawing operations such as *copyBits* will be confined.

You have undoubtedly noticed that when you type or draw inside a pane in Smalltalk, you do not have to do anything special to ensure that your changes don't overrun the edge of the pane in which you are writing or drawing. Smalltalk handles this clipping for you automatically.

When you create and manage your own panes and forms, you may need to be concerned about preventing action in one part of a window from overlapping into and affecting the appearance of other portions of the

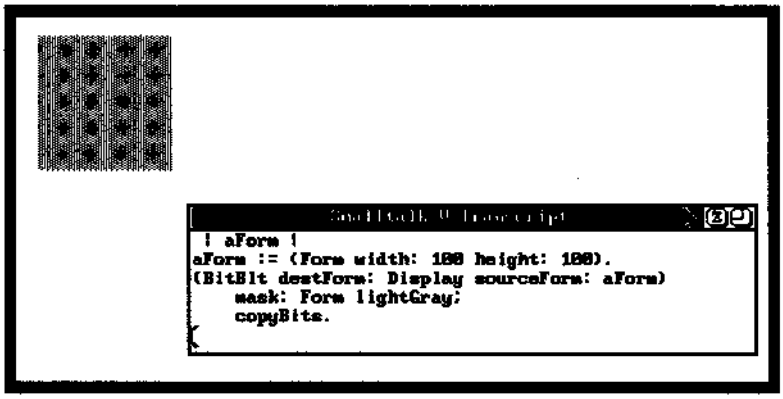

Figure 8-5. A BitBlt With a Light Gray Mask Form

window.

This process is handled by the *clipRect:* message, which takes a rectangle as an argument.

If you haven't deleted the code we used near the end of the previous section to show how a mask form works, you can use that same code and add a new line so that it now looks like this:

```
I aForm I
aForm := Form width: 100 height: 100.
(BitBlt destForm: Display sourceForm: aForm)
   mask: Form lightGray;
   clipRect: (20 @ 20 extent: 30 @ 30);
   copyBits.
```

Figure 8-6 shows you how the screen should look when you select the above code and *do it.* Notice that the light-gray mask form now appears in only a part of the area (specifically within a rectangle defined with an upper-left corner of 20,20 and a lower right corner of 50,50). This is because the drawing process ordered by the *copyBits* message has been confined to the clipping rectangle described in the line immediately before the *copyBits* message is sent.

The Class Pen

Whenever you draw a line on a Smalltalk display, you do so with an instance of the class Pen. This class is a sub-class of the class **BitBlt** which we have just finished examining. When you create an instance of the class Pen and then tell it to draw a line, it actually uses its **BitBlt** to copy from its source form to its destination form at each position along the line you describe.

Since the drawing portion of a Pen — i.e., its point, or nib — is a source form, you can change its size or its pattern by merely altering the form or its associated mask form, as discussed above.

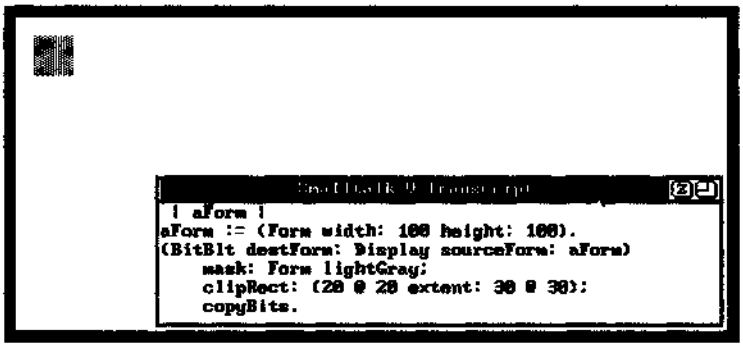

Figure 8-6. Using a Clipping Rectangle with a Form

Any instance of Class Pen keeps track, among other things, of its current location, direction, and downState (i.e., whether it is drawing or moving without leaving a trace behind).

We'll take a look at nine of the most useful methods in the class Pen in some detail. We can divide these methods as follows:

- two methods that deal with the status of the Pen (i.e., whether it is drawing or not)
- two methods that deal with the **Pen's** point, or nib
- five methods that deal with movement and location of the Pen

Methods Relating to Pen Status

There are two methods that relate to the status of an instance of the class **Pen:** *up* and *down.*

Sending an instance of Pen the *up* message effectively stops drawing as it moves around in response to other messages that relocate its position in the form. Conversely, sending it the *down* message resumes drawing.

A new instance of the class **Pen** starts with its point *down.*

Methods Relating to a Pen's Nib

When you create a new instance of class Pen, you generally give the drawing instrument a point, or nib, to use as a default. As you use the Pen, you may wish to change its nib from time to time. As we indicated earlier, a Pen nib is actually a source form used as the **Pen** draws lines during movement.

The *defaultNib:* method sets the size and shape of the nib of a Pen. It takes an integer or a point as an argument. With no *defaultNib:,* Smalltalk/V defaults to a nib of one pixel by one pixel in size. This is equivalent to supplying 1 or the point 1 @ 1 as an argument to the *defaultNib:* message. If you use an integer argument, Smalltalk/V creates a nib that is that integer number of pixels wide and high. In other words, the command:

```
Pen defaultNib: 5.
```

is identical in effect to the command:

```
Pen defaultNib: 5@5.
```

Whenever the **Pen** draws, it uses its current *defaultNib:* setting. To change the size of the nib, you must send another *defaultNib:* message with a new integer or point value.

If you want to alter the source form the Pen uses to draw, then you send it a *changeNib:* message. This message takes the name of a **Form** as an argument and changes the nib of the pen to draw with that form as its new source form. Note that the size and weight of the nib remain as set in the most recent *defaultNib:* message, or at 1 @ 1 if no *defaultNib:* message has ever been sent to the current instance of Pen.

Other characteristics of the lines drawn by the Pen are set with methods inherited from Class **BitBlt.** These include *mask:* and *destForm:*. These methods were discussed earlier when we described the class **BitBlt.**

Methods Relating to Pen Movement and Location

There are numerous methods in the class Pen that deal with the present location, orientation, and movement of the Pen. We'll focus our attention on the six most frequently used:

- *direction :*
- *turn:*
- *place:*
- *home:*
- *go:*
- *goto:*

Together, these five methods permit you to do most of the drawing you'll want to do in all but the most sophisticated graphic packages in Smalltalk/V.

The *direction:* Method

The *direction:* method points the Pen in a specific direction. Its argument is an integer with a value between 0 and 359. Figure 8-7 shows the main compass points and their values in degrees as understood by a Smalltalk Pen. Actually, the integer argument to the *direction:* message can be a value outside the range of 0 to 359. But if you supply such a value, Smalltalk/V adjusts it to fit within the arguments. For example, if you send an instance of the class Pen a *direction: 630* message, Smalltalk/V appears to subtract 360 from 630 and point the Pen in the same direction as if you'd supplied a direction value of 270. In fact, what happens is that Smalltalk/V simply rotates the Pen through the 360 degrees of the circle as many times as the integer dictates. Similarly, a value of -15 for the *direction:* of a Pen results in an actual direction of 345 degrees.

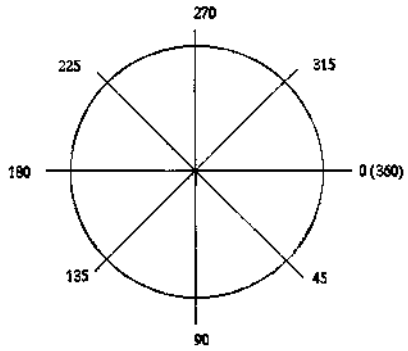

Figure 8-7. Pointing a Pen in a *direction:*

The *turn:* Method

To change the direction of the Pen relative to its present heading, use the *turn:* method. Its argument is the number of degrees by which you wish to change the *direction* of the Pen. It may be positive (for clockwise rotation) or negative (for counter-clockwise rotation).

For example, if the current *direction* of the Pen is 135 degrees, send it the message:

```
turn: 90
```

would result in a new *direction* of 225 degrees. No drawing takes place during *a turn:*.

The *place:* Method

The *place:* method takes a point as an argument and positions the Pen at that point. No drawing takes place while the Pen is relocated from its present position to the new point indicated by the *place:* message.

The *home* Method

You can place the Pen in the precise center of its destRect by sending it a *home* message. Sending this message does not alter any other aspect of the Pen, including, for instance, its direction.

The *go:* Method

To move the Pen relative to its present position, send it a *go:* message with an integer as an argument. The integer tells the Pen how many pixels to move in its current direction. For example, if the Pen is located at point 50 @ 50 in the **Form** in which it is drawing, and is pointed in direction 270 (i.e., straight toward the top of the screen) and you send it a message like this:

```
Pen go: 200.
```

then it will move 200 pixels straight up the screen. (Actually, its vertical movement is adjusted depending oil the aspect ratio of the display, but that is a nicety you can ignore for the moment.)

The *goto:* Method

You can achieve absolute movement of the Pen to a defined point on the screen while drawing a line (if the Pen is *down,* of course) with the *goto:* message. This message takes a point as an argument and moves the Pen from its current position directly to that point in the current **Form.**

It is important to note that this kind of movement by nature overrides the Pen's current direction setting if it is necessary to do so. The direction of the Pen remains unchanged after movement with the *goto:* method, however.

Class *GraphPane*

Smalltalk/V defines a specific type of pane called a **GraphPane.** Its purpose, as its name implies, is to accommodate drawing activities in much the same way an instance of the class **TextPane** is designed to accommodate the display and edit text.

You know that drawing in Smalltalk requires an instance of class **Form.** An instance of class **GraphPane** is associated with a Form which contains a backup of the bitmapped image shown in the pane. This allows the pane to refresh its contents if, for example, they should be obscured by an overlapping window.

This basic behavior of a **GraphPane** is the major reason for using a pane rather than an arbitrarily defined **Form** for drawing.

The basic behavior and methods of panes we discussed in Chapter 6 and worked with in Chapter 7 are valid for instances of class **GraphPane** as well. Creating a graphic subpane requires the same basic syntax as that for creating a text pane.

9
The Fourth Project: A Graphing Application

Introduction

This chapter presents a graphic application which demonstrates the use of many of the classes and methods we examined in Chapter 8. In the process, you will get a chance to see a highly incremental approach to Smalltalk programming in which we build the application in three distinct stages.

As with our previous projects, we will begin this chapter by describing the application's purpose and operation. Then we will discuss its design in terms of the Smalltalk/V class library and other considerations. Finally, we'll construct the application in small segments, explaining each as we proceed.

The program in this chapter was designed and developed by the authors in conjunction with Morton Goldberg, who wrote the code and extended the original concept into a more intriguing example than it would have become without his efforts.

Designing the Application

This application will be a scaled-down version of a business graphing package. It will lack some of the functionality and polish that a commercially published program for converting business data into graphs would have, but within its limited scope and purpose, it will give the user a good deal of flexibility and control over the appearance of the graphs it produces. The window of this application—which we will call Plot Window to keep things simple — consists of two panes. One pane displays the data being graphed. The other pane displays the graph itself in the form of a collection of horizontal or vertical bars. The user can choose the orientation of the bars (horizontal or vertical), their spacing, proportionality, and pattern (which could be extended on a color display to specify the color of the bars). Obviously, a more robust application would allow the user more types of graphs.

Data to be graphed can be entered directly by the user into the data pane in the application window, or it can be read from a file. As you might expect, then, data entered into the data pane can also be saved into a disk file.

Menus will provide the user with the capability to erase the plot, redraw it, return to the Plot application's default settings, and edit (cut, copy, or paste) data.

Figure 9-1 shows the basic structure of the application, with the window showing its two subpanes and all of the application's menus.

TheSubpanes

As you can see from Figure 9-1, this application's window has two subpanes. The one on the left is used to enter the data to be graphed. It is an instance of class TextPane. The one on the right is where we graph the data. It is an instance of class **GraphPane.**

Note, too, that the window has two icons in the upper-right corner: a resize icon and a collapse icon. As you'll see when we build the application, we define a minimum size for the window that means that the user can make the plot window larger, but not smaller. We do this for two reasons. First, it is useful for you to know how to maintain a minimum window size in case your application would be difficult to run or unusable if its window were reduced below a certain size. Second, we do it because our application is an example of such a situation. If the user resized the window to be much smaller than we initialize it to be, the bars of the graph could become so small as to be meaningless. The differences in bar size between even relatively disparate quantities could become invisible.

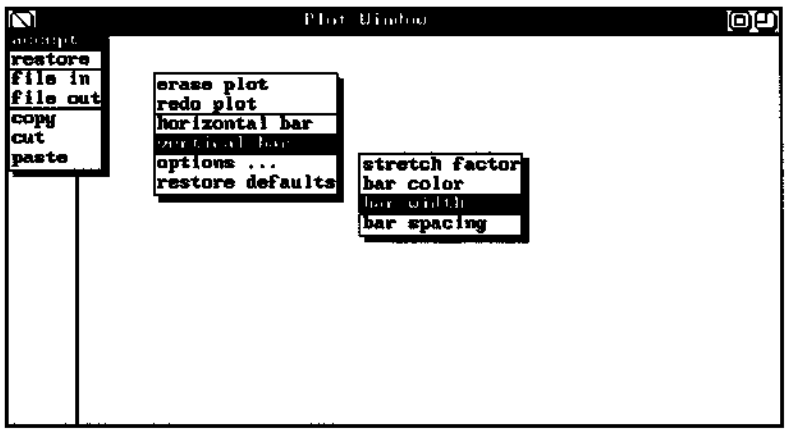

Figure 9-1. Basic Structure of Plot Application Window

The Class *PlotWindow*

We create an entirely new class for this application, as you will usually do. We call this class **PlotWindow.** Because this is a largely self-contained application, we make it a descendant of the root class **Object** rather than subclassing. Start by creating this class, editing the class definition template to add the following instance variables: plotPane, plotSelector, factor, barWidth, barFill, barSpacing, and barPen.

Building the Application: Stage One

As we indicated at the beginning of this chapter, we're going to develop the Plot application using an incremental style. You may often find this approach useful, particularly in situations where you wish to demonstrate to someone (a customer or user, for example) how an application is going to look before you've spent a lot of time and energy developing its more complex behavioral methods.

When you take this approach, you need to decide how many stages to use and how much functionality you'll incorporate at each stage. These decisions tend to be somewhat arbitrary and application-specific, so we can't give you a lot of guidance. The issues involved are basic computer science concerns that are addressed in many other books. We can, however, say that the first stage of an incremental development in Smalltalk should almost always be one that produces a basic shell of what the application will ultimately look like. It should have little or no functionality. Its purpose is to demonstrate to a prospective user what the application looks like, what its basic menu choices are, and how it will behave in the broadest sense of the term.

We have chosen to develop this application in three stages:

1. Non-functional shell
2. Basic graphing capability, with most user options not yet incorporated
3. Inclusion of all user options

The *open* Method

As will often be the case in incremental Smalltalk program development, we'll begin by creating the method that sets up the application window and its subpanes. By convention, we'll refer to this as the *open* method. The code

for this method appears below. Other than the inclusion of an instance of class **GraphPane,** there is nothing in this method you haven't seen in earlier chapters, so we won't spend time describing it here.

```
open
    "Open the Receiver. Define the pane sizes and behavior,
     and schedule the window. "
  I topPane I
  topPane := TopPane new.
  topPane
    model: self;
  minimumSize: self initWindowSize extent;
  label: 'Plot Window'; rightIcons: #(resize
  collapse). topPane addSubpane: (TextPane
  new model: self; menu: ftdataMenu; name:
  #dataPane; change: #string:from:;
  framingBlock:
  [ :box I box origin extent: 50 @ box height ] ) . topPane
  addSubpane: (plotPane := GraphPane new model: self;
  menu: #plotMenu; name: #plotPane:; framingBlock:
          [ :box I (box origin + (50 @ 0) ) corner: box corner ] )
  topPane dispatcher open scheduleWindow
```

The *initWindowSize* Method

As we saw in Chapters 6 and 7, most of the time we want to include an *initWindowSize* method in any pane definitions we create. We are tying the minimum size of the application window to its initial size, so we must create a method to define that initial size. The code follows:

```
initWindowSize
    "Answer the initial size for a receiver Window."
  'M Display boundingBox insetBy: 30 @ 25)
```

Since we plan on creating a window that occupies most of the screen's display area, we inset it initially only a little from the top left corner of the screen.

Methods for the Text Pane

The text pane where the data will be entered requires three basic methods, as you can see from its definition in the *open* method above: *dataMenu, dataPane,* and *string:from:*. We have seen the functional equivalents of these methods before. The code for the three methods appears below:

```
dataMenu
  "Answer the menu for the data pane. " ^Menu labels:
  'accept\restore\copy\cut\paste' withers
     lines: #(2)
    selectors: #(accept cancel copySelection cutSelection
       pasteSelection)

dataPane
    "Answer the initial data pane contents, ah empty string."


string: aString from: aDispatcher
    "The data pane's contents have been changed, so change
     the plot. Answer true."
  ^true
```

The menu for this pane will, in this preliminary version at least, allow the user to choose from the options: accept, restore, copy, cut, and paste. We will see that later we will decide (based, in a real-world situation, on the customer's feedback in all likelihood) that we need to add other functions to this menu. But for the moment, we'll leave the prototype in this state.

Notice that the *change:* method—named, in this case, *string:from:*—does nothing in this prototype version of the application. Later, we'll fill it in with code that will result in it changing the plot when the data in the **TextPane** changes.

For now, we just need to have the method present to avoid an error that would otherwise arise if the user selected "Save" from the menu.

Methods for the Graphing Pane

From the *open* method, you can see we need to define two methods for the **GraphPane** to avoid compilation errors and complete work on our semi-functional prototype version of the Plot application: *plotMenu* (which defines the menu for the pane) *and plotPane:* (which plays the usual role of a *name:* method in a subpane definition). Here is the code for these two methods:

```
plotMenu
     "Answer the menu for the plot pane."
   ^Menu
      labels: 'erase\horizontal bar\vertical bar' withers
      lines: #()
      selectors: #(clearPlot setHorizontal setVertical)

plotPane: aRect
  "Make a blank form the size of the screen. Display the form in
aRect on the screen. Answer the form." | blankForm |(blankForm
:= Form width: Display width height: Display height)
      displayAt: aRect origin clippingBox: aRect.
   ^blankForm
```

Notice that the menu allows the user to erase the current graph or to choose either a horizontal bar or a vertical bar.

The *plotPane:* method creates a new blank **Form** the size of the screen (which is why you see the **Display** being used as a sizing reference). In practice, we'll send the *plotPane:* message with a argument that is a rectangle. You'll see how this works when we flesh out the application later.

Demonstrating the First Version

In keeping with the principle we presented earlier, we'll define an *example* class method for our new **PlotWindow** class. Like most such methods, it is quite simple:

```
example
    "Open a PlotWindow for demonstration purposes"
    PlotWindow new open
```

Now we can demonstrate how this window will look and describe its behavior by executing the following line of code:

```
PlotWindow example
```

You can now display the two pane menus, show how the minimum size of the window is constrained, collapse the window, and describe what will happen when the menu choices are selected in the finished application. (Of course, we haven't defined the methods that will be called when the menu choices are actually made, so if you or the user inadvertently selects a menu choice at this point, an error walkback will appear. Some prototype designers would take our design a step farther and define dummy methods for all of these menu choices. You can make this determination for yourself. It will probably vary with the application and the nature of the user for whom the demonstration is being presented.)

Incidentally, the menu options in the **TextPane** will work as expected, since the methods they call are standard **TextEditor** methods. We'll learn more about text editing classes and methods in Chapters 10 and 11. Another thing you can demonstrate to the user is that an attempt to close the window after the contents of the **TextPane** have been changed will result in a confirmation window being presented by Smalltalk.

For a prototype, this little application is reasonably demonstrable. You can show the user the window and its subpanes, demonstrate its movement, resizing and collapsing, examine the menu choices you've decided to include, and show how data entry and editing will work in the **TextPane.** You can also show the user a built-in safeguard against accidentally deleting data. And yet you've had to write relatively little code to get this far.

We won't reproduce the entire listing of this prototype version of the application here as we have done in earlier chapters. Instead, we will list the next version's code in its entirety since that is the first one that will actually do something when you run it.

Building the Application: Stage Two

In the second stage of our application construction, we will add the ability to draw the bar graphs. We will also define and initialize some factors that the user will later be able to adjust affecting the style, appearance, spacing, and other characteristics of the graphing bars themselves. Finally, to make the application complete, we will fill in the *string:from:* method that we need to cause the graphing pane to update its contents in response to the user's modifying the data in the **TextPane.**

Plotting the Plots' Arguments

Before we can write the methods that will draw the bars, we need to decide what factors make up the size, appearance, and relative positioning of the bars in a graph. Then we can determine which factors we want to control within the application and which ones we wish to allow the user to modify. That will give us the information we need to create an initialization routine to set up default values for all of these variables and to write the bar-generating methods *horizontalBar:* and *verticalBar:* to make use of these variables.

Three of these values are relatively self-evident from an examination of any bar graph. These are the width of each bar (which you can also think of as the thickness of the lines), the spacing between bars, and the pattern or color to use to fill in the bars. This last value, as you can probably guess from our discussion in Chapter 8 about drawing in Smalltalk/V, will determine the mask to be used for the **Pen** we create to draw the bars.

A study of data graphing techniques reveals one other characteristic of a bar in a graph which we may wish to accommodate. We need some way to control the scaling of the data in the **GraphPane** so that we can convey the maximum amount of information to the user in the space available. Depending on the ranges of values the user enters, the scaling needs to be adjusted. For example, if the user enters a series of values all within the range of 10-50, and then later wants to plot a series of values ranging from 100 to 500, we don't want the second plot to be 10 times as wide as the first (or, conversely, for the first plot to be one-tenth as wide as the second). This could result in either miniscule bars that are very hard to differentiate from one another at low ranges so we can accommodate larger ranges or bars that are so wide that getting to the ends so we can assess their relative sizes would be cumbersome at best and impossible at worst.

To accommodate this need, we have chosen to use a stretch factor, which is an integer by which we will multiply each of the data values to be plotted.

A quick examination of these four constraints on a bar graph reveals that we probably want the user to be able to control them in all situations. In the final version of the application, then, we'll devise a method to allow the user to modify any or all of these characteristics from a popup menu without having to modify the code of the application. For this version, however, we're going to establish reasonable default values for all of these variables and define a method to initialize them. This is, of course, preferable to "hard-coding" the values directly in the bar-graphing methods, since it will make it easier later to modify them either in the code (as we develop and experiment with it) or from a menu (when we make that capability available

to the user). We will, however, add the methods necessary to permit the user to determine which type of graph to use.

The *initialize* Method

Now that we know what variables we need to accommodate, we can easily write the *initialize* method that will set up the arguments for a bar graph unless the user modifies any or all of them later.

We need to initialize one additional factor, namely the type of bar to be drawn. We had already determined in the prototype to let the user choose a horizontal or vertical bar. We're going to decide, quite arbitrarily of course, to start out assuming the user wants to use horizontal bars.

Here, then, is the code for the *initialize* method:

```
initialize
    "Initialize instance variables with default values."
    plotSelector := #horizontalBar : .
    factor := 4. "stretch the data values by this much"
    barWidth := 2 * SysFont height. barSpacing :=
    SysFont height. barFill := Form gray. barPen := Pen
    new mask: barFill
```

Notice that we use the system font' s height as the yardsticks by which to set up default values for width and spacing of the bars. This is preferable to hard-coding some value since it allows the display to be flexible depending on the system and the font it uses. Also notice the last line of this method where we create an instance of class Pen with which to draw the bars and give it the mask value stored in the instance variable *barFill.*

The mere inclusion of this method and the definition of these instance variables necessitate two other changes to our earlier code.

First, we have to declare these instance variables in the class definition. We've done this before, so we won't take the time to reproduce the code here. As we go along in this development, you'll notice that we add some other instance variables to the list as well. We'll point these out as they arise; be sure to add them to the class definition.

Second, we must call this *initialize* method from somewhere in the *open* method so that the initialization actually takes place. This simply requires us to add a line anywhere in the *open* method that says:

```
self initialize.
```

Drawing the Bars

With the constraint variables defined, we are ready to write the methods that actually draw the bars in the **GraphPane.** They are quite similar to one another, so we'll take a close look at the *horizontalBar:* method and then reproduce the *verticalBar:* method with little commentary. The drawing process can be thought of as taking place in five steps:

1. Create a form in which the drawing will take place.
2. Define a pen with which to draw.
3. Loop through each data element, calculating and drawing each bar.
4. Set up a backup **Form** where the **GraphPane's** contents can be stored in the event the pane needs refreshing at some point.
5. Use *copy Bits* to cause the actual display of the graph as described in Chapter 8.

Here is the code for the *horizontalBar:* method. We'll analyze its various components with the above five-step process in mind.

```
horizontalBar: theData
    "Draw a horizontal bar plot of theData, an ordered
    collection of integers." | form x y |
    form := Form width: Display width height: Display height.
    x := 0. y := 0. barPen
    defaultNib: 1 @ (barWidth * Aspect) rounded/-
    direction: 0; destForm: form. theData do: [ :value |
        barPen place: x @ y. barPen
        go: factor * value.
        y := y + ((barSpacing + barWidth) * Aspect) rounded ]
    plotPane form: form. plotPane paneScanner copyBits
```

Creating the Form

The first executable line of code in the *horizontalBar:* method defines a new **Form** the size of the display screen. We saw this approach in Chapter 8.

Defining the Pen

Having created an instance of class Pen called *barPen* in the *initialize* method, we now send this new drawing instrument a series of messages.

The first message sets the pen's *defaultNib:* to be one pixel high. The width of the nib is calculated by multiplying the width of the bars defined in the *initialize* method by the aspect ratio of the display, a value that is stored in a special Smalltalk/V global variable called *Aspect.* (Note that this variable's name begins with a capital letter to indicate that it is either a global or a class variable.)

Next, we set the pen's direction to be 0, which, as we saw in Chapter 8, means that it will begin drawing from left to right (or, directionally, to the east) when it is moved.

Finally, we tell the Pen to use the form created earlier as its destination **Form.**

Calculating and Drawing the Bars

We use a *do:* block to loop through the data in the instance variable *theData* that is passed as an argument with the *horizontalBar:* message. (We will see shortly how this variable is determined from the values entered by the user in the **TextPane.** This processing is handled by the *string:from:* method that is defined as this pane's *change:* method.)

For each entry in this variable, which is an instance of class **OrderedCollection,** we place the pen at its next starting point, then use the *go:* method to cause the pen to draw the bar. Finally, we relocate the vertical position of the pen by incrementing the value of its *y* instance variable. Then we repeat the process.

Notice that we use the stretch factor, which we stored in the instance variable *factor,* to scale each element in the data collection as we draw it. Note, too, that we determine where to place the pen for the next horizontal bar by adding the spacing and width of the bars — factors that are set up in the *initialize* routine — and then multiplying them by the aspect ratio of the display.

Setting Up a Backup Form

The next to last line in the *horizontalBar:* method uses the **GraphPane** instance method *form:* to define the form declared at the first step in this drawing process as the backup form for the **GraphPane's** contents.

Displaying the Bars with *copyBits*

The last line in the *horizontalBar:* method uses the *copyBits* message which, as you know, is the method that causes the drawing to appear in the

destination form, in this case the **GraphPane.** Another function of this method is to hide the cursor if it is in the **GraphPane** when the drawing takes place.

The *verticalBar:* Method

Now that we understand the *horizontalBar:* method, we can look at its counterpart for drawing vertical bars. Here is the code for the *verticalBar:* method:

```
verticalBar: theData
    "Draw a vertical bar plot of theData, an ordered collection
     of integers." | form x y |
     form := Form width: Display width height: Display height. x
     := 0.
     y := form height.
     barPen
     defaultNib: barWidth @ 1;
     direction: 270; destForm:
     form. theData do: [ : value |
     barPen place: x @ y. barPen go: factor *
     value * Aspect, x := x + barSpacing +
     barWidth ] . plotPane form: form;
        scrollUp: plotPane frame height - form height.
     plotPane paneScanner copyBits.
```

There are, as you can see, relatively few differences between this method and the *horizontalBar:* method we have just examined. We should, however, point out some of the differences.

We initialize the *y* value to the height of the form (which in this case is the same as the height of the display) rather than to 0 because we want to start drawing vertical bars not from the left of the screen but from the bottom of the **GraphPane.**

Notice that we set up the *defaultNib:* for the pen slightly differently because we no longer need to adjust its dimension by the value *of Aspect* since we are giving it an absolute vertical size of 1. Similarly, to get the pen to draw from the bottom of the screen to the top, we need to set its direction to 270 rather than 0.

In the *go:* message, we now must take into account the aspect ratio of the display to keep the bars proportional to the size and shape of the screen. Similarly, we need not take into account this aspect ratio for the *x* position

of the pen drawing the bars. So the last two lines of the *do:* block are slightly different from the *horizontalBar:* method's last two lines.

Finally, there is a new line, second from the end of the method. It uses the **GraphPane** method *scrollUp:* to scroll the pane's contents so that you can see the bottom of the pane no matter how large the values being plotted become. You can then use the standard scrolling techniques to move the view around so you can see the tops of any bars that might be too high for the pane's clipping rectangle.

Defining Graph Selection Methods

Remember from the discussion of the *initialize* method that we start the Plot application with horizontal bars as the default plot type. We store this value in an instance variable called *plotSelector*. The **GraphPane** menu invokes two specific methods — *setHorizontal* and *setVertical* — which should determine the plot type to use. These methods, then, must set the value of *plotSelector* in accordance with the user's wishes. Here is the code for these methods:

```
setHorizontal
   "Change the plot type to horizontal bar"
   plotSelector := tthorizontalBar:

setVertical
   "Change the plot type to vertical bar"
   plotSelector := #verticalBar:
```

The clearPlot Method

Another **GraphPane** menu method we need to create is the one that erases the graph. This turns out to be surprisingly easy. No need to determine where graphic objects appear and draw over them with some other pen. All we have to do is define a new form for the pane that is the same size as the current form's frame. Here's the code:

```
clearPlot
    "Clear the plot pane by giving it a new blank form."
  plotPane form: (self plotPane: plotPane frame)
```

The *string:from:* Method

The last method we have to write for this stage of our application's development is the *string:from:* method that handles the conversion of the data into a usable form and creates a new plot with the resulting data. Recall that in our prototype, we created a dummy version of this method. It is now time to round it out. Here is the code:

```
string: aString from: aDispatcher
    "The data pane's contents have been changed. Convert the
    new data into an ordered collection of integers and make a
    new plot. Answer true." I input data token I input :=
    aString asStream. data := OrderedCollection new. [ input
    atEnd ] whileFalse: [
            token := input nextWord.
                token notNil
                    ifTrue: [ data addLast: token asInteger ] ].
     self perform: plotSelector with: data. ^true
```

This method takes the string that is its first argument and converts it to a stream. The string is the data entered in the **TextPane.** We need it in a stream form so that we can parse it and break it into its individual entries. There is no way to parse a string directly, so we convert it to a stream first.

Next, we define an instance of the class **OrderedCollection** called *data* which we will use to hold the parsed values.

We work our way through this collection an item at a time, adding each new "word" (in this case, a number) to the end of the collection until we have converted all of the elements of the stream to an **OrderedCollection.**

Finally, we use the *perform:with:* message to instruct the model to carry out the method whose name is presently stored in the instance variable *plotSelector* with the **OrderedCollection** as its data argument. This, as we have seen in examining the *horizontalBar:* and *verticalBar:* methods, actually generates the graph.

Demonstrating the Second Version

This version of the program is fully functional, so we can demonstrate all of its capabilities.

Whether you use the *example* approach or just call the *open* method directly, create a new **Plot Window.** It should look exactly as it did during the demonstration of the prototype earlier.

Now enter a small number of values — we recommend you keep them in the range of 1 to 125 for the moment because the stretch factor of 4 that we've applied makes values larger than about 125 disappear off the right edge of the pane.

After you have entered a few values, select the "accept" option from the data pane's popup menu. A horizontal graph of the data elements using a gray pen should appear in the **GraphPane,** similar to that shown in Figure 9-2.

Now you can choose the "erase" option from the **GraphPane** popup menu and the graphing area will be clear. Choose "vertical bar" from this same menu.

Put the cursor into the data pane and make some kind of a change to the contents. You can leave the values the same by simply pressing the Return key at the end of the list or you can edit one or more values. You can even clear all the values out of the pane and enter new ones. As soon as you select the "accept" option from the pane popup menu, the new vertical graph should appear, similar to the one in Figure 9-3.

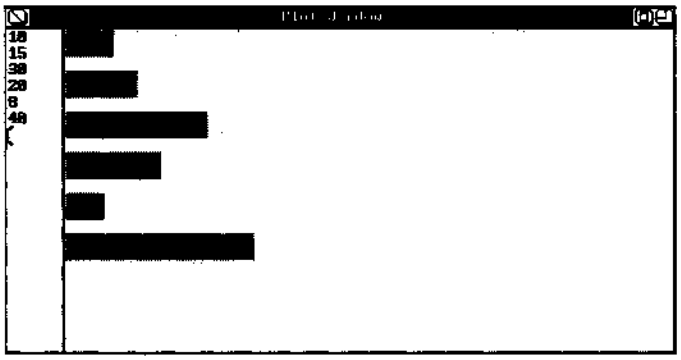

Figure 9-2. Sample Horizontal Bar Graph

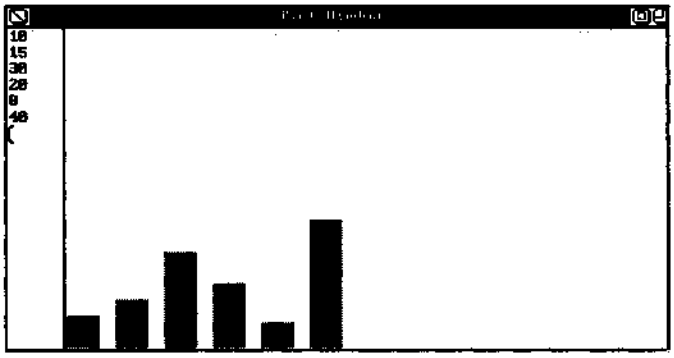

Figure 9-3. Sample Vertical Bar Graph

You should also collapse, resize, move, and close the window to be sure all of those built-in functions work as expected.

The Complete Listing of Second Version

Because this is a fully functional application now, and because you may wish to stop with this version rather than building the somewhat more robust and polished version we'll look at next, we reproduce here the complete listing of the Plot application in its present incarnation. As usual, it is in fileIn format.

```
Object subclass: #PlotWindow
  instanceVariableNames:
     'plotPane plotSelector factor barWidth barFill barSpacing
     barPen '
  classVariableNames: ''
  poolDictionaries: '' !

! PlotWindow methods !

clearPlot
    "Clear the plot pane by giving it a new blank form."
    plotPane form: (self plotPane: plotPane frame)!

dataMenu
    "Answer the menu for the data pane."
     ^Menu labels: 'accept\restore\copy\cut\paste'
    withers
        lines: #(2).
       selectors: #(accept cancel copySelection cutSelection
        pasteSelection)!

dataPane
    "Answer the initial data pane contents/ an empty string."


horizontalBar: theData
     "Draw a horizontal bar plot of theData, an ordered
     collection of integers."
      I form x y I
     form := Form width: Display width height: Display
     height. x := 0. y := 0.
```

```
    barPen
    defaultNib: 1 @ (barWidth * Aspect) rounded;
    direction: 0; destForm: form. theData do: [ :
    datum I
          barPen place: x @ y.
          barPen go: factor * datum.
        y := y + ((barSpacing + barWidth) * Aspect) rounded ]
   plotPane form: form. plotPane paneScanner copyBits!

initialize
   "Initialize instance variables with default values."
    plotSelector := #horizontalBar:.
    factor := 4. "stretch the data values by this much"
    barWidth := 2 •* SysFont height. barSpacing :=
    SysFont height. barFill : = Form gray. barPen := Pen
    new mask: barFill!

initWindowSize
    "Answer the initial size for a receiver Window."
    ^(Display boundingBox insetBy: 30 @ 25) !

open
    "Open the Receiver. Define the pane sizes and behavior,
    and schedule the window." I topPane I self initialize.
    topPane := TopPane new. topPane
      model: self;
    minimumSize: self initWindowSize extent;
    label: 'Plot Window'; rightIcons: #(resize
    collapse). topPane addSubpane: (TextPane new
    model: self; menu: #dataMenu; name:
    ttdataPane; change: #string:from: ;
    framingBlock: [ :box I box origin extent: 50
    @ box height ]). topPane addSubpane:
    (plotPane :=
```

```
            GraphPane new model:
              self; menu:
              #plotMenu; name:
              #plotPane:;
              framingBlock: [:box I
               (box origin + (50 @ 0)) corner: box corner ]) .
        topPane dispatcher open scheduleWindow!

    plotMenu
        "Answer the menu for the plot pane." ^Menu labels:
         'erase\horizontal bar\vertical bar[1] withers
            lines: #(1) selectors: #(clearPlot setHorizontal
           setVertical)!

    plotPane: aRect
        "Make a blank form the size of the screen. Display the form
        in aRect
         on the screen. Answer the form."
         I blankForm I
         (blankForm := Form width: Display width height: Display
         height)
           displayAt: aRect origin clippingBox: aRect.
         "blankForm!

    setHorizontal
       "Change the plot type to horizontal bar."
       plotSelector :=#horizontalBar: !

    setVertical
       "Change the plot type to vertical bar. "
       plotSelector := #verticalBar: !

    string: aString from: aDispatcher
        "The data pane's contents have been changed. Convert the
        new data into an ordered collection of integers and make
        a new plot. Answer true." I input data token I input :=
        aString asStream. data := Orderedcollection new. [ input
        atEnd ] whileFalse: [
               token := input nextWord.
               token notNil
                ifTrue: [ data addLast: token aslnteger ] ] .
        self perform: plotSelector with: data. [A]true!

    verticalBar: theData
        "Draw a vertical bar plot of theData, an ordered collection
         of integers."
```

```
 I form x y I
form := Form width: Display width height: Display
height. x := 0.
y := form height.
barPen
defaultNib: barWidth @ 1;
direction: 270; destForm:
form. theData do: [:value I
barPen place: x @ y. barPen go: factor *
value * Aspect. x := x + bar Spacing +
barWidth ] . plotPane
   form: form;
  scrollUp: plotPane frame height - form height.
plotPane paneScanner copyBitsl !
```

Building the Application: Stage Three

In this final stage of building the Plot application, we are going to add the following functionality:

- user selection and alteration of the fill pattern, spacing, width, and stretch factor for the bars, using a method of menus and sub-menus
- file-based data retrieval and storage
- more intuitive and usable re-drawing of the plot on user demand

These changes have nothing to do with graphics and you can skip building this implementation of the application, if you choose. But if you are interested in any of the above design issues, you may find the relatively small amount of reading and the few code changes outlined in the rest of this chapter interesting.

User Selection of Graph Arguments

As you'll recall, we have three graph arguments — fill pattern, spacing between bars, and width of individual bars — the user should be allowed to change. The process for these changes varies only in small details from one choice to another, so we'll examine the first closely, then look briefly at the remaining two.

There are many ways we could permit the user to indicate a desire to change one of these characteristics of a plot. We could create one large menu, for example, that listed all of the options in ruled-off groups. We could design another pane for the window that would contain buttons with which the user could activate these modifications.

But in the interest of extending our experience with Smalltalk to the maximum possible extent, we're going to use a design here that we have not seen before. We're going to show you how to build a series of menus that appear to cascade from one another. Essentially, we'll add a new item to the plotMenu that will generate a menu from which the user will pick the characteristic to change. Depending on the user's choice on this menu, another menu may appear from which the user can choose the desired value for the option.

Changing the *plotMenu*

We're going to add three choices to the *plotMenu* method of our earlier version of this application. The first of these changes will allow the user to redraw the plot without having to change the data in the **TextPane** by forcing an update of the **GraphPane** unless the data pane is empty. The second will result in the menu of options the user can change. The last will return all of the characteristics of a plot to their original default values, a frequently desired alternative that will encourage users to experiment with other settings because they'll know they can always return to the original settings with a simple menu selection. Here is the code for the new *plotMenu* method:

```
plotMenu
    "Answer the menu for the plot pane. "
  ^Menu
     labels: ('erase\redo plot\horizontal bar\vertical
      bar[1] ,
        [1]\options ...\restore defaults') withers
     lines: -#(2 4) selectors:
      #(clearPlot newPlot setHorizontal setVertical
        optionPicker initialize)
```

We will look at the *newPlot* method later. The *initialize* method is already in place; this new menu alternative gives the user a direct way to invoke it.

The *optionPicker* Method

When the user chooses the "options..." entry in the new **GraphPane** menu, we call the method *optionPicker*. This is the second-level menu we mentioned earlier. Here is the code for this method:

```
optionPicker
  "Display a menu displaying the variables which the user may
  change. If the response is not nil, perform the method that
  prompts for a new value for the selected variable." I menu I menu
  := Menu labels: ' stretch factor\bar color\bar width\bar
  spacing'
           withers
      lines: #()
     selectors: # (factor barFill barWidth barSpacing) .
   response menu popUpAt: Cursor offset for : Self
   response notNil
```

This new menu presents the following choices to the user:

- stretch factor
- bar color (we use "color" rather than "pattern" because it is more general)
- bar width
- bar spacing

The line which is third from the bottom is the workhorse of this method. It packs a lot of power into a single line of code, thanks to Smalltalk/V's built-in capability. The *popUpAt:for:* method takes a point and an object as arguments and handles three tasks in one single step:

- displays the menu at the point
- passes control to the menu
- sends the user's response, or nil if the user doesn't choose a menu response, to the named object

Note that we use the *perform:* method again. By now, you are undoubtedly beginning to gain an appreciation for its power. Here, we use it to call the method whose selector corresponds to the menu choice made by the user.

The *barFill* Method

We begin our examination of the option-changing methods with the *barFill* method which, as its name implies, lets the user choose the type of fill pattern to use in drawing the bars in the graph. Here is the code for this method:

```
barFill
  "Display a menu displaying the fill colors available for
  bar plots. If the response is not nil, accept the selected
  color." | menu response |menu := Menu
      labels: 'black\dark gray\grayMight gray' withers
       lines: #()
      selectors: #(black darkGray gray lightGray).
   response := menu popUpAt: Cursor offset,
   response notNil
     if True: [barPen mask: (Form perform: response)]
```

This method is practically self-explanatory. Again, we create a menu with the alternatives for the colors or patterns we can use to draw bars. The user makes a selection from this menu and we use *the perform:* method to define the mask for our *barPen* object.

The *barSpacing, barWidth,* and *factor* Methods

The methods that change spacing and width of the bars in the graph as well as the stretch factor by which data is scaled to the **GraphPane** are so similar that it makes sense to present them together. The only difference among them lies in the instance variable to which the user's response is assigned.

These methods present one problem that the *barFill* method did not. When you provide the user with a menu of choices from which to respond to a question, you pre-validate the user's answer. It is impossible for the user to enter an invalid or illegal response. For example, in response to the *barFill* method menu above, the user cannot answer "purple with green polka-dots." However, in the case of the spacing and width arguments, we need the user to provide a number because the program depends on this. We need a way to ensure that the user enters a number. Not only that, we have to guard against outlandishly large or small values that would result in unusable graphs.

To accomplish these purposes, we'll create a new method called *promptFor:default:validateWith:,* which will ask the user for an entry, provide a default value for the user to accept, and then validate the user's input. We'll look at this method in the next section. For now, you can see how it works in practice by examining these two methods:

```
barSpacing
    "Prompt for a new value of the spacing between bars. If the
    user's response is not nil, validate it. If it is not
    valid, tell the user, otherwise accept it."
```

```
  barSpacing :-
    self
      protptFor: 'bar spacing'
      default: barSpacing
      validateWith:
        [ :ii I ((ii isKindOf: Integer) and: [ ii > 1 ]) and:
                [ ii < 16 ] ]

barWidth
  "Prompt for a new value for the width of bars. If the user's
  response is not nil, validate it. If it is not valid, tell the
  user, otherwise accept it." barWidth : = self
      pronptFor: 'bar width'
      default: barWidth
      validateWith:
        [ :ii I ((ii isKindOf: Integer) and: [ ii > 2 ]) and:
                [ ii < 50 ] ]

factor
  "Prompt for a new value of the data stretch factor. If the
  user's response is not nil, validate it. If it is not valid,
  tell the user, otherwise accept it." factor := self
      proirptFor: 'data stretch factor'
      default: factor validateWith:
        [ :ii I (ii isKindOf: Integer) and: [ ii > 0 ] ]
```

Notice that in each of these methods, we use the new prompting method, with appropriate arguments, to ask the user for a new value for the corresponding instance variable. The validation uses the *isKindOf:* method to ensure that the entry is not only numeric but a whole number. It then checks the range of the entry to be sure it is within reasonable limits.

The *promptFondefault:validateWith;* Method

Here is the code for the method we used with *barSpacing* and *barWidth* above to prompt the user for an entry and validate the user's response:

```
pronptFor: aString default: anObject validateWith: aBlock
    "Prompt the user to supply a new value for what is
    described by aString. Present anObject as the default. If
    the user's response is nil, answer anObject. If it is
```

```
not nil, validate it. If it is not valid, tell the user, and
answer anObject, otherwise answer the user's response." I
response | response := Prorrpter
prompt: ' Enter a new value for ' , aString
defaultExpression: anObject printString.
(response isNil)
ifTrue: [ ^anObject ] .
(aBlock value: response)
ifTrue: [ ^response ]
ifFalse: [ Menu message:
        response printString , ' is not a valid ' , aString.
        "anObject ]
```

You may use this compact general-purpose method in your own programs. It uses the **Prompter** class' *prompt .'default:* method. We used other methods of this class earlier in the book, so it is no stranger to us.

Our new method is designed so that it takes three arguments. The first is a string that is used to complete the user prompt "Enter a new value for" with the appropriate characteristic or property you are prompting the user to change. The second argument is a string that represents the default value that will be used if the user doesn't alter it. The final argument is a block that this method executes. The block must return a logical value *(true or false)* so that our new method can respond accordingly.

You can now see how the. *barSpacing* method works. It presents a prompter with the label, "Enter a new value for bar spacing" and places the current value of the instance variable *barSpacing* into the prompter as a default response. If the user does anything but delete this default response, then the method checks to ensure that the user's entry is a valid integer with a value between 2 and 15. The *barWidth and factor* methods are functionally identical.

File-Based Data Retrieval and Storage

The second major change we want to make to our Plot application is to allow the user to store data elements in a disk file from which they can be retrieved. To do this, we need to modify the menu in the data pane to offer the options of loading and saving data from and to a DOS file, and we need to write the methods that will handle the file input/output processing.

The Modified *dotaMenu* Method

We only need to add two items to the menu that appears in the **TextPane** menu. Here is the code for the method as it will now appear:

```
dataMenu
    "Answer the menu for the data pane."
  ^Menu
     labels:
     ('accept\restore' , '\file
     inXfile out' ,
     '\copy\cut\paste') withers
     lines: #(2 4) selectors:
       #(accept cancel
         fileln fileOut copySelection cutSelection
        pasteSelection)
```

The *fileln* Method

Here is the code for the method that will be called when the user indicates a desire to file in some data previously stored in a file. (Note that you must add **dataPath** to the list of instance variables for the class.)

```
fileln
  "Prompt for a specification of a file containing data
  offering the last file accessed as a default. Providing
  the user responds with a non-nil, non-empty string, try
  to open a stream on the path. If this fails, prompt again;
  otherwise ask the data pane to file the data in. Finally,
  mark the data pane as modified so the 'accept' choice on
  the pane menu will respond." I path stream dataPath
  dataPane I path := Prompter
   prompt: "Enter a DOS path to a data file'
   default: dataPath. (path isNil or: [ path
   isEmpty ] )
     ifTrue: [ ^self ] .
  dataPath := path.
   stream := File pathName: dataPath.
   (stream size > 0)
```

```
    ifFalse: [ self fileIn. ^self ].
dataPane fileInFrom: stream, stream
close. dataPane dispatcher
modified: true
```

This code is fairly self-explanatory, particularly in view of its extensive comment. We use the **Prompter** class' *prompt:default:* method to ask the user for the name of the file to read. If the user has previously opened a file, we have stored its name in the instance variable *dataPath* and offer it as the default response. If the user answers with a possible file name, then we try to open a stream on it. We check the size of the stream; if it is 0, then we know that we have not been successful in opening the file for some reason and we call *the fileIn* method again to give the user a chance to recover from entering a wrong file name.

Once we have a valid stream, we use the **TextPane** *methodfileInFrom:* to refresh the contents of the data pane with the contents of the file. Then we close the stream.

The last line of the method is interesting. Recall from our earlier use of this application in its second-stage incarnation that we had to make some kind of editing change to the contents of the data pane before the "accept" menu option would work. This is how the **TextEditor** "accept" method works. It does nothing if the text hasn't changed since it was last used. If we load data from a file, the **TextPane** will not see its contents as having been edited because they have instead been replaced by other data without the user's editing intervention. To get around that problem here, our method sends the *modified:* message with a argument of *true* to the **TextPane's** dispatcher. The "accept" menu choice will now work as expected.

The *fileOut* Method

The method that writes information from the **TextPane** to a DOS file is almost identical to *the fileIn* method. Here is its code:

```
fileOut
  "Prompt for a specification of a file to receive data
  offering the last file accessed as a default. Providing
  the user responds with a non-nil, non-empty string, open a
  stream on the path and ask the data pane to file the data
  out." | path stream | path := Prompter
  prompt: 'Enter a DOS path to a data file'
  default: dataPath. (path isNil or: [ path
  isEmpty ] )
```

```
    ifTrue:  [ ^self ].
  dataPath := path.
  stream := File pathName: dataPath.
  dataPane fileOutOn: stream, stream
  close
```

This method is simpler than *the fileln* method because we don't have to worry about not finding the file (we'll create it if it doesn't exist) and we need not worry about whether the "accept" menu choice works since, presumably, the user is filing information with which he is finished working, at least for the moment. In every other respect, this method is identical to the *fileln* method.

Redrawing the Plot on User Demand

Earlier, we saw that we added a "redo plot" option to the **GraphPane** popup menu. We promised to write the corresponding *newPlot* method later. It is now time to take care of this. Here is the code for the method:

```
newPlot
  "If there is data to plot, plot it using the currently
  selected set of plot arguments." plotData notNil
    ifTrue: [ self perform: plotSelector with: plotData ]
```

This is a simple and straight-forward method. If the instance variable *plotData* — which, you may recall, is created by the *string:from:* method based on the contents of the **TextPane**—is not empty, then this method calls the method whose name is presently stored in the instance variable *plotSelector,* sending along the data as an argument. This causes the plot to be redrawn. (Note that you must modify the *String:from:* method to use *plotData* instead of *data.)*

This method makes it fast and easy for the user to change any of the characteristics of the plot and see the results of the changes. This is a significant improvement over the previous version in which the user had to make a change to the bar type, then change the data and choose the "accept" menu option from the **TextPane** menu just to redraw the graph.

The Complete Listing

Here is the complete listing of the final version of the Plot application, in fileln format.

```
Object subclass: #PlotWindow
 instanceVariableNames:
 'plotPane dataPane datapath plotData plotSelector factor '
 'barWidth bar Spacing bar Pen ' classVariableNames: ''
 poolDictionaries: ' ' !

! PlotWindow methods 1

barFill
  "Display a menu displaying the fill colors available for bar
  plots. If the response is not nil, accept the selected
  color." I menu response I menu := Menu
     labels: 'black\dark gray\grayMight gray[1] withers
      lines: #()
   selectors: #(black darkGray gray lightGray).
   response := menu popUpAt: Cursor offset, response
   notNil if True: [ bar Pen mask: (Form perform:
   response) ] !

barSpacing
  "Prompt for a new value of the spacing between bars. If the
  user's response is not nil, validate it. If it is not
  valid, tell the user, otherwise accept it." barSpacing : =
  self
      promptFor: 'bar spacing'
      default: barSpacing
      validateWith:
         [ :ii I ( (ii isKindOf: Integer) and: [ ii > 1 ]) and:
                 [ ii < 16 ] ]!

barWidth
  "Prompt for a new value of the spacing between bars. If the
  user's response is not nil, validate it. If it is not
  valid, tell the user, otherwise accept it." barWidth :=
  self
      promptFor: 'bar width'
      default: barWidth
      validateWith:
         [ :ii I ((ii isKindOf: Integer) and: [ ii > 2 ] ) and:
                 [ ii < 50 ] ]!

clearPlot
    "Erase the plot pane by giving it a new blank form. "
  plotPane form: (self plotPane: plotPane frame)!

dataMenu
    "Answer the menu for the data pane."
  ^Menu
     labels:
       ('accept\restore' ,
```

```
    '\file inXfile out' ,
    '\copy\cut\paste') withers
    lines: #(24) selectors:
      #(accept cancel
        fileln fileOut copySelection cutSelection
       pasteSelection) !

dataPane
   "Answer the contents for the data pane, initially an empty
    string."


factor
  "Prompt for a new value of the data stretch factor. If the
  user's response is not nil, validate it. If it is not valid,
  tell the user, otherwise accept it." factor := self
     promptFor: 'data stretch factor'
     default: factor validateWith:
       [ :ii I (ii isKindOf: Integer) and: [ ii > 0 ] ] !

fileln
  "Prompt for a specification of a file containing data
  offering the last file accessed as a default. Providing the
  user responds with a non-nil, non-empty string, try to open a
  stream on the path. If this fails, prompt again; otherwise
  ask the data pane to file the data in. Finally, mark the data
  pane as modified so the 'accept' choice on the pane menu will
  respond." I path stream I path := Prompter
  prompt: 'Enter a DOS path to a data file'
  default: dataPath. (path isNil or: [ path
  isEmpty ])
    ifTrue: [ ^self ]. datapath :=
  path, stream := File pathName:
  dataPath. (stream size > 0)
     ifFalse: [ self fileln. ^self ].
  dataPane filelnFrom: stream, stream
  close. dataPane dispatcher modified:
  true!

fileOut
  "Prompt for a specification of a file to receive data
  offering the last file accessed as a default. Providing the
  user responds with a non-nil, non-empty string, open a
  stream on the path and ask the data pane to file the data
  out." I path stream I path := Prompter
```

```
      prompt: 'Enter a DOS path to a data file'
      default: dataPath. (path isNil
   or: [ path isEmpty ] )
     ifTrue: [ ^self ] .
   dataPath := path.
   stream := File pathName: dataPath.
   dataPane fileOutOn: stream, stream
   close!

horizontalBar: theData
    "Draw a horizontal bar plot of theData, an ordered
    collection of integers." I form x y I
   form :•= Form width: Display width height: Display height. x
   := 0. y := 0. barPen
   defaultNib: 1 @ (barWidth * Aspect) rounded;
   direction: 0; destForm: form. theData do: [
   :value I
           barPen place: x @ y.
           barPen go: factor * value.
           y :- y + ( (barSpacing + barWidth) * Aspect) rounded ] ,
  plotPane form: form. plotPane paneScanner copyBits!

initialize
    "Initialize instance variables with default values. "
  plotSelector := thorizontalBar:.
  factor := 4. "stretch the data values by this much"
  barWidth := 2 * SysFont height. barSpacing : =
  SysFont height. barPen : = Pen new mask: Form gray!

initWindowSize
     "Answer the initial size for a PlotWindow. "
   ^ (Display boundingBox insetBy: 30 @ 25) !

newPlot
   "If there is data to plot, plot it using the currently
   selected set of plot arguments." plotData notNil
     ifTrue: [ self perform: plotSelector with: plotData ] !

open
   "Open a PlotWindow. Define the pane sizes and behavior, and
   schedule the window." I topPane I self initialize. topPane
   := TopPane new. topPane
     model: self;
     minimumSize: self initWindowSize extent;
     label: 'PlotWindow';
```

```
  rightIcons: #(resize collapse) .
  topPane addSubpane: (dataPane :=
  TextPane new model: self; menu:
  #dataMenu; name: #dataPane; change:
  ttstring:from: ; framingBlock:
    [ :box I box origin extent: 50 @ box height ] ) . topPane
  addSubpane: (plotPane : = GraphPane new model: self; menu:
  #plotMenu; name: ftplotPane:; framingBlock:
           [ :box I (box origin + (50 @ 0) ) corner: box corner ]).
  topPane dispatcher open scheduleWindow!

optionPicker
   "Display a menu of the variables which the user may change. If
   response is not nil, perform the method that prompts for a new
   value for the selected variable. " i menu response I menu := Menu
   labels: ' stretch factorXbar colorXbar widthXbar spacing' withers
       lines: #()
      selectors: #(factor barFill barWidth barSpacing).
    response := menu popUpAt: Cursor offset, response
    notNil
      ifTrue: [ self perform: response ] !

plotMenu
     "Answer the menu for the plot pane."
   ^Menu
      labels: ('eraseXredo plotXhorizontal barXvertical
       bar[1] ,
         [1]Xoptions . . . \restore defaults') withers
      lines: #(2 4) selectors:
       #(clearPlot newPlot setHorizontal setVertical
         optionPicker initialize)!

plotPane: aRect
    "Make a blank form the size of the screen. Display the form in aRect
     on the screen. Answer the form." I blankForm I (blankForm :=
   Form width: Display width height: Display height)
     displayAt: aRect origin clippingBox: aRect.
   ^blankForm!

promptFor: aString default: anObject validateWith: aBlock
     "Prompt the user to supply a new value for what is
     described by aString. Present anObject as the default.
```

```
    If the user's response is nil, answer anObject. If it is not
    nil, validate it. If not valid, tell the user, and answer
    anObject, otherwise answer the user's response." I response I
    response := Prompter
    prompt: 'Enter a new value for ' , aString
    defaultExpression: anObject printString.
    (response isNil)
    ifTrue: [ ^anObject ] .
    (aBlock value: response)
    ifTrue: [ ^response ]
    ifFalse: [ Menu message:
           response printString , ' is not a valid ' , aString.
           ^anObject ] !

setHori zontal
 "Change the plot type to horizontal bar."
 plotSelector := #horizontalBar: .

setVertical
 "Change the plot type to vertical bar."
 plotSelector := ftverticalBar: .

string: aString from: aDispatcher
 "The data pane's contents have been changed. Convert the new
 data into an ordered collection of integers and make a new
 plot. Answer true." I input token I input := aString asStream.
 plotData := OrderedCollection new. t input atEnd ] whileFalse:
 [
       token := input nextWord,
       token notNil
        ifTrue: [ plotData addLast: token aslnteger ] ].
 self newPlot. ^true!

verticalBar: theData
   "Draw a vertical bar plot of theData, an ordered collection
    of integers."
  I form x y I
  form := Form width: Display width height: Display height.
  x := 0.
  y := form height.
  barPen
  defaultNib: barWidth @ 1;
  direction: 270; destForm:
  form. theData do: [ :value
  I
          barPen place: x @ y.
          barPen go: factor * value.
          x := x + barSpacing + barWidth ] .
```

```
plotPane
   form: form;
  scrollUp: plotPane frame height - form height.
plotPane paneScanner copyBits! !
```

10
The Text World

Introduction

In this chapter, we will look at the key classes involved in manipulating text on the screen of a Smalltalk application. In one sense, text is graphic because each character, symbol, and space is really made up of a collection of bits. But in this chapter, we will focus on the text as a collection of characters that has meaning in some context. This does not mean, however, that we will ignore the bit-mapped image that makes up its appearance on the screen. We will begin this chapter with an overview discussion of how text is displayed, represented, and manipulated in Smalltalk/V. Then we'll take a look at four of the six classes that play some role in the text world of Smalltalk. As we discuss each class, we will point out important instance variables and methods in addition to describing the overall role of the class in the process of text management.

Behind the Text in Smalltalk

All editable text in Smalltalk/V is displayed in an instance of the class **TextPane.** (Text which is not intended to be edited can be drawn onto any graphical context including a **GraphPane** or the **Display.)** Identified with this pane, text is represented as an **OrderedCollection of Strings.** Each character in each string can be identified by a **Point,** whose *x* value is its index within the string and whose *y* value is the index of the string in the **OrderedCollection.** Thus the fourth character in the third string of the array has a **Point** location of (4,3).

As we know from previous experience, it is easiest to display text in an instance **of TextPane.** Associated with each such instance is an instance of the class **TextEditor.** This latter class processes user interaction with the pane, such as cursor movement, scrolling commands, menu requests, text selection, and editing (copy, cut, paste, and related operations) commands.

It is assisted in this process by an instance of the class **StringModel,** which actually carries out the editing operations. This whole process comes about as a result of the fact that whenever you create a new instance of the class **TextPane** it automatically gives you a **TextEditor** as its dispatcher, a **CharacterScanner** (discussed below) as its pane scanner, a **StringModel** as its *textHolder* and a **TextSelection** (also discussed below) as its *selection.* These classes are automatically associated with any instance of the class **TextPane;** you don't have to do anything to get them, all you have to do is use them as needed.

Text is represented in Smalltalk in two separate but related ways. First, every character has an ASCII value associated with it. This value is standard across all computer systems that use it, which includes almost all desktop and minicomputers. On all of these systems, for example, the ASCII value 64 always means a capital letter "A." ASCII values range from 0 to 255. Second, every character in Smalltalk has associated with it a bitmapped image that represents its appearance in the **TextPane** on the computer screen. This bitmap depends on two major factors: the character's ASCII value and the font in use (including its size and style, if appropriate).

Conversion between ASCII character codes and their bitmap equivalents is handled by an instance of the class **CharacterScanner,** a subclass of the graphic class **BitBlt** which we used in Chapters 8 and 9. The current font in use on the system is stored in an instance variable of this class. The font is used as the source form and the **TextPane** is the destination form. With that information, you can see how the drawing of characters takes place in a Smalltalk/V system.

Whenever users want to edit existing text in a **TextPane,** they follow the Smalltalk paradigm of select-then-operate. In other words, they first select the text on which they wish to perform some editing operation, then invoke the operation, usually by selecting it from a menu but also possibly by other means (e.g., typing to replace selected text rather than first cutting it by means of a menu selection). An instance of the class **TextSelection** keeps track of the beginning and ending points of the user's text selection, and handles the non-editing manipulation of the text (e.g., its graying when the user interface calls for selected text to be grayed, as when the window is no longer active).

With this overview in mind, let's take a look at each of the classes that can be involved in the display and editing of text in a Smalltalk/V **TextPane.** To summarize, these classes are:

- **TextPane**
- **TextEditor**
- **StringModel**
- **CharacterScanner**
- **TextSelection**
- **String**

We will examine the first four of these classes in some detail. You will seldom need to worry about the other two — **TextSelection and String,** so we won't discuss their instance variables or methods.

The Class *TextPane*

We spent some time examining the class **TextPane** in Chapter 6 in the context of the MPD paradigm and methods that are common to all types of panes in Smalltalk/V. If you are unclear about the creation and management of such panes, it might be a good idea to review Chapter 6 before proceeding with this discussion.

In this chapter, we will focus on several methods that we did not explore in Chapter 6 but which you may find useful when you create text-based applications. Several of these methods will appear in Chapter 11 when we build a data entry form class as an example of how to handle text-based applications. Specifically, in this chapter, we'll look at methods that handle the addition of text to the end of a **TextPane's** current contents, scroll the pane's contents to show a designated portion of the text, and handle the selection and de-selection of text in the pane.

Methods for Appending Text

The class **TextPane** defines two methods for adding text to the end of the contents of one of its instances: *appendChar:* and *appendText:*. The first takes a single character as an argument and places that character at the end of the text in the pane. The second takes a string as an argument and places its contents at the end of the text in the pane.

Recall that, since all text editing other than text entry is handled by other classes in the Smalltalk/V text world, this class need only concern itself with appending text to the current contents of the pane.

Methods for Scrolling the Text

There are two ways that a text-based application might want to alter the scrolling status of a **TextPane's** contents aside from the user's direct modification of the scrolling via the pane's built-in scrollbars.

First, we might want to scroll the text in the pane so that the end of the text is visible. We may want to do this, for example, when we first open a pane

which already has some content and where the user's expected first desire is to resume or begin entering text. To handle this, the class **TextPane** provides *\hQforceEndOntoDisplay* method, which does what its name suggests. Second, we may want to scroll so that currently selected text is visible in the pane. This task is handled via the *forceSelectionOntoDisplay* method.

Methods Related to Selection of Text

There are two types of selection in a **TextPane.** The one we think of most often involves the selection and highlighting of one or more characters in the pane's contents. The second, less obvious selection, is called the gap selection and arises when no characters have been selected. In that case, the selection refers to the position of the I-beam cursor in the **TextPane** indicating where the next character typed will be placed. To select text in a **TextPane,** you can use one of three methods:

- *selectAll,* which, as its name implies, selects all of the text in the pane
- *selectFrom:to:,* which takes two points as arguments and selects all of the text between those two points (where the points are the x-y coordinates of a character as described earlier in the chapter)
- *selectTo:,* which extends an existing selection to the point provided as an argument

To position the gap selection (which is the same thing as positioning the I-beam cursor), you can use one of these three methods:

- *select After:,* which places the gap selection after the point supplied as an argument
- *selectBefore:*, which places the gap selection before the point supplied as an argument
- *selectAtEnd,* which places the gap selection at the end of the **TextPane's** contents

The Class *TextEditor*

Since the purpose of the class **TextEditor** is to process user input, it is seldom necessary for you to do anything to its methods other than use them. Many of its methods, such as those that actually place text in the pane, are

used by other methods in the Smalltalk/V text world and you will be unlikely ever to use these methods.

Still, there are five methods in this class that are worth exploring and knowing something about. Two deal with keeping track of the status of the contents of the pane, two deal with special characters you might have a need to place in the text, and one deals with zooming the pane to full-screen size.

Tracking the Status of Text

It is often useful to know whether the contents of a **TextPane** have been modified since they were last saved or since some other operation (such as a database update) was performed. Smalltalk handles the automatic tracking of this status for you but if your program needs either to know the status of the text or to change it for some reason that is not being monitored by Smalltalk, you can use the *modified* and *modified:* methods in class **Text-Editor** to do so.

As you can tell by their names, *modified returns a.* Boolean value indicating whether the contents of the pane have been modified since the last time the text was saved, while *modified:* sets this value to be *true or false* depending on the argument.

Putting Special Characters in Text

You can put a carriage return, a space or a tab at the end of the current contents of a **TextPane** using the methods called cr, *space,* and *tab.* This is a useful thing to be able to do when, for example, you are re-opening some formerly saved text and need to place one of these characters at the end to prepare the contents for further additions by the user or additional text to be loaded from a file.

Zooming the Pane

You have undoubtedly noticed that a **TextPane,** unique among the sub-pane types, can be zoomed so that its contents occupy the entire screen. You may, for example, have been surprised early in your Smalltalk experience to click on the Zoom icon in the Class Hierarchy Browser and see only the text pane and not the entire window zoom to full-screen size.

In a case where you may wish to give the user the full screen in which to edit text without requiring that the user zoom the pane manually, you can send the pane the *zoom* message.

The Class *StringModel*

For the most part, you will find yourself calling on the methods in the class **StringModel** as you build text-based applications in Smalltalk/V. Seldom will you have to override these methods because their behavior is relatively generic and in general encompasses the actions users expect when they edit text in a **TextPane.**

A few of the methods of this class are interesting enough to spend a few moments discussing. These methods can be grouped into the following broad categories:

- information-gathering methods
- searching methods
- editing methods

Information-Gathering Methods

Some information about the contents of a **TextPane** may be useful to you in designing and building text-intensive applications. The class **StringModel** includes three methods that reveal data about the contents of the pane.

The *lineAt:* method takes an integer as an argument and answers the string contained in the line indexed by that integer. Recall that a **StringModel** holds its text in an array of lines.

You can find out how many lines of text are in the **StringModel** with the *totalLength* method.

One particularly intriguing and potentially useful method is the *maxLineBetween:and:* method. The two arguments to this method are both integers that represent line numbers in the array of strings. The method answers the length of the longest line between the two lines provided as arguments. This would be useful, for example, in dynamically adjusting the width of the **TextPane** to show all, or as much as possible, of the contents of the pane's text lines.

Searching Methods

Two methods in the **StringModel** class are designed to search for a specified string. The *searchFrom:for:* method takes a selection as its first argument and a string, character, or text pattern for its second argument. It begins searching from the end of the text selection until it finds the first instance of the pattern in the second argument and then returns that matched selection if it finds one. It returns nil if it fails to find a match.

To search in reverse in the pane, use the *searchBackFrom:for:* method, which is identical to the *searchFrom:for:* method except in the direction in which it conducts its search, and that it starts its search at the beginning of the *selection.* Both methods search each line from left to right, however.

Editing Methods

There are three editing methods that are of possible interest to you if you are building a text-based application. Most of the common editing operations involve some combination of these methods. Earlier in the chapter we said that most editing operations operate on a selection of text. That selection is passed as an argument to these methods.

The *delete:* method deletes the text contained in the selection and leaves the cursor positioned just before the selection's location.

The other two editing methods replace text, *replace:withChar:* replaces the text selection supplied as the first argument with a single character supplied as the second. The other method, *replace:withText:,* replaces the selection with a string supplied as the second argument.

Both methods inform the **TextPane** of the change in contents and both answer a point which describes the position of the last replacement character.

The Class *CharacterScanner*

Like the classes **TextEditor and StringModel,** you will rarely interact directly with a **CharacterScanner.** You will usually create a specific instance of this class in a text-based application, but for the most part its methods handle their processing in the standard, accepted way and therefore are seldom candidates for your own specialized methods.

Five of this class' methods can be used to set up or modify its visible characteristics, such as its foreground and background colors or the font it uses. Four of its methods involve the display of text in the clipping rectangle of the **TextPane** with which the **CharacterScanner** is associated. Another two are available to enable you to erase some portion of that pane.

Methods to Control Appearance

When you want to initialize an instance of class **CharacterScanner,** you can use one of two methods. The one we'll look at because it is the more common of the two is *initalize:font:*. This method takes two arguments. The first is the rectangle which you wish to become the clipping rectangle of the scanner. The second is the font you want to use to display text within the scanner's **TextPane.** (We discussed clipping rectangles in conjunction with graphics in Chapter 8.)

Once a CharacterScanner has been created and initialized, you can change the font it uses with the *setFont:* method or its color scheme with the *setForeColor:backColor:* method. This method requires two colors as arguments. When you initialize an instance of **CharacterScanner,** Smalltalk/V defaults these values to black as the foreground and white as the background.

To invert some or all of the contents of the frame of the **TextPane,** you can use the method *reverse:*. This method takes a rectangle as an argument and reverses the color of the text within that rectangle. For example, if these colors have not been changed since the **CharacterScanner** was initialized, this method causes the text to appear to be white characters on a black background. You can undo the effect of the *reverse:* method with the *recover:* method. This method also takes a rectangle as its argument.

If you wish to change the font associated with a **CharacterScanner,** you send it the *setFont:* message along with the name or other identifier of the font you wish it to use. Since the system does not support multiple fonts in a single **TextPane,** this message changes the font of all text already displayed in the pane in addition to becoming the font in which all newly typed information will be displayed.

Methods to Display Text

Of the four methods available to a **CharacterScanner** object to cause it to display text, three are quite similar to one another and the fourth operates slightly differently. The three similar methods are:

- *display:at:*
- *display:from: at:*
- *display:from: to: at:*

Since the last of these is the most robust, we'll begin by explaining it. You can then see quickly how the other two differ. The *display:from:to:at:* method requires four arguments. The first is the identifier of the string to be completely or partially displayed. The next two arguments tell the **CharacterScanner** the index positions in the string array at which to begin and end the text to be displayed. The final argument is a point that defines the location in the frame of the receiver at which the display of the string is to start.

To display the entire string, then, you would use the first method since it doesn't require a starting or ending position in the string. To display from some point other than the beginning of the string to the end of the string, you would use the second method.

The most common use of these methods is to replace a portion of a displayed line with some other text. Since the present line or portion of the line may not be the same length as the replacement text, the *display:from:at:* method adds one bit of behavior the other two methods do not involve: it blanks the rest of the line from where the display of the replacement text ends.

This leads to the need for the fourth text-displaying method we'll examine, *show:from:at:*. This method behaves the same as *display:from:at:* except that it does not alter the other text in the string.

Methods to Blank Portions of the Pane

You sometimes wish to remove text from a **TextPane** display, perhaps without affecting the contents of the stored text itself. To do this, you can use either the *blank:width:* method or the *blankRestFrom:* method.

The first of these methods blanks all or part of a line by painting to the background color a rectangle whose origin is supplied as the first argument and whose width is calculated from the second argument, an integer. The rectangle has the height of the current font.

If you want to blank the bottom portion of a frame starting at some specific known row of text, you can use the *blankRestFrom:* method, which takes a single integer argument. This integer is the row starting from which you wish to blank the frame.

11
The Fifth Project: A Form Designer

Introduction

In this chapter we will put into practice some of the theory of Smalltalk/V text classes and methods we learned about in Chapter 10. This application will also prove useful for inclusion in or use with other applications you may build.

Project Overview

As we saw early in our exploration of Smalltalk/V, the primary built-in means by which you can pose questions to the user and deal with user responses is the class **Prompter.** This class, combined with the judicious use of menus, provides means by which you can ask users to provide information to the program. These answers can either be selections from a pre-determined list (similar to a menu) or an editing rectangle in which the user types a response, perhaps selecting an optional default answer supplied by the program. /

This method of user interaction is severely limiting, because the user can only see and respond to one question at a time. In some cases, users may wish to change an earlier answer or plan later responses if they knew what other questions they were going to be called upon to answer during a session with the program.

The project in this chapter simulates a blank-form style of data entry that allows your programs to pose a number of questions that require text answers. The user can move around among the fields in the form and select an "OK" or "Cancel" button to indicate that all answers have been provided and may be processed by the program. Figure 11-1 shows a typical form that might be created by an application using this project's classes. Since the

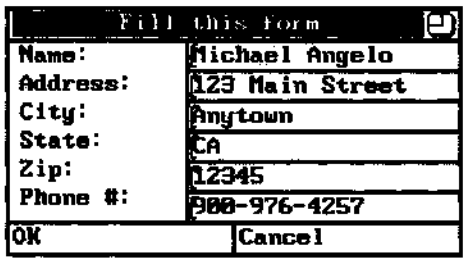

Figure 11-1. Typical Form Created by Project

purpose of this project is to provide *a* framework for the creation of such forms, no two forms will necessarily be identical.

Designing the Project

We will begin by stating the purpose of the program, then using that statement to identify the objects the application will have to contain. Finally, we'll determine the responsibilities of each of these objects in the application.

Statement of Purpose

This application is given two arrays of strings as input. One array holds questions, or name fields, which the user may not edit and which identify for the user the type of input needed. The second array holds answers or entry fields into which the user will type responses to the questions or labels. Given this input, the application object will create a nonmodal (i.e., non-preemptive) window in which the user can edit answers to all entries and then use buttons to accept or cancel the answers provided.

We should note that this project is not an application *per se.* Rather, it is a complex type of window **(TopPane)** which supports form-like properties and behavior. The only real application we will build is a test application so that we can demonstrate and test the project.

Since we are going to be focusing on the pane and its behavior and since our objective is to create a form-like data entry interface project, we'll refer to the new class we're creating as **FormPane.**

Defining the Objects

From the above description of this project, we can conclude that the application requires the following objects:

- application
- window

« name (or question) fields which may not be edited by the user

- entry (or answer) fields which may be edited by the user
- buttons

Object Responsibilities

Let us now examine each of these objects in turn and define the responsibilities we will assign to each.

Application Object Responsibilities

As is most often the case in Smalltalk applications, the application object will act as the model to the window, which contains all of the fields. It will also initialize our special type of window, which will automatically format the fields and buttons correctly.

What may not be obvious from this brief description is that the application object will not play its usual role of acting as a model to the sub-panes in the **FormPane** window. We are creating a new type **of TopPane** with complex behavior that emulates an electronic data entry form. We want the application that uses this **FormPane** to be able to do so without any understanding of the sub-panes and their behaviors.

This results in an intriguing design problem. The sub-panes need a model and if we're not going to call upon the application object for this task, we must decide on some other object to handle it. Your first inclination (which matches ours) would probably be to permit the **FormPane** to be the model to its sub-panes. However, you would find (as we did) that this approach does not work. The Smalltalk/V MPD architecture we discussed in Chapter 6 makes the assumption that the model to any **Pane** will not be another **Pane.** This requires us to create an object that will have the assignments of acting

as the model to the sub-panes and playing "go-between" or intermediary between the sub-panes and the **FormPane.** As it turns out, this approach has a major benefit: it allows us to divide the many methods of this project into more refined categories (methods for the **FormPane** itself, and methods for managing and interacting with its sub-panes).

We will therefore create a new object called **FormIntermediary.** The role pf this object is to act as a model to the sub-panes and to mediate between them and their **FormPane** so this class will end up assuming most of the knowledge that we might otherwise place into the **FormPane** class directly.

Window Object Responsibilities

The window object will be an instance of class **TopPane.** It will have two basic responsibilities.

First, it will automatically calculate the correct placement for all of the field and button sub-panes within its borders based on a number of factors.

Second, it will be responsible for informing the model of changes that take place as a result of user responses.

Name Field Object Responsibilities

The name or question field objects have no responsibilities in this application. Their only role is to serve as a guidepost to assist the user in data entry. They are therefore entirely passive.

Since these fields are not going to be subject to editing, we can use a **GraphPane** for them. As it happens, this class includes some behavior — for example, scrolling — that is undesirable for our purposes. If, for example, we allowed labels in the **GraphPane** to scroll, the alignment between the questions or labels and the fields into which their associated data will be entered may be lost. We will have to override some of these methods.

Entry Field Object Responsibilities

The entry or answer field objects must support the editing of text by the user, restoring to a default value if one is supplied, and zooming (to allow long answers to questions without destroying the integrity and appearance of the form).

General Approach

Based on the above analysis, we can see that we will create this application by sub-classing **TopPane** for our **FormPane** class, giving the new class the special behavior our project requires. The editable text fields (answer or

entry fields) can be created from the existing **TextPane** class we discussed in Chapter 10. The "OK" and "Cancel" buttons can be created as one-item **ListPane** objects just as the Counter project of Chapter 5 and the CHB itself handle this task. The **FormIntermediary** class will be a sub-class of class **Object.**

Knowledge Needed

The protocol for interaction between our **FormPane** and its model (our application) will involve the application setting arguments for the **FormPane.** This means that we must identify the knowledge our **FormPane** will require. These in turn become candidates for instance variables.

The FormPane needs to know one thing that is peculiar to our project. We must add an instance variable to contain information about the instance of the new **FormIntermediary** class in handling its interaction with the sub-panes. So we need a new instance variable called *intermediary* to contain this information.

The rest of the specialized knowledge needed in our project can be placed in the **FormIntermediary** class and consists of the following information (or instance variables):

- *formPane,* which holds the name of the instance of **FormPane** to which the **FormIntermediary** instance is attached or the **FormPane** itself.
- *questionStrings,* which we will create as an **IndexedCollection** of strings that will hold the name or question fields.
- *defaultAnswerStrings,* another **IndexedCollection** of strings to hold default answers for the entry fields where they exist.
- *textFieldPanes,* yet another **IndexedCollection,** this one containing **TextPanes** representing the entry fields so that we can query them for user-entered values when processing is complete and we must inform the **FormIntermediary's** dependents of the results of the data entry.

Building the Project

To build this project, we will have to take a number of steps. Their order is, for the most part, insignificant, but dividing our work into these steps makes it possible to get a prototype up and working quickly and then to modify its various key behavioral characteristics individually.

The steps we will take are as follows (in the order in which we will discuss them):

1. Create the new classes.
2. Write skeletal interaction methods for the new classes.
3. Build a simple test application.
4. Write the methods to place the sub-panes in the **FormPane.**
5. Install **TextPanes** for each editable entry field.
6. Modify undesirable behavior in our superclasses.
7. Add some capabilities at the **FormPane** menu level.
8. Format the name (question) fields.
9. Write a more complex test application.
10. Disable scrolling in the **GraphPane.**

Creating the New Classes

We begin our construction of this project by building the two new classes we've identified — **FormIntermediary and FormPane.** To create **FormIntermediary,** we'll sub-class **Object.** Find **Object in** the CHB and sub-class it to create the new class by editing the new class template furnished by Smalltalk/V so it appears as follows:

```
Object subclass: #FormIntermediary
    instanceVariableNames:
    'f ormPane questionStrings def aultAnswerStrings textFieldPanes '
    classVariableNames: ' ' poolDictionaries: ' '
```

To create **FormPane,** find **TopPane** in the CHB and sub-class it to create the new class by editing the new class template furnished by Smalltalk/V so it appears as follows:

```
TopPane subclass: #FormPane
    instanceVariableNames: '
    intermediary '
    classVariableNames: ' '
    poolDictionaries: ' '
```

Skeletal Interaction Methods

With our two key classes constructed, our next step is to provide them with minimal interactive functionality. By building only the basic methods that support providing the **FormPane** with the collections of question and answer strings needed to handle communication, we can get a first version of the program working relatively quickly. This is almost always a sound design idea, particularly when you are constructing an application of any reasonable size.

As we saw briefly in Chapter 6, interaction among the components in a Smalltalk/V program with its MPD model is made highly flexible by the design of the system. This design does not use hard-coded message selectors so that the means of interaction between elements of the application are predetermined. The selectors to be used for such tasks as informing a pane's model of a change in the pane could have any names and accomplish any additional purpose other than simple model notification. Naming and providing these selectors can be thought of as setting arguments. Our project needs to set a number of characteristics. These Include:

- *label:*
- *addSubpane:*
- *rightlcons:*
- *leftlcons:*
- *open*
- *close*
- *model:*

In addition, because of the somewhat unusual relationship among components in our project, we will create two other methods normally associated with sub-panes: *name:* and *change:*. These methods will perform for a **FormPane** tasks similar to those they normally handle for a **SubPane.**

From this brief discussion, we can already see that our definition for the class **FormPane** is not adequate because it does not include two instance variables needed for this design, namely *name* and *changeSelector* to accommodate the last need we discussed. A quick examination of the Encyclopedia of Classes or of the class **TopPane** in the CHB reveals that these two methods are not inherited from the **FormPane** superclass. So you should now open the class definition for **FormPane** and edit the definition to add two more instance variables. The finished class definition should look like this:

```
TopPane subclass: ttFormPane
 instanceVariableNames:
    [1] intermediary name changeSelector '
 classVariableNames: ''
 poolDictionaries: ' '
```

Now we are ready to write the first methods for the class **FormPane.** We'll start, somewhat but not entirely arbitrarily, with an *initialize* method. (You'll note that we frequently begin with an inherited method for our work even though we know we'll ultimately be overriding its behavior. This points up another advantage of OOP with a strong class library like Smalltalk/V's: you get a good deal of default behavior that, while it may not be what you ultimately desire, it gives you a place to begin with a working application.) Here is the code for this first method:

```
initialize
   intermediary := FormIntermediary new.
   intermediary formPane: self, super
   initialize
```

Notice that we place the **super initialize** line last in this *initialize* method. This is a case where doing so is not only acceptable but preferable to the somewhat more frequent approach of placing it first in an *initialize* method. It is acceptable because our *initialize* method does nothing with the existing instance variables of the superclass **TopPane,** so we need not guard against their being initialized to some value other than the one we need in our application. It is preferable in this situation because by putting this message last in our *initialize* method, we ensure that our method returns the same result as that which was returned before we created our own new class.

Notice in the second line of the *initialize* method, we send the newly created **FormPane** object *aformPane:* message. We do this so that we can give this object a name by which we can refer to it elsewhere in the system. Recall that we defined an instance variable *formPane* in the class definition for **FormIntermediary.** We need to refer to the **FormPane** (which is a sub-class **of TopPane)** several times throughout the application as we dynamically work with its sub-panes and their contents. In the class **FormIntermediary,** then, add the following method:

```
formPane: a FormPane
   formPane := a FormPane
```

This will initialize the instance variable *formPane* appropriately when the **FormPane** is created for an application.

The *name:* method will allow us to provide a selector to be used as the name of the **FormPane** and also as a message to the model so it has a way of retrieving information needed by the **FormPane** when it is opened. The method named by this selector should be able to return an instance of the

class **Association** whose key is the collection of questions and whose value is the corresponding collection of default answers or values. Here is the code for the *name:* method:

```
name: aSelector
    "Selector used as the name of the FormPane, and to get the
    initial information displayed in the pane." name :=
    aSelector
```

The *change:* method allows us to provide the name of a selector that will serve as a message to the model to inform it of changes that take place in the pane. The method named by this selector should be able to take an argument that will be an **IndexedCollection** containing the values entered by the user in the **FormPane.** Here is the code for this method:

```
change: aSelector
    "Used as a message to the model to inform it of a change in the
    FormPane. Must take an argument of the FormPane values. "
    changeSelector := aSelector
```

Following the normal MPD paradigm, we now need to use our *name* selector method to handle the initialization of the strings of questions and default answers *(questionStrings* and *defaultAnswerStrings).* We cannot handle these initializations from an *initialize* method because class **TopPane** initializes all of its instances immediately after they are created. This would cause an interesting problem here since **FormPane** is not in a position to be initialized immediately after its creation; it lacks a *name* selector, a *change* selector, and even a model. Therefore, we must find a place later in the processing — after these selectors have been provided to the **FormPane** but before the sub-panes are installed in the **FormPane,** to initialize these instance variables.

We will create a special method to handle this task and separate it from the *open* method that will normally set up a new FormPane because this makes it easier to sub-class **FormPane** and to initialize these variables from other places in the application if that were to become necessary. The effect is to hide implementation detail, which, as we know, is an objective of OOP.

Here, then, is the code for the *initializeFields* method that will set up the two arrays of strings for questions and default answers:

```
initializeFields
    "Queries the model and informs the FormIntermediary."
    I anAssociation I
    anAssociation := model perform: name.
    intermediary questionStrings: anAssociation key.
    intermediary defaultAnswerStrings: anAssociation value
```

We also need to set the instance variables of a **FormIntermediary** instance, so we add the following two methods:

```
questionStrings: indexedStrings
   questionStrings := indexedStrings

defaultAnswerStrings: indexedStrings
   defaultAnswerStrings := indexedStrings
```

As we have seen in our previous projects, a programming custom in Smalltalk is to have an application that creates a **TopPane** — as most do — explicitly install that pane's sub-panes. We can adapt that idea here and include in the **FormPane** class a method with which it can configure itself. Eventually, this method will have to become fairly complex as it calculates the positioning of each of its potentially numerous sub-panes. For this first pass at this important method, though, we'll design a minimal configuration process that simply enables us to test its basic behaviors. Here's the method:

```
configureSubpanes
   "Format the layout of the window's components."
   self initializeFields.
   self addSubpane: (ListPane new model: intermediary;
       name: #okBu1 on;
       change: #okButton:;
       framingRatio: (0 @ 0 extent: (1/2) @ 1)). self
   addSubpane: (ListPane new model: intermediary;
       name: #cancelButton;
       change: ttcancelButton:;
       framingRatio: ((1/2) @ 0 extent: (1/2) @ 1))
```

This method first calls the *initializeFields* method we just created. Then it adds the two buttons — "OK" and "Cancel" — as single-element **ListPane** instances. We *use framingRatio:* to make the sizes of the buttons relative to those of the window. Later, we'll see that we need to change this approach but for our first draft of the project, this technique is easier and certainly adequate.

These buttons need methods that will be invoked when the user clicks on them. If the user selects the "OK" button, the button method will notify the model of the user's responses. If the user clicks the "Cancel" button, no update is needed. These methods will be part **of FormIntermediary,** since we designed this class for this specific purpose. Also, this new class is the model to these button sub-panes. Each of these methods requires an argument, since all **ListPane** objects require arguments for their change methods. Since our **ListPane** objects each have only one element, the argument we pass is a mere formality and we will ignore its value.

The behavior of the "OK" button involves extracting the **contents** of each entry field. The fields return their contents as strings, which are placed into a collection before passing them to **FormPane.** Eventually, the **FormPane** object will pass this information to its model, the application.

Here is the code for the method for the "OK" button:

```
okButton: dontCare
    "Inform formPane of the values in the question/field subpanes.
    As a model to a ListPane, it must take an argument of the
    list index. We ignore it because the list only has a single item.
    I result I
    result := OrderedCollection new.
    textFieldPanes notNil
        ifTrue: [textFieldPanes do:
            [:aSubpane I
            result add: aSubpane contents]].
    formPane takeResult: result
```

As you can see, this method is quite straightforward. The core of its behavior is the *do:* construct that loops through all of the elements of the *textFieldPanes* and adds their contents to the **OrderedCollection** called *result.*

If the user cancels a FormPane, then we need to send the result *nil* along to the model. Here is the code that will accomplish that:

```
cancelButton: dontCare
    "Because the user has canceled, we provide nil as the result. The
    model to formPane should know that nil indicates that no values
    were changed, so use what it has as the default if desired."
    formPane takeResult: nil
```

Now we need a method for the class **FormPane** called *takeResult:* that deals with the information passed to it by the **FormIntermediary** button methods we just built. Here is its code:

```
takeResult: result
    "Inform the model of the change and result."
    model perform: changeSelector with: result,
    self dispatcher close
```

The *powerful perform:* method takes care of handling the processing of the result for us. The last line is a variation on a theme we see each time we create a new **TopPane.** Rather than opening a dispatcher, this line closes it. (This is the same method used to close a **TopPane** when the user clicks the window's close icon.) This ensures that the process of closing the window in our code will be identical to the processing Smalltalk/V undertakes.

Finally, before we have a working rudimentary project, we need to set up methods in **FormIntermediary** that will inform the button **ListPane** instances of their initial contents (i.e., give them their name labels if we think of them as buttons). These methods are pretty simple:

```
okButton
    "Label for the OK button."
```

```
cancelButton
    "Label for the Cancel button.
    ^#(Cancel)
```

This completes our basic functioning project. Although it does not yet do any of the **TextPane** processing that is eventually at the heart of our design, it does enable us to build a test application which will provide a test case for our implementation. That is our next task.

Building the Test Application

As will most often be the case with applications, we'll start by sub-classing **Object** to create the application's class. Find **Object** in the CHB and sub-class it to create the **FormApp** class. Edit the new class template provided so that it appears as follows:

```
Object subclass: #FormApp
  instanceVariableNames:
    ' questionStrings defaultAnswerStrings answerStrings '
  classVariableNames: '' poolDictionaries: ''
```

Now let's add some methods to make our **FormApp** do something interesting and useful. As we envisioned it earlier, a form application would:

- open a **FormPane** with some questions and some default answers
- be aware of the collection of actual answers and do something with it

Notice that this implies two separate collections for answers, one for the defaults and one for the user's actual replies. It might seem logical to keep just one collection, substituting the user's actual responses for default answers as they are supplied. But that would be too inflexible. What if you want to compare the user's actual answers with the default replies to see what fields in a database need to be updated, for example?

We'll begin by adding our by-now-traditional *example* metamethod. Remember that this must be a class method, while all of the others we'll create here will be instance methods.

```
example
    "FormApp example."
    | aFormApp |
    aFormApp := self new.
    aFormApp openOn: #('Name: ' 'Address: ' 'Phone #: ')
        defaultAnswers: #('Fred' '123 Main Street' '900-976-4257')
        label: 'Fill this form'.
    aFormApp open
```

The key message in this method is *openOn: default Answers :label:,* which takes three arguments:

- an array of labels or questions that act as prompts to guide user input
- an array of strings that are the default responses to those questions
- a label that provides an overall prompt or title for the form

Next, let's build the **openOn:** method. (Its name follows the Smalltalk convention for naming such methods when it takes an argument to specify what is to be opened. If no argument were necessary, we would use the more familiar *open* name.) Here is the code; it should be quite familiar to you by now:

```
openOn: questions defaultAnswers: answers label: aLabel
    "Open a FormPane with the specified questions, answers, and label."
    I aFormPane I
   questionStrings := questions.
   defaultAnswerStrings := answers.
   aFormPane : = FormPane new
       model: self;
       label: aLabel;
       name: #formlnfo;
       change: #userResponse:;
       yourself.
   aFormPane configureSubpanes. aFormPane
   dispatcher open scheduleWindow
```

The only new element here is the use of *yourself in* the last line of the cascade of messages that creates *aFormPane.We* include that message to insure that *aFormPane* gets bound to the instance created by the *new* message, rather than simply assuming that *change:* will return the right object for our needs. We notice in the above method that we called the *name* method for the *openOn:* method *formlnfo*. Here is the code for that method:

```
formlnfo
   "Returns an association of questions and answers for the FormPane."
   ^Association key: questionStrings value: defaultAnswerStrings
```

We make use of the class **Association** here to set up a pair of parallel structures (actually contained in one object) matching name fields and entry fields. This device is clearly made to order for our need to keep the two lists "in sync" with one another.

Like all programmers, we have a curiosity about the success of our programming even before we're really ready to do anything with the output. So in our **FormApp** application, we'll use the Smalltalk/V Transcript window to print the results of our data entry. Obviously, if we were writing an application for some purpose other than testing, we would be doing something more useful with the user's entries in the **FormPane,** but the important

thing for our application is that we confirm that it retrieves the results correctly. Here is the code for our *userResponse:* method:

```
userResponse: theStrings
    "The method for the change selector as model for a FormPane."
    theStrings isNil
        ifTrue: [answerStrings := defaultAnswerStrings.
            Transcript cr;
            show: 'User selected Cancel. Defaults are used.']
        ifFalse: [answerStrings := theStrings]. Transcript cr;
        show: 'The responses are: '; show: (answerStrings
        printString); cr
```

Ready to test our work? (Hopefully, you've been saving the Smalltalk/V image periodically. Now would be a good time to do it again.) Evaluate the following expression in the Transcript or a Workspace:

```
FormApp example
```

The result should look like Figure 11-2. Try resizing the window and noting what happens to the buttons' widths and heights. Try clicking on the "Cancel" button.

Writing Methods to Place Sub-Panes

We are now ready to flesh out our form project by installing the **GraphPane** for the question prompts and the **TextPanes** for the answers. Before we do that, though, we need to pause to think about how we're going to position all of the parts of a **FormPane** with respect to one another. When we do so, the first thing we should notice is that we have a potential problem in the way we've set up the two buttons for the project. Right now, the height and width of the buttons are tied to the size of the pane. If the user resizes the pane, the buttons resize both horizontally and vertically. This can result in a situation where the height of the buttons becomes too small to read their contents.

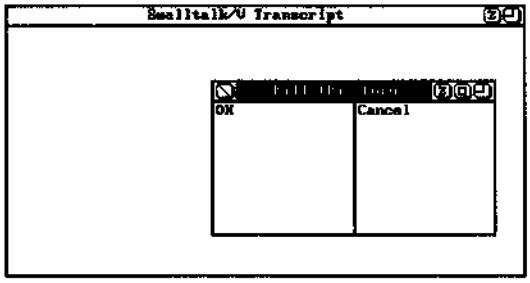

Figure 11 -2. First Pass at Test Application Execution

To solve this problem, we recall from our discussion in Chapter 6 that if we want to specify a static height for a button (as is done, for example, with the "class" and "instance" buttons in the CHB), we need to use a *framingBlock:* rather than a *framingRatio:* as we have done in our first version of the *configureSubpanes* method. The argument to *framingBlock:* is a block of code which itself takes an argument. This argument in turn is the interior frame of the window. Executing the block of code should return a rectangle (in Display coordinates) which is then used to determine the placement of the sub-pane. Modify the *configureSubpanes* method to read as follows:

```
configureSubpanes
    "Format the layout of the window's components." I
    buttonHeight I self initializeFields. buttonHeight
    := ListFont height + 4. self addSubpane: (ListPane
    new model: intermediary; name: #okButton; change:
    #okBu1ton:; framingBlock: [:box I (box origin +
    (0 @ (box height - buttonHeight))) corner: (box origin + ((box
    width // 2) @ box height))]). self addSubpane: (ListPane new
    model: intermediary; name: #cancelButton; change:
    tcancelButton:;
        framingBlock: [:box I (box origin + ((box width // 2) @
            (box height - buttonHeight))) corner: box corner])
```

The *two framingBlock:* messages create rectangles that specify the position of their sub-panes in Display coordinates. The height is calculated to be static *(box height - buttonHeight,* which in turn is set to be four pixels larger than the height of the font used in the **ListPane)** while the width is set up to be proportional to the size of the **FormPane.**

If you now test the **FormApp** application, you will note that the buttons maintain a constant height while adjusting their widths to the size of the

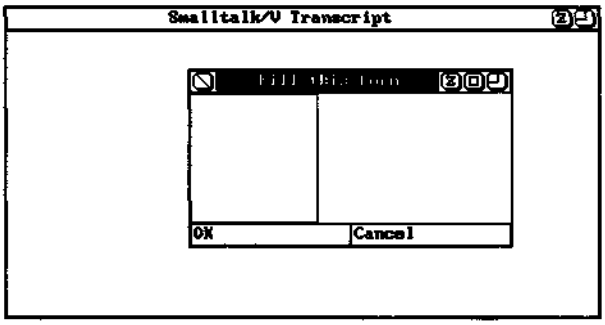

Figure 11-3. Buttons with FormPane at Full Default Height

FormPane, as shown in Figures 11-3 and 11-4.

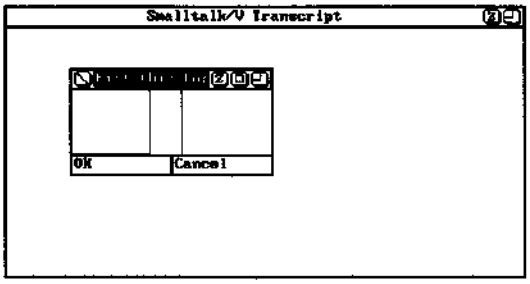

Figure 11 -4. Buttons with FormPane Reduced

Note that by choosing *aframingBlock:* approach for the two buttons in the **FormPane** class, we also forced other panes that occupy portions of the same static space to use *aframingBlock:* approach rather than *aframingRatio:*. You can see evidence of this by examining the code for the *openOn:* method in the class **ClassHierarchyBrowser.** Note that the "Class" and "Instance" buttons use *theframingBlock:* method to establish their size. As a result, the method selector pane also uses this method, while the class and text panes can *useframingRatio:* because they don't share the static space carved out by the buttons. As we will see, we have the same situation in our **FormPane.** We have already decided that we will use an instance of **GraphPane** for the name fields in the left-hand pane of our application and instances of **TextPane** for each data entry field in the right portion of the **FormPane.** We will begin by creating the left-hand **GraphPane** first. We will add some code to the existing *configureSubpanes* method of the class **FormIntermediary** to set up this pane. The added code should be placed following the line that gives a value to the variable *buttonHeight.* In addition, you must add a new local variable to the method; the new variable is *called portion.* Rather than reproduce the entire listing of the new *configureSubpanes* method, we'll just give you the new portion but don't forget to add *portion* to the local variable list in the second line of the method.

```
portion := 2/5.
self addSubpane: (GraphPane new) model: intermediary;
    name: #questionsForm: ; framingBlock: [:box I box
    origin corner:
        (box origin + ((box width * portion) truncated @
        (box height - buttonHeight)))]).
```

We need a method in **FormIntermediary** to support the *name:* selector. Here is the code for that method:

```
questionsForm: aRect
   "Self initializes. Must happen after theTopPane is bound
   which is why we put this code here rather than in an initialize
   method."
    I aForm I
   aForm := Form width: (aRect width)
```

```
        height: (aRect height) .
    aForm display At: aRect origin.
    •"aForm
```

Save the image and try the test application again by executing

```
FormApp example
```

in a Workspace or the Transcript. You should be impressed with your work so far! Even though the application isn't complete yet, it is beginning to take shape and you can see how it is going to behave.

Installing the Text Panes for Each Editable Field

We're ready now to tackle a fairly difficult part of our assignment. We must code *a framing Block:* method that uses variables intelligently because the number of data entry fields that a **FormPane** must accommodate is not known in advance. We want our methods to be as reusable as possible. As you' 11 see, handling this assignment requires us to do some clever programming. The work we want to do will be handled within the *configureSubpanes* method **of FormPane.** But that method is already getting fairly long and we can tell that the new code we're about to write will be (in the words of software professionals) "non-trivial." So we'll add a single line to the *configureSubpanes* method to invoke another method that will actually handle the initialization of the multiple **TextPanes** needed to serve as entry fields. That line looks like this:

```
self conf igureTextPanes: portion buttonHeight: buttonHeight.
```

Place the above line after the conclusion of the definition of the **GraphPane** and before the code that sets up the "OK" button.

Now we need to write the *conf igureTextPanes .'buttonHeight:* method. This will be the trickiest method in our project.

If we stop and think for a moment, there are only two basic ways to write a method that will create and place an arbitrary number of **TextPanes in a FormPane** for us. One way would be to create hard-coded entries for an arbitrarily large number of such frames and then use only as many of these entries as we have data entry fields in the application. This is inefficient and inflexible and hardly reusable. The alternative is to find a way to write a single method that would use an indexing approach (i.e., setting an index variable to 1 for the first pane, 2 for the second, etc.) and multiply an incremental amount by this index value to calculate dynamically the locations of all of the **TextPanes.**

Examine *thcframingBlock:* for the **GraphPane** in the *configureSubpanes* method. Notice that it uses a local variable of the method — *portion* — as

part of its calculations. It might therefore seem logical for us to define a local variable to be used by *aframingBlock:* method for the *configure-TextPanes: buttonHeight:* method and increment it for each **TextPane** we create. But we run into a technical problem if we attempt to approach the project this way. It is impossible to reference the value of such a variable at the time the **TextPane** is created. That is because we are only able to obtain the ultimate (resultant) value of such a variable in Smalltalk/V. As a result, despite the value of the index at the time each **TextPane** is created, all of the **TextPane** *framingBlock:* executions would be bound to the value of that local variable at the time its containing method (i.e., *configureTextPanes:buttonHeight:)* completed its execution, not the value at the time each block is created.

We can get around this problem by using a separate method to create the block rather than creating all of them in the middle of the execution of a single method. This way, the local variable references from the block will refer to the variable bindings at the time the method that created the sub-pane was executed. By executing this new method once for each pane, we can ensure that we have unique values for that variable and thus non-overlapped placement of all **TextPanes.**

With that background, here is the method that will handle the configuration of the **TextPanes.** Notice that it uses a custom approach to framing block construction called *textFramingBlock: buttonHeight .'portion: quantity:.*

```
configureTextPanes: portion buttonHeight: buttonHeight
   "Configure the layout of the TextPanes, and provide the
   collection of them to intermediary. Portion is the amount of
   horizontal space used by the Graph pane, leaving (1 - portion)
   for the TextPanes." I theTextPanes quantity I quantity :=
   intermediary numberOf Entries . theTextPanes := OrderedCol lection
   new: quantity. 1 to: quantity do: [:index I
       theTextPanes add:
        (TextPane new framingBlock: (self textFramingBlock: index
       buttonHeight: buttonHeight
       portion: portion
       quantity: quantity) ;
       yourself) ] .
   theTextPanes do: [:aTextPane I self addSubpane: aTextPane] .
   intermediary textFieldPanes: theTextPanes
```

Now we can write the *textFramingBlock:buttonHeight:portion:quantity:* method, whose code is reproduced below. Notice the use of comments to describe the mathematical implications of each line of the method's code.

```
textFramingBlock: index
buttonHeight: buttonHeight
portion: portion
```

```
quantity: quantity

    ^[:box I (box
    origin +
         ("dx" (((box width) * portion) truncated)
                @ "dy" (((box height - buttonHeight) // quantity) *
          (index -1))
             )
    corner:
         box origin +
              "dx" ((box width)
                  @ "dy" (((box height - buttonHeight) // quantity)  *
              index))
```

The somewhat unusual formatting of this method should also make its logic easier to follow. If you read this alongside a *conventional framingBlock:* method, you can see what this method does. It uses known quantities like *buttonHeight and portion and quantity* to calculate the right placement for the current **TextPane.** The *index* is used as a multiplier to define the top and bottom locations of each **TextPane's** rectangle.

As you can see from the above methods, we must now supply two more supporting methods to **FormIntermediary.** The first is *textFieldPanes:,* which takes an **OrderedCollection** as an argument. The second is *numberOfEntries* which simply returns the number of questions the application is designed to ask. Here are those two methods:

```
textFieldPanes: anIndexedCollection
   textFieldPanes := anIndexedCollection

numberOfEntries
   ^questionStrings size
```

Now when we try our test application again, we sense a bit more success. Resizing it demonstrates the self-configuration characteristics of the subpanes. But we still have some problems to work out. For example, there is an extra space between the bottom **TextPane** and the button due to our truncation operation where we lose a partial pixel for each entry and the accumulated effect is noticeable.

Your first impulse might be simply to switch from truncation to rounding, but this won't fix the problem in all cases. If that process consistently rounds down, the problem will persist. The best solution is probably to have the last **TextPane** use the top of the buttons as its bottom coordinate rather than calculating it from the top of the window with the truncation errors. Here is a revised *textFramingBlock:buttonHeight:portion:quantity:* method that handles this situation:

```
textFramingBlock: index buttonHeight: buttonHeight portion: portion quantity:
quantity
    "Generate the framingBlock for the TextPanes. If the Text Pane is
    the bottom one, indicated by the == test, then be sure it butts
    up against the buttons (i.e. , use the alternate block) . " index.
    == quantity if False: [^[:box|  (box origin +
        ("dX" (((box width) * portion) truncated)
        @
        "dY" ( ( (box height - buttonHeight) // quantity) * (index - 1) )
        ) corner:
        (box origin + "dX" ( (box width)
        @
        "6V" ( ( (box height - buttonHeight) // quantity) * index) )


    if True: [^[:box i (box origin +
         ("dX" (((box width) * portion) truncated)
        @
        "dY" ( ( (box height - buttonHeight) // quantity) * (index - 1) )
        ) corner:
        (box origin + "dX" ( (box width) @
        "dY" (box height - buttonHeight) ) ) ) ] ]
```

Testing our application now demonstrates that the problem is handled.

Modifying Undesirable Behavior in the Superclasses

If you experiment with the test application for a few minutes, you'll notice that we still have some problems to resolve before we can consider ourselves to have created a reusable project. Specifically, you'll find that:

- selecting the "OK" or "Cancel" button results in the system asking if we want to save pane contents
- zooming of the **FormPane** always results in zooming only the first **TextPane**
- default values don't appear in the data entry fields as expected

To address these problems, we're going to make the following changes to the project:

- disable the querying about saving changes
- disable zooming
- disable collapsing
- remove the **FormPane's** close box (to ensure the user exits the form via one of our buttons)
- initialize the **TextPanes** to their default values

Disabling Query on Ending Interaction with FormPane

We need to determine where in the system the question about saving changes originates. Since we know that the *close* method of the **TopDispatcher** class is the source of the action that closes the window, we begin our search there. Use the CHB to examine the source code for this method. Notice that it sends *self a closelt* message. Let's track down *closelt.* We can find it in the class **Dispatcher.** Here is its source code:

```
closelt
    "Close the receiver window and resume the
    Scheduler main processing loop. " self
    topDispatcher pane textModified ifTrue:
    [^self]. self closeWindow. Scheduler resume
```

As you can see, this method checks to see if *textModified* produces a *true* result and, if it does, the method does not close the window. Clearly, then, we need to find out how the *textModified* method works. Consulting the *Method Index* in the Smalltalk/V 286 documentation, you can see that *textModified* is implemented only in the class **TopPane.**

Examining its source code in the CHB, we determine that its purpose is to query the user and return *true* if the user wishes to continue editing,/a/,se if they wish to discard changes. Because we will force the user to use the "OK" or "Cancel" button to terminate editing in our application, we can safely assume that the answer will always *be false* because we will already have processed the panes' contents.

Since **FormPane** is a subclass of **TopPane,** we can simply override the *textModified* method with our own:

```
textModified
    "The saving of modifications is handled by an alternate
    mechanism, so ignore this one. "
    ^false
```

Disabling Zooming, Collapsing, and Closing

The most direct way to eliminate the possibility that the user will zoom, collapse, or close the **FormPane** inappropriately is to remove the icons that enable this behavior. We saw in Chapter 6 that the methods *leftlcons:* and *rightlcons:* determine which icons will be available in a window.

We could simply handle the removal of these icons in our FormApp instance when we create the **FormPane** and set some of its arguments. But since we are going to want to do this for every application that uses **FormPane,** it is better handled in the class **FormPane** directly.

Before we can remove the icons, we have to find out where they are now being set. In the CHB, locate **TopPane,** select its *leftlcons:* method, and

choose "senders" from the pane pop-up menu to find out which **TopPane** method sets the default value. We discover that the *initialize* method **of Top-Pane** handles this assignment. This is fortuitous because we've already overridden that method in our **FormPane** class. Recall that when we did that, we sent the *super initialize* message first so that the return value from the initialization would be consistent. Now we need to send that message last so that our override of the icons isn't in turn overridden by the **TopPane's** *initialize* method. But we still want to be sure that we return the same result as if we had called the superclass' *initialize* method first. We accomplish that by storing the result of that method call in a local variable. Here's the code for the modified *initialize* method for **FormPane:**

```
initialize I
    result I
    result := super initialize,
    intermediary := Formlntermediary new.
    intermediary formPane: self, self
    leftlcons: #() . self rightlcons: #
    (resize) . '""result
```

A quick test of the **FormApp** reveals that our changes have worked.

Putting Default Values into Editing Fields

The final change in our list is to initialize the **TextPanes** so that they display the default answers when we open the pane.

We might start by simply locating a method that will add text to a **TextPane.** As we saw in Chapter 10, the *appendText:* method handles this assignment. Since we're starting with empty panes, appending the default answers to the existing text would work just fine for our purposes.

This turns out, however, not to be an optimal solution. Why? Because we really want to make use of the existing initialization mechanism for **TextPanes,** which set up pane contents by the standard process of querying the model. This is also the method by which a pane restores its original contents; in our case, that means the default responses. So it makes eminent sense for us to use this approach to our own pane content initialization.

To gain a better understanding of how and when a **TextPane** is initialized, examine the *open* method of the class **TopPane** in the CHB. Notice that it sends the *open* message to each of its sub-panes. Since we are interested in **TextPanes,** we examine the *open* method of that class. Note that it performs a *self cancel.*

The logic behind this approach is that the "cancel" menu item for a **TextPane** reverts the pane to its original contents prior to any user editing. Whether it is reverting to this value or setting it initially, the mechanism is identical. The process involves simply setting the pane's contents to the value provided by the model.

Now let's examine the *cancel* method of the class **TextPane.** Notice the line that says in part:

```
string: (model perform: name)
```

If the model provides a selector to use for getting the pane';s initial value, the *cancel* method uses it. The problem with this approach for our project is that we wish to use the same method to initialize all of the **TextPanes.** But the expression above does not let us know which **TextPane** we are addressing. Thus this mechanism makes it impossible to know which *defaultAnswerString* to provide.

As we can see, then, there are two ways for us to provide the initial value for a **TextPane.** We can use the *cancel* method or we can set up a *name* method in the **TextPane's** model. We'll do the latter.

Let's begin by subclassing **TextPane** to create a new class **EntryTextPane** that supports informing its model of its identity. Open the CHB, choose **TextPane** and then "new subclass" from the pane pop-up menu. Edit the new class template so that it looks like this:

```
TextPane subclass: #EntryTextPane
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: "'
```

This class will have only one method, *cancel.* We set it up so that our model can support either a *name* selector method or a *cancel:* method which takes the **TextPane** as an argument. Here is the code:

```
cancel
    "Private - Restore the last saved
    version of the text."
    name == #yourself
    ifTrue: [textHolder string: model]
    ifFalse: [
        (model respondsTo: #cancel:) if
        True: ['"model cancel: self]
        ifFalse:[
            textHolder
            string: (model perform: name with: self)]].
    self
    topCorner: 1@1;
    refreshAll
```

Next, we update the *configureTextPanes:buttonHeight:* method to set up the name for the **Entry TextPanes,** so that it looks like this:

```
configureTextPanes: portion buttonHeight:  buttonHeight
    "Configure the layout of the TextPanes, and provide the collection of
    them to intennediary. Portion is the amount of horizontal space used
    ty the Graph pane, leaving (1 - portion) for the TextPanes."
     I theTextPanes quantity I
    quantity := intermediary numberOfEntries
```

```
theTextPanes := OrderedCollection new: quantity. 1
to: quantity do: [:index I
    theTextPanes add:
     (EntryTextPane new framingBlock: (self textFramingBlock: index
    buttonHeight: buttonHeight
    portion: portion
    quantity: quantity) ;
    model: intermediary; "This line added"
    name: #defaultAnswer: /"This line added"
    yourself)].
theText Panes do: [:aTextPane I self addSubpane: aTextPane] .
intermediary textFieldPanes: theTextPanes
```

Now **FormIntermediary** obviously needs a *default Answer:* method so that it can deal with the updating process established in the above method. Here is the code for this new method:

```
defaultAnswer: entryTextPane
    "Find the index of the entry, and extract the string of the same
    index from the defaults."
    ^defaultAnswerStrings at: (textFieldPanes indexOf: entryTextPane)
```

This method uses the index that is tracking which entry field is being worked with to retrieve the default answer from the *defaultAnswerStrings,* which was initialized to be an **Association** so that we could retrieve its contents by index value.

> Adding Menu Capabilities

Another modification we'd like to support to our new **FormPane** project is to provide a custom popup menu providing some of the key functionality that exists in **TextPanes** and **TextEditors.** We're going to make it possible for the user to restore the pane's contents, to perform the standard editing operations (cut, copy, and paste), and to zoom any individual **TextPane** in conformance with the project design described earlier.

As we have seen in earlier projects, we can associate a popup menu with a pane by setting its *paneMenuSelector* to a selector of a method supported by the model. We'll add that capability to our *configureTextPanes:button-Height:* method, just before the *yourself* line that signals the end of the creation of the **EntryTextPane** instances. Insert this line in that place:

```
menu: #entryFieldMenu;
```

Before we create our menu, let's take a look at what is already available to us in the way of default menus and behaviors. We look through the class **TextPane** in the CHB to find the method that invokes a menu. The

performMenu method has possibilities, so we examine its code. Reading the comment, we can see that we need to deal with the dispatcher class. As we saw in Chapter 10, the dispatcher class for a **TextPane is a TextEditor.** We browse through this class and look at its *menu* metamethod. It returns the contents of *StandardEditMenu.* You may have noticed already that this selector begins with a capital letter, meaning that it is most likely a global variable. Select it and then choose "showlt" from the pane popup menu. We find out it is not globally bound.

So we check the class definition and find that it is a class variable. Now that we know what it is, we need to find out where it is initialized so we can see what's in it by default. There are only a few metamethods to look at and *initialize* turns out to be the right choice. By examining its code, we can see that it defines a single menu, the last choice of which is "next menu" and whose selector is *misc.* From our experience working with the system, we know that this menu selection brings up a second menu and we want to examine what's in it as well, so we look at the *misc* method.

Now we know what the selector names are that correspond to the various menu choices that we want to include in our **EntryTextPanes.** Here is the code for our method to be placed in the **FormIntermediary** class:

```
entryFieldMenu
    ^Menu
        labels: 'restore\copy\cut\paste\zoom' withers
        lines: #(1-4)
        selectors : # (cancel copySelection cutSelection
        pasteSelection zoom)
```

Run the test application and bring up the entry field menu to confirm that we have indeed accomplished our objective. Selecting "zoom" now enables us to zoom any selected **EntryTextPane** rather than just the topmost pane as happened before when we chose the zoom icon in the window.

Formatting the Name Fields

At this point, our application is not yet displaying the prompt labels (name fields) in the **GraphPane.** Since this is primarily cosmetic (i.e., it has limited interactive functionality), we have saved this task to near the end of constructing our project. The time has now come to handle this responsibility, however.

Look at the *name* selector in the **GraphPane** definition of our *configureSubpanes* method in the class **FormIntermediary.** Notice that it is *questionsForm:.* This method now does nothing but set up the blank pane area. Let's rewrite it so it looks like this:

```
quest ionsForm: aRect
   "As the model of a GraphPane, this method supports the name
   selector. By returning nil, it informs the GraphPane to use this
   method when the window is -.refrained. " I aForm characterscanner
   spacing theString I aForm := Form width: (aRect width)
       height: (aRect height) .
   characterscanner : = Characterscanner new
       initialize: (Display boundingBox) f ont : SysFont .
   aForm displayAt : aRect origin.
   spacing := (aRect height // self numberOf Entries) .
   1 to: (self numberOf Entries) do: [: index I
       theString := (questionStrings at: index) .
   characterscanner display : theString
       at: (aRect origin +
            (4 @ ( (spacing \\ 2) +
            (spacing * (index - 1) )))) ].
```

Let's examine this code in some detail. It accomplishes two basic tasks:

1. renders a white background
2. draws text in the **GraphPane**

First, we create a new **Form** the same size as the pane:

```
aForm := Form width: (aRect width)
    height: (aRect height) .
```

Next we create a **Character Scanner** whose job (as you'll recall from Chapter 10) is converting ASCII codes to bitmaps in the system font. Because we will specifically position each string to be drawn, we can let the **CharacterScanner** be set up to use the whole display:

```
characterscanner : = CharacterScanner new
    initialize: (Display boundingBox) font: SysFont.
```

Now we display our blank background (blank because all **Forms** are blank when they are created):

```
aForm displayAt : aRect origin.
```

Now for the real work. We iterate through the entries, asking the *characterScanner* to display each string with consistent horizontal spacing (4 pixels from the left) and with a calculated vertical spacing:

```
1 to: (self numberOf Entries) do: [: index I
   theString := (questionStrings at: index) .
   characterScanner display: theString at:
   (aRect origin +
          (4 @ ( (spacing \\ 2) +
          (spacing * (index - 1) )))) ].
```

Any GraphPane's *name* method can return one of two values. If it returns a **Form,** that form will be used by the **GraphPane** when it is resized. If the *name* method returns *nil,* then it will ask the model to redraw the **GraphPane** by sending the *name* selector as a message to the model. This is how we arrange to have the area redrawn when the window is resized. In our case, *nil* is the right answer, so the last line of the method returns that value.

Creating a More Complex Test Application

Our new project has a great deal of functionality. Let's design a somewhat more complicated test application to demonstrate its powers more appropriately. Here's the code for a new metamethod called *example2:*

```
exairple2
    "FormApp exanple2."
    I aFormApp I
    aFormApp := self new.
    aFormApp openOn: # (' Name: ' 'Address:' 'City:[1] 'State:' 'Zip:'
         'Phone #: ')
        default Answers: # (' Michael Angelo' '123 Main Street' ' Any town'
        'CA'    '12345' '900-976-4257')
        label: 'Fill this form'.
    aFormApp open
```

Try out this new example with the following line in a Workspace or the Transcript. The results should look like Figure 11-5.

```
FormApp example2
```

Things are coming along quite nicely. In fact, we have only one remaining problem and that is that the **GraphPane's** contents still scroll. We need to disable that automatic feature of such panes.

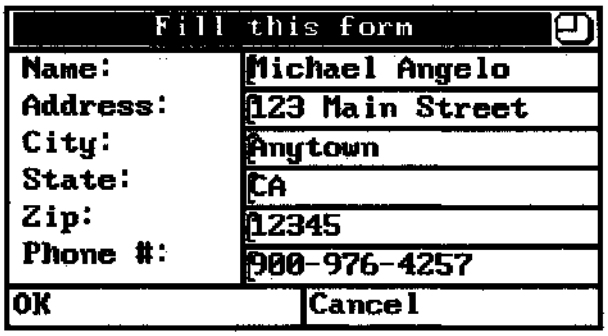

Figure 11-5. A More Complex Sample Application

Disabling Scrolling in the GraphPane

Before we can disable this scrolling behavior, we have to know how it gets invoked in the first place. Clearly, a pane doesn' t scroll without an interpretation of a user action of some sort. User actions are the province of **Dispatchers,** so the first thing we need to know is what kind **of Dispatcher** is associated with a **GraphPane.** Examine the *defaultDispatcherClass* method of a **GraphPane** in the CHB. It indicates that a **GraphDispatcher** plays this role.

Now we examine the **GraphDispatcher** class to see how it invokes scrolling. We're in luck; it has only one method, *processFunctionKey:*. Unfortunately, as we look at this method, nothing seems to invoke any scrolling. So we examine the inherited version *of processFunctionKey:* in the class **ScrollDispatcher.**

As we examine this code, we quickly conclude that all subclasses of **ScrollDispatcher** inherit scrolling. We might therefore think that a good way to eliminate scrolling would be to create a new subclass of **GraphDispatcher** called, for example, **NoScrollDispatcher.** A careful examination of this alternative reveals that it has some problems. To see them, select the *processFunctionKey:* method for any of the **Dispatcher** classes and select "implementors" from the pane popup menu. This opens a browser on all implementors of this method and enables us to see how this method in **GraphDispatcher** uses the one from **ScrollDispatcher,** which in turn uses the method from **Dispatcher.**

We want most of this inherited behavior. All we really want to rid ourselves of is the scrolling behavior. The best way to do this is to create a **NoScrollDispatcher** class that is a subclass of **Dispatcher** rather than of **ScrollDispatcher.** That way, the new class would not inherit the scrolling capabilities to begin with and we wouldn't waste our time figuring out how to override them.

In the CHB, select the class **Dispatcher** and create a new subclass, editing the default template to appear as follows:

```
Dispatcher subclass: #NoScrollGraphDispatcher
 instanceVariableNam.es: ' ' classVariableNames: ' '
 poolDictionaries: ' FunctionKeys CharacterConstants '
```

Copy *the processFunctionKey:* method **of GraphDispatcher** into this new class. The code should look like this:

```
processFunctionKey: aCharacter
```

```
"Private - Perform the requested function
from the keyboard or mouse. "
(SelectFunction == aCharacter or:
    [EndSelectFunction == aCharacter])
if True: [^pane selectAtCursor] . SetLoc
== aCharacter ifTrue: [
    Terminal mouseSelectOn
        if True: [^pane selectAtCursor] ] .
super processFunctionKey: aCharacter
```

Now we just need to subclass **GraphPane** so it will use our **NoScrollDispatcher** as its dispatcher. We'll make this class a **NoScrollGraphPane.** Open the CHB, select **GraphPane** and create a new subclass, editing the resulting template so it looks like this:

```
GraphPane subclass: #NoScrollGraphPane
 instanceVariableNam.es: ''
 classVariableNames: ''
 poolDictionaries: '
```

This new class will have only one method since its only purpose is to ensure the proper **Dispatcher** is used:

```
defaultDispatcherClass
    "Answer the default dispatcher
    of a NoScrollGraphPane."
    ^NoScroliGraphDi spat cher
```

Finally, edit the *configureSubpanes* method in the class **FormPane** so that the one reference in it to **GraphPane** is changed to refer to a **NoScrollGraphPane.**

Save the image and test to your heart's content. Everything works as originally envisioned.

An Alternative Approach

We could have dealt with the scrolling problem of the previous section in a completely different way. We could have defined a new class between **GraphPane and SubPane** that handled all but the scrolling methods, which we would place in **GraphPane** instead. That would have required a bit more detail and knowledge and, for the purposes of this book, wouldn't have added much to the discussion. Still, it is an alternative that you may want to examine for your own interest.

There are almost always several ways to accomplish a particular objective in Smalltalk.

Complete Source Code

Following is the complete source code listing of the project in this chapter.

```
TextPane subclass: #EntryTextPane
  instanceVariableNames: ' '
  classVariableNames: ''
  poolDictionaries: ' ' !

!EntryTextPane class methods ! !


!EntryTextPane methods !

cancel
    "Private - Restore the last saved
    version of the text." name == #yourself
        ifTrue: [textHolder string: model]
        ifFalse: [
            (model respondsTo: #cancel:)
                ifTrue: ["model cancel: self]
                ifFalse:[
                    textHolder
                        string: (model perform: name with: self)]].
    self
        topCorner: 1@1;
        refreshAll! !


Object subclass: #FormIntermediary
  instanceVariableNames:
    l formPane questionStrings defaultAnswerstrings textFieldPanes questionsForm
questionsScanner '
  classVariableNames: ' '
  poolDictionaries: '' !

IFormlntermediary class methods ! !

IFormlntermediary methods !

cancelButton
     "Label for the Cancel button."
    A#(Cancel)!

cancelButton: whoCares
    "Because the user has canceled, we provide nil as the result. The
    model to formPane should know that nil indicates that no values were
    changed, so use what it has as the default if desired." formPane
    takeResult: nil!

defaultAnswer
        "comment" I I Adefault AnswerStrings at: (textFieldPanes
    indexOf: self)!

defaultAnswer: entryTextPane
    "Find the index of the entry, and extract the string of the same
     index from the defaults." AdefaultAnswerStrings at: (textFieldPanes
    indexOf: entryTextPane)!

defaultAnswerstrings: indexedStrings
```

```
    defaultAnswerStrings := indexedStrings !

entryFieldMenu
    ^Menu
        labels: ' restoreXcopy \cut\paste\zoom' withers
        lines: #(14)
       selectors: # (cancel copySelection cutSelection
                    pasteSelection zoom) !

formPane: aFormPane
    formPane := aFormPane !

numberOf Entries
    ^questionStrings size!

okButton
     "Label for the OK button. "


okButton: whoCares
     "Inform formPane of the values in the question/ field subpanes. As a
     model to a ListPane, it must take an argument, of the list index. We
     ignore it because the list only has a single item. " I result I
    result := OrderedCollection new.
    textFieldPanes notNil
        ifTrue: [textFieldPanes do: [:aSubpane I
                  result add: aSubpane contents] ] .
    formPane takeResult: result!

questionsForm: aRect
    "As the model of a GraphPane, this is this method supports the name
    selector.  By returning nil, it informs the GraphPane to use this
    method when the window is refrained." I aForm characterScanner spacing
    theString I aForm := Form width: (aRect width)
                  height: (aRect height).
    characterScanner := CharacterScanner new
            initialize: (Display boundingBox) font: SysFont .
    aForm display At: aRect origin.
    spacing := (aRect height // self numberOf Entries) .
    1 to: (self numberOf Entries) do: [: index I
            theString := (questionStrings at: index).
            characterScanner display: theString at:
            (aRect origin +
                                    (4 @ ( (spacing \\ 2) +
                                          (spacing * (index - 1))))))] .


questionStrings: indexedStrings
    questionStrings := indexedStrings!

textFieldPanes : anIndexedCollection "Set the
    instance variable." textFieldPanes :=
    anIndexedCollection! !


Dispatcher subclass: #NoScrollGraphDispatcher
  instanceVariableNames: ' ' classVariableNames:
  ' ' poolDictionaries:
     'FunctionKeys CharacterConstants ' !

!NoScrollGraphDispatcher class methods ! !
```

```
!NoscrollGraphDispatcher methods !

processFunctionKey: aCharacter
        "Private - Perform the requested function
    from the keyboard or mouse."
    (SelectFunction == aCharacter or:
    [EndSelectFunction == aCharacter])
    ifTrue: [^pane selectAtCursor].
    SetLoc == aCharacter ifTrue: [
            Terminal mouseSelectOn
                ifTrue: [^pane selectAtCursor]].
    super processFunctionKey: aCharacter! !


GraphPane subclass: #NoScrollGraphPane
  instancevariableNames: ' '
  classVariableNames: ''
  poolDictionaries: ' ' !

!NoScrollGraphPane class methods ! !


!NoScrollGraphPane methods !

defaultDispatcherClass
        "Answer the default dispatcher
         of a NoScrollGraphPane."
    ^NoScrollGraphDispatcher! !


TopPane subclass: #FormPane
  instancevariableNames:
     'intermediary name changeSelector
  classVariableNames: ' '
  poolDictionaries: ' ' !

!FormPane class methods ! !


!FormPane methods !

change: aSelector
    "Used as a message to the model to inform it of a change in the
    FormPane. Must take an argument of the FormPane values."
    changeSelector := aSelector!

configureSubpanes
    "Format the layout of the window's components."
    I buttonHeight portion I self initializeFields.
    buttonHeight := ListFont height + 4. portion :=
    2/5.
    self addSubpane: (NoScrollGraphPane new model: intermediary;
            name: #questionsForm:; framingBlock: [:box I box origin
            corner:
                            (box origin + ((box width * portion) truncated
                                    (box height - buttonHeight)))] ).
    self configureTextPanes: portion buttonHeight: buttonHeight.
    self addSubpane: (ListPane new model: intermediary; name:
    ttokButton; change: ftokButton:;
```

```
            f ramingBlock: [:box I (box origin +
             (0 @ (box height - buttonHeight) ) ) corner: (box origin + ( (box
             width // 2) @ box height) ) ] ).
    self addSubpane : (List Pane new model: intermediary;
            name: ttcancelButton; change: ttcancelButton:
            ;
            framingBlock: [:box I (box origin + ( (box width // 2) @
                              (box height - buttonHeight))) corner:
                             box corner]


conf igureTextPanes : portion buttonHeight : buttonHeight
    "Configure the layout of the TextPanes, and provide the collection
    of them to intermediary.  Portion is the amount of horizontal space
    used by the Graph pane, leaving (1 - portion) for the TextPanes." I
    theText Panes quantity 1
    quantity := intermediary numberOf Entries .
    theTextPanes := OrderedCollection new: quantity. 1
    to: quantity do: [: index I theText Pane s add :
        (Entry Text Pane new framingBlock: (self textFramingBlock: index
                        buttonHeight: buttonHeight portion: portion
                        quantity: quantity) ; model: intermediary; name:
                        #defaultAnswer : ; menu: ttentryFieldMenu;
                        yourself) ] .
    theTextPanes do: [:aTextPane I self addSubpane: aTextPane] .
    intermediary textFieldPanes : theTextPanes!

initialize
    I result I
    result := super initialize.
    intermediary := Formlntermediary new.
    intermediary formPane: self. self
    leftlcons: #() . self rightlcons: #
    (resize). '"result !

initializeFields
    "Queries the model and informs the Formlntermediary."
    I anAssociation I
    anAssociation := model perform: name. intermediary
    questionStrings: anAssociation key. intermediary def
    aultAnswerStrings : anAssociation value!

name: aSelector
    "Selector used as the name of the FormPane, and to get the
    initial information displayed in the pane." name :=
    aSelector!

takeResult: result
    "Inform the model of the change and result."
    model perform: changeSelector with: result.
    self dispatcher close!

textFramingBlock: index buttonHeight : buttonHeight portion: portion quantity:
quantity
    "Generate the framingBlock for the TextPanes. If the Text Pane is the
    bottom one, indicated by the == test, then be sure it butts up against the
    buttons (use the alternate block) ."
```

```
    index == quantity
      ifFalse: [^[:box I (box origin +
                 ("dX" (((box width) * portion) truncated)
                     @
                 "dY"  (((box height - buttonHeight)  // quantity)  *  (index - 1))
                 )  corner: (box origin +  "dX"   ((box width)
                      @ "dy"   (((box height - buttonHeight)  // quantity)   *
                index))
          )) ]] ifTrue:  [^[:box   i   (box
      origin +
                 ("dX"   (((box width)   * portion)   truncated)
                      @
                 "dY" (((box height - buttonHeight) // quantity) * (index - 1))
                 ) corner: (box origin + "dX" ((box width) @
                     "dY" (box height - buttonHeight))))]]!

textModified
    "The saving of modifications are handled by an alternate
    mechanism, so ignore this one." ^false! 1


Object subclass: #FormApp
  instanceVariableNames:
    'questionStrings defaultAnswerstrings answerStrings '
  classVariableNames: '' poolDictionaries: '' !

!FormApp class methods 1

example
    "FormApp example." I aFormApp I aFormApp := self new.
    aFormApp openOn: #('Name: ' 'Address: ' 'Phone #: ')
    defaultAnswers: #('Fred' '123 Main Street' '900-976-4257') label:
    'Fill this form'. aFormApp open!

example2
    "FormApp example2." I aFormApp I aFormApp := self new. aFormApp openOn:
    #('Name:' 'Address:' 'City:' 'State:' 'Zip:' 'Phone #: ')
              defaultAnswers: #('Michael Angelo' '123 Main Street' 'Anytown' 'CA'
'12345' '900-976-4257')
              label: 'Fill this form'.
    aFormApp open! !


!FormApp methods !

formlnfo
    "Returns an association of the questions and answers for the FormPane."
    Association key: questionStrings value: defaultAnswerstrings!

openOn:  questions defaultAnswers:  answers label:  aLabel
     "Open a FormPane with the specified questions,   answers,   and label.
     I aFormPane I
    questionStrings   := questions.
    defaultAnswerstrings  := answers.
```

```
    aFormPane := FormPane new
        model: self;
        label:  aLabel;
        name:  ftformInfo;
        change:  #userResponse:;
        yourself.
    aFormPane configureSubpanes. aFormPane
    dispatcher open scheduleWindow!

userResponse: theStrings
    "The method for the change selector as model for a FormPane."
    theStrings isNil
        ifTrue: [answerStrings := defaultAnswerStrings.
                 Transcript cr;
                show: 'User selected Cancel.  Defaults are used.'1]
    ifFalse: [answerStrings := theStrings]. Transcript cr; show: 'The
    responses are: '; show: (answerStrings printString); crl !
```

Index