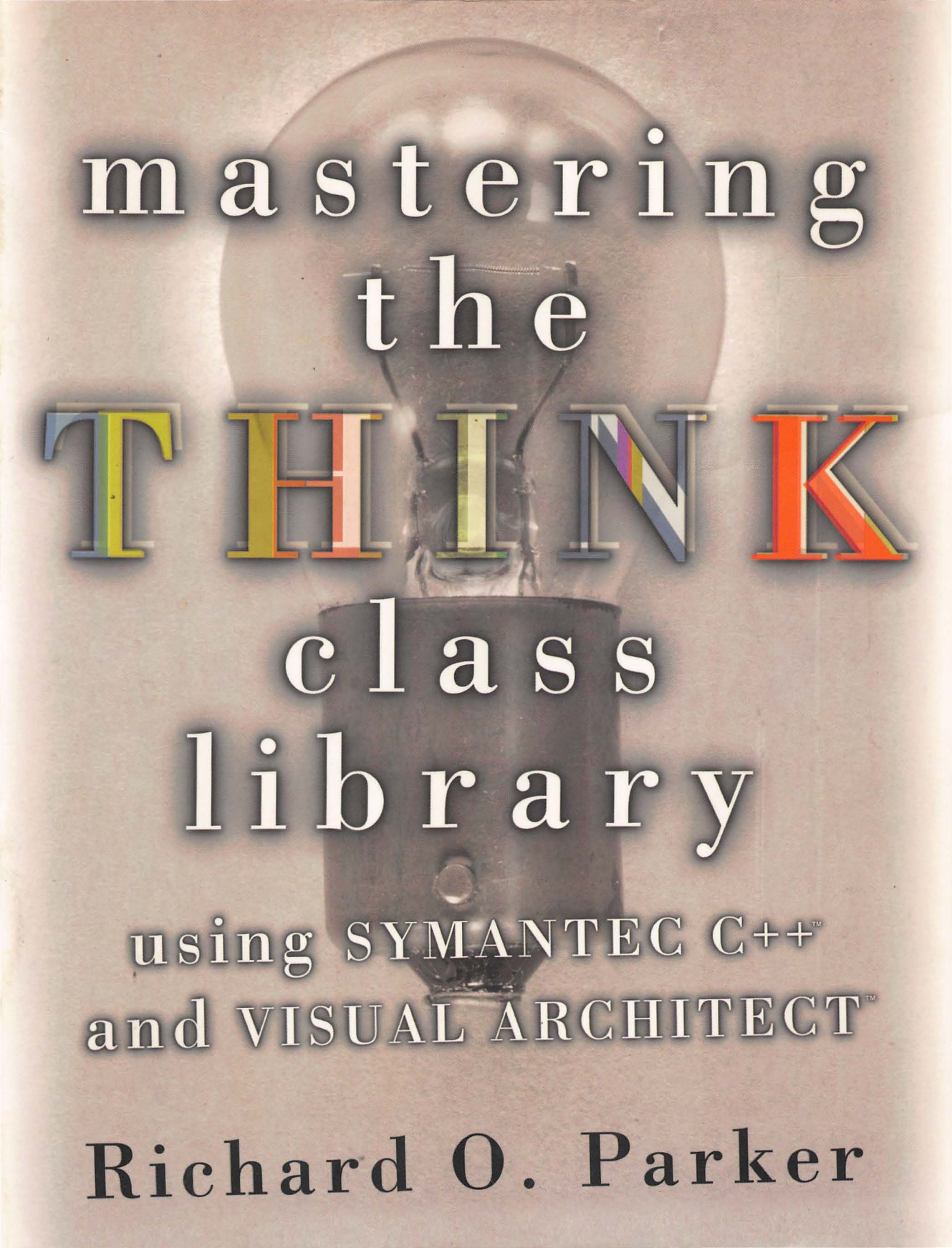

mastering the THINK class library

using SYMANTEC C++™ and VISUAL ARCHITECT™

Richard O. Parker

Mastering the THINK Class Library

Using Symantec C++™ and Visual Architect™

Richard O. Parker

Addison-Wesley Publishing Company

Reading, Massachusetts • Menlo Park, California • New York

Don Mills, Ontario • Wokingham, England • Amsterdam

Bonn • Sydney • Singapore • Tokyo • Madrid • San Juan

Paris • Seoul • Milan • Mexico City • Taipei

Library of Congress Cataloging-in-Publication Data

Parker, Richard O.
Mastering the THINK Class Library : Using Symantec C++ and Visual Architect / Richard O. Parker.
p. cm.
Includes index.
ISBN 0-201-48356-4 (alk. paper)
1. Symantec C++ for the Macintosh. 2. Macintosh (Computer)--Programming. I. Title.
QA76.73.C153P39 1995
005.26'2—dc20 95–218
CIP

Sponsoring Editor: Martha Steffen
Project Manager: Sarah Weaver
Production Coordinator: Deborah McKenna
Cover design: David High
Set in 11 point Adobe Garamond by Pure Imaging

1 2 3 4 5 6 7 8 9-MA-9998979695
First printing, July 1995

Addison-Wesley books are available for bulk purchases by corporations, institutions, and other organizations. For more information please contact the Corporate, Government, and Special Sales Department at (800) 238-9682.

This book is dedicated to my extended family, and especially to my mother, my son Rick, my daughter Kathryn, her husband Jerry, and my grandchildren Shannon, Aaron, and Emily.

My sister Marianne, her husband Terry, her son Dafydd, and their son Chris are always close to my heart.

My family is more important to me than any of them imagines. May they always give more than they get and love one another—that's what makes life worthwhile.

Contents

Chapter 5

Chapter 8

Chapter 9

Preface

This book is an exploration of the internal workings of the THINK Class Library (TCL). The book uses the Visual Architect (VA) and Symantec C++ compiler to aid the exploration effort, allowing us to show how various user interface elements are constructed and the code that is generated to operate them.

It is a rare individual who appreciates the complexity of writing an application for present day Macintosh users. There are many considerations not immediately apparent that the TCL handles, automatically, behind the scenes. In writing this book, I show that using the Visual Architect and the TCL allows you to safely ignore many of these considerations and concentrate on the features of your application. I also point out the things to which you must pay attention during the application development process.

The book covers most of the entire scope of the TCL; however, it is not a reference manual, per se. It presents the library in terms of its major components (a functional organization) and shows illustrations, VA tutorials, and sample code fragments. The plan is to provide insight into how the TCL works internally, so that when you make use of its features, you will be better equipped to understand and use them to their fullest extent. In that regard, the book *can* be treated as a reference by topic, rather than by object class. If you want to learn how to implement a particular feature or how the TCL handles a particular situation, then this book is for you. If you want only to look up the calling sequences for member functions of a particular class, then you should refer to the class descriptions in the product documentation or in THINK Reference.

The book includes many VA tutorials that illustrate how to create visual elements of your application's user interface and how then to modify the generated code to fully implement your application's intentions. Instead of showing complete examples, in most cases I have described as much detail as possible.

In addition to the VA tutorials, the book covers a broad range of topics, including the following:

- An introduction to the structure, visual hierarchy, and chain of command in the TCL, and an introduction to the Visual Architect (VA).
- Detailed descriptions of how the application framework is initialized when your application starts running are provided. The entire application construction sequence is described.
- Descriptions of how documents are created and how support for both single and multiple document (file) types are provided in the TCL. A detailed code example of how to read and write simple text files is provided.
- The creation of various types of views with the Visual Architect is described, along with descriptions of the generated code for a business account main view, a splash screen view, a floating palette view, and a tear-off menu view.
- Both modal and modeless dialogs are described. The code for a complex text style modal dialog, a category editor modeless dialog, and a dynamic modeless dialog are examined in detail.
- A variety of standard controls and their behaviors are described. The methodology of handling push buttons, radio buttons, checkboxes, pop-up menus, one and two-dimension tables, and text fields is described. Semantic events are defined and described, as they pertain to the various TCL controls.
- The entire event loop and the actions for each type of event handled by the TCL, are described. The descriptions are interspersed with diagrams that illustrate the dynamic relationships of the various objects that participate in handling events.
- Template classes and their use in the TCL to define various types of collections are covered. The use of "iterators" to loop through the elements of a collection is described.
- The topic of Object I/O is covered both with respect to how the TCL uses this facility to create instances of visual objects when a window or dialog is opened, and how you can make use of the Object I/O facilities to read and write your application's data. A complete Object I/O example is presented. The

use of the SimpleSaver facility for performing simplified input and output is also described.

- The TCL v2.0 is highly factored, with support for sending and receiving Apple events both to and from its own classes, but also to and from classes that you may create. The entire topic of Apple event handling is described in great detail. The support functions, callback routines, and handling of object specifiers are covered in detail.

- Chores and tasks, as well as how the Undo/Redo facilities of the TCL are described. Examples of periodic chores and tasks for undoing a text style and mouse tracking are provided.

- Drawing and printing are covered, with respect to main document windows and secondary windows. The entire process of printing a window's contents is described.

Many of the descriptions in the book are punctuated with VA tutorials or dynamic class construction diagrams. The diagrams provide a view of the dynamic state of the application, when it is performing a particular function, that is not immediately apparent when looking at a class hierarchy diagram.

Some of the more detailed descriptions provide a verbal trace through the TCL code that handles a particular operation. These descriptions can provide insight into the ramifications of performing even some of the most trivial operations. The descriptions are intended both to illuminate the complexity of the procedures and also to make you glad that the TCL is handling most of those details for you.

Some of you may wish after reading this book that some of the examples had been expanded more fully. It was with a great deal of soul searching that I decided that covering more of the breadth of the TCL's features, instead of focusing attention on only a few features, was the better and more helpful approach. This is not to say that the examples are trivial. Far from it. Many of the examples include detailed VA tutorials and full source code for implementing the feature or features being described.

Notation Used in This Book

In order to make this book easier to read and to stress certain elements in the text, we have chosen to use a number of different type styles and type faces. These are as follows:

- Class and member function names are written in the body text typeface (Adobe Garamond), in the same form that they are shown in the code. So, for example, you will see us write about the MakeNewWindow function of the x_CMain class.
- References to variable names in the text are in the Courier typeface. A variable name might be written as `elementSize`. Values of variables are also in Courier, so you will see us represent a particular variable's value as `0` or `FALSE` in the text.
- Acronyms and abbreviations that are three letters or longer are shown in small capitals to improve readability (for example, TCL is TCL and ASCII is ASCII in this book).
- File names are shown in boldface body type. You will see these as **CMain.h** or **CAppleEventObject.cp**. Menu and command names are also shown in boldface type, so you will see that the **Print** command in the **File** menu is shown this way.
- All code examples are displayed in the Courier typeface.

Acknowledgments

The author would like to thank several people who made this book possible. First and foremost is Bob Foster of the Object Factory, without whose tireless efforts the current versions of the TCL and VA would not exist, and from whom the author received many tips and explanations of the inner workings of the library.

Many thanks go to David Neal, Dave Alcott, Kevin Irlen, and Tom Emerson of Symantec Corp. I could not have written the book without their help and advice.

Of course, I cannot forget my editor, Martha Steffen, at Addison-Wesley, who let loose of the reins and let me follow my own path in writing this book; and the help of my agent, Carole McClendon, who was paramount in getting this book project off the ground. Thanks to you both.

Chapter 1

Introduction to the TCL

Let's clear up a couple of confusing terms before getting into the middle of the TCL and how it works. You will often hear the terms "class library" and "application framework" used in the same sentence and perhaps are wondering whether there is a difference between the two, or if they are synonymous. Let me try to clear this up, if I can.

A class library is a collection of classes that provide a specified set of features. A library of matrix arithmetic classes would qualify as a genuine class library, as would a set of classes that implement an interface to Apple's MIDI manager or the communications toolbox. Each of these is merely a collection of classes, with their associated member functions and data, that enable a C++ program to acquire new functionality by simply including the classes in the program.

An application framework, on the other hand, is a complete application skeleton—in the respect that it contains all of the fundamental features that enable a Macintosh application to run. By writing just a few lines of code and including the essential classes and member functions from the TCL, you can create a complete application that is missing only the custom code to implement the unique features of your final product. Such an application framework contains provisions for establishing a menu bar, a set of default menus, and perhaps even a default window. The application also includes the ability to handle events and process menu commands; however, unless a particular command is handled by the framework, the framework may simply ignore the command.

I hope that you can see the distinction between just a class library and an entire application framework. The TCL is a complete application framework, embodying the fundamental features that are common to all Macintosh applications. By supplementing the TCL

with your own custom classes and member functions, you can build a custom Macintosh application.

So with respect to the two foregoing definitions, the THINK class library should rightfully be renamed the THINK Application Framework. However, because the former is forever frozen in the pages of history, I will refrain from belaboring the distinction. Instead, I will use the term TCL, even though I am using an application framework.

Looking More Deeply into the TCL

Beneath the label, the TCL is a rich collection of classes and member functions. The TCL is so richly endowed that many parts of it work automatically, completely behind the scenes, to provide important functions that you would otherwise have to include in your own code. In addition to the broad range of functions included in the library—I will cover these in much greater detail later—it is important to describe how the operation of the application framework differs from that of a typical procedural application.

You might think that an application framework is no different from a library of very capable subroutines, but there is a great difference between the two. In the first place, an application framework is upside-down in comparison to a subroutine library. In the case of the subroutine library, you write your program to perform most of the tasks that are unique to your application and then call subroutines to perform special functions that the library routines implement. In contrast, an application framework, once given control, performs all of the tasks that it possibly can and then calls *your program* to handle the tasks that it is unable to perform.

So once you have given control to the TCL, it runs by itself, accepting and handling the user's actions (events) to the extent of its abilities and then calls upon your application, as needed, to perform the remaining tasks. Examples of tasks that the TCL can handle without assistance include zooming and closing windows, dragging windows from one area of the screen to another, implementing pull-down menus, allowing the user to choose a menu command, redrawing the contents of many types of windows,

handling pop-up menus, checking checkboxes, changing the active selection for radio controls, and many other tasks. Examples of tasks that your program must implement include what to do when the user chooses a particular menu command, updating (redrawing) the contents of graphic or other special windows and their contents, deciding what to do when a particular checkbox or radio button is clicked, and other tasks that are unique to the functionality of your application.

Basic Structure of the TCL

The TCL is organized as a tree with multiple roots. The designers of the TCL labored long and hard to personalize the framework by defining names for the classes that were indicative of their functions. For example, many of the foundation components of the framework are rooted in a class named CCollaborator that contains functions to accept messages from objects in other parts of the tree and passes them on to other objects in the tree. In this way, all of the classes in this subtree are capable of talking with one another via the collaboration mechanism. The tree rooted in the CCollaborator class is the largest in the framework, encompassing over eighty separate classes.

In addition to the tree of CCollaborator objects, there are several other trees in the TCL, some of which consist of only a single class. Also, because the TCL supports multiple inheritance, various trees are interconnected with inheritance links to other trees. Trees with multiple classes in their hierarchy include CEnvirons, which establishes various types of drawing environments, including subclasses that manage text and color text environments for individual window panes; the CStream-rooted tree, which handles various types of stream I/O; the CFile-rooted tree, which handles more conventional Macintosh file I/O; the CChore-rooted tree, which handles periodic tasks; the CTask-rooted tree, which handles text and table-related tasks that can be undone; and many other trees and single root classes.

Some of the classes that stand alone in the framework are of great importance to every Macintosh application. The CDecorator class is responsible for arranging the sizes and positions of all of the application's windows. The CSwitchboard class is the first to access most events before dispatching them to other classes to be han-

dled. The CBartender class is responsible for handling the menu bar and events that result in the choice of a menu command. The CError class posts standard Macintosh Alerts for error conditions recognized by the TCL and also by your application. And the CException class contains more comprehensive exception handling functions. For even more sophisticated error handling, the TCL implements a large subset of the typed exception handling proposed by the draft C++ standard.

Many of the individual classes in the TCL inherit functionality from a class called CAppleEventObject. This class provides the basis for responding to and sending Apple Events in the course of an application's execution. Much of the support for handling Apple Events is built into the various framework classes. You can also add your own classes to handle (or record) Apple Events as they occur or produce fully factored applications by sending Apple Events to your own application. The TCL is already set up to provide these features, automatically, when you set the value of a single Boolean variable (factoring) to TRUE. When this is done, the TCL will send Apple Events to itself to handle many standard operations, such as opening files, printing files, and closing, dragging, zooming, and resizing windows. Using the features of the TCL as a basis, you can create fully factored applications that can, in turn, be driven by AppleScript, Userland Frontier, or other OSA-compatible scripting languages.

Defining the Visual Hierarchy

In addition to the various class trees and inheritance links that are built into the framework, there are two additional groupings of classes. These are called the Visual Hierarchy and the Chain of Command.

The Visual Hierarchy is easy to explain. It consists of all of the windows and each of the individual panes that make up each window in the completed application. All of the windows are managed by the CDesktop object and each CWindow object may contain one or more CPane objects. The visual hierarchy of an application that has three open windows might appear like what is shown in Figure 1-1. Note in the figure that the CDesktop object is at the top of the visual hierarchy. That object keeps track of all open windows and is the dispatcher for mouse events that affect

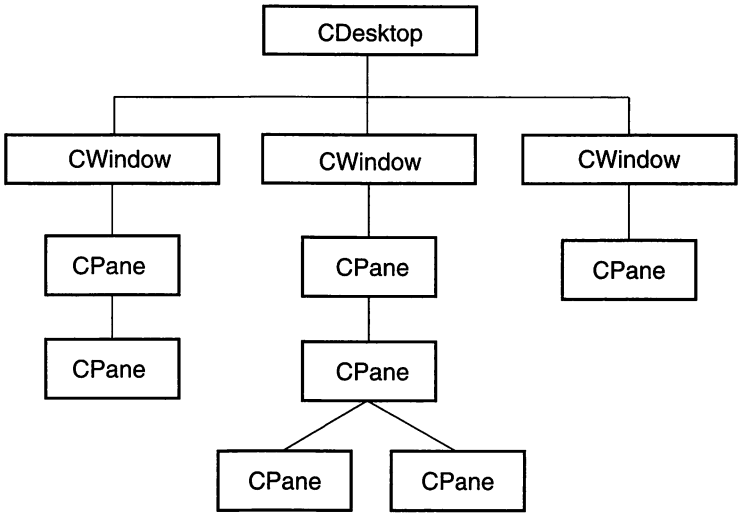

Figure 1-1
Visual Hierarchy example

the windows. Each window is represented by a CWindow object that may contain one or more CPane objects. CPane is really the base class for a whole collection of visible object classes, including CPanorama, CSubviewDisplayer, CIconPane, CRadioGroupPane, CPopupPane, CSizeBox, CScrollPane, CControl, and all of their subclasses. Figure 1-1 illustrates one possible containment model, but a given window can contain any number of panes and each pane can contain other panes, as is illustrated for the middle CWindow object in the figure.

It's not important to understand all of the details regarding the visual hierarchy right now. We will look at the individual components in a later chapter. The important point is that the CDesktop object is responsible for all of the CWindow objects, and each of the CWindow objects is responsible for all of its CPane objects. It is also true that CPane objects can be responsible for any subpanes that they contain. Mouse and window activation and update events travel down in the hierarchy from the CDesktop to the CWindow to the individual CPane objects. I will cover how these events are handled in a later chapter.

Defining the Chain of Command

In addition to the visual hierarchy described in the previous section, I need to introduce the concept of the "chain of command." While mouse clicks and window events travel down the visual

hierarchy and reach their appropriate destination because of the position information they contain (that is, the coordinates of a mouse click), commands (including menu commands, click commands, and keystrokes) travel along a different path, according to a different set of rules. The TCL defines a global variable called `gGopher`, which contains the pointer to the first object to receive any command. The object to which the `gGopher` variable points is usually an active pane in the currently active window and is referred to as the current "gopher." The chain of command for a typical application hierarchy is shown in Figure 1-2.

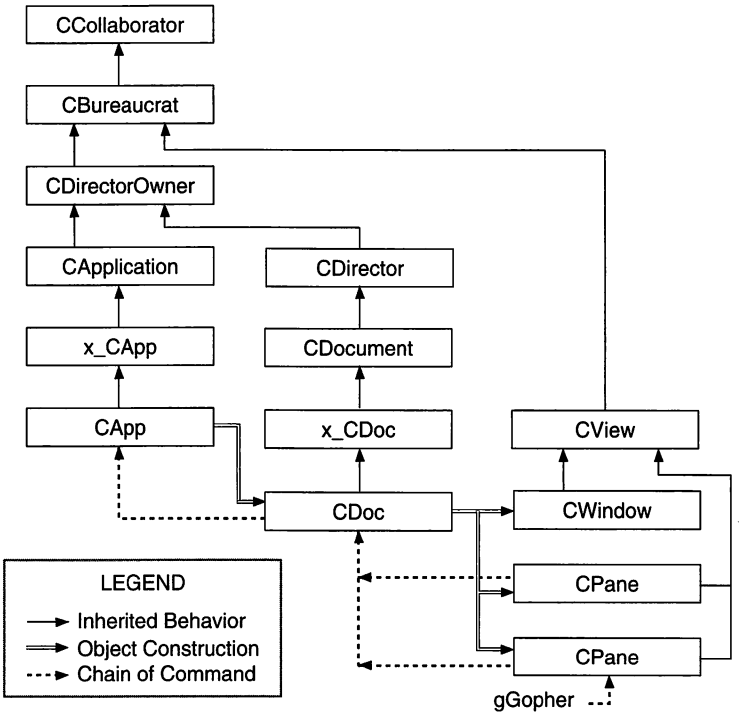

Figure 1-2
Chain of command for typical application

Figure 1-2 illustrates the hierarchy of the major objects in a typical application. Although the CPane objects are part of the chain of command, the CWindow object rarely assumes this role. Instead, the window's director (the CDoc object in this case) is next in the chain of command, after a CPane object has relinquished that role. The CApp object is the last to handle a command and does so only if all other members of the chain of command have passed on that responsibility.

Note that the gopher points directly to (what is assumed to be) the active pane in the currently active window. Commands and keystrokes are sent to this object first. If that object is unable to handle the command or keystroke, it is passed up to the next level in the class hierarchy, not to another pane in the same window. If the document object (CDoc in our case) decides that another CPane object in the current window should handle the command, it will arrange for that pane to become the gopher and will reissue the command. It is rarely necessary for the document object to operate in this manner. A given CPane object is active because the user has clicked on it, causing it to activate.

When an application is first initialized (I will cover that topic in much more detail later) the CApp object becomes the gopher. When the user chooses the **New** command from the **File** menu, the CreateDocument member function in the x_CApp class creates a new instance of the CDoc class and then creates the initial CWindow object and its enclosed CPane objects. The CWindow object is created by its CDirector object (which is the CDoc object in Figure 1-2) and the director's Activate member function is called. This results in the CDirector (CDoc) becoming the gopher. When the user clicks in one of the CPane objects in the window, then that object becomes the gopher, receiving all further commands and keystrokes until some other pane or the director becomes the gopher. Your program can force a given pane to become the gopher by calling BecomeGopher (a member function of the CBureaucrat class) for that pane. When a window is closed or deactivated, its director will become the gopher until that or another window is activated and one of its panes is clicked.

If all of the talk about gophers and the chain of command seems confusing, don't worry. These concepts should become clear after you learn a bit more about how the TCL operates. Just remember that most of the machinations described in the foregoing sections happen automatically, without the need for you to intervene.

More About Commands

Commands are events that are created by virtue of some action taken by the user. The most well known of these are the commands associated with the user's choice of a menu item. Officially, the vari-

ous items in an application's menus are called menu commands. When the user chooses a menu command, a command token (a unique 32-bit value) is sent to the current gopher to be handled. If the current gopher can't handle the command, then the command is passed on to the next higher member of the chain of command to handle. The application object is the last to receive a command if no other object is capable of handling it. If the application can't handle the command, then the command is thrown away.

Commands can also be generated when the user presses the Command-key shortcut associated with a menu command. This action results in the creation of the same kind of command token, which is then sent to the current gopher, as described earlier.

In addition to menu choices and keyboard shortcuts, commands can also be associated with mouse clicks on push buttons, checkboxes, or radio buttons. When you add one of these buttons to a window or dialog either by using the VA or creating it programmatically, you can specify a "click command" for the button. When the application is running and the user clicks on that button, a command token will be generated, just as was the case for a menu command or keyboard shortcut, and the command will be sent to the current gopher to handle.

Using the Visual Architect

The Visual Architect (VA) is both a graphic user interface design tool and a code generator. Using the VA, you can create the user interface for applications that have multiple windows, open multiple files of the same or different types, display various modal or modeless dialogs or floating tool palettes, and respond to commands associated with menus that you specify. You can also construct splash screens and error alerts using the VA.

The entire user interface need not be designed all at once. You can design the various features of your application one by one, generate code, implement and test any custom features, and then proceed to design the next set of features, repeating these steps as many times as necessary.

When any particular set of features has been designed, you can instruct the VA to generate code to implement the appearance of

those features. Of course, any custom behavior associated with the user interface features will have to be written manually, but the VA includes comments in the generated code to instruct you where to add your custom code, and, in general, what it should accomplish. When you generate code, the VA will generate a pair of files for each major feature (application, document, menu, window, alert, dialog, and so on). You can see an example of this technique in Figure 1-2, where the application class has been broken into a base class (x_CApp) and a subclass (CApp). The document subclass is handled in the same way with the creation of the x_CDoc and CDoc classes. Each of these is written into a separate source file, with its own unique header file. The class declaration is written into the header file (for example, **CDoc.h**), whereas the member functions are implemented in the code generated into the source file (for example, **CDoc.cp**).

Each time you generate code, the VA will regenerate the appropriate base class files (both source and header). Once generated, the subclass files are never regenerated. The approach allows you to modify the user interface at any time without losing any of your custom code in the subclass files.

You can use the VA to develop a user interface and its corresponding code in an incremental manner; however, if you make drastic changes to your user interface, you may find that you will have to modify some of the code in a subclass file to compensate for these changes. So although the VA is an incredible help in generating the necessary skeleton code for a user interface design, it helps to have spent some time with pencil and paper, sketching your expected user interface design, before you actually use the VA to implement the design and generate code.

When you update your project for the first time, you will find that you will need to compile almost every class in the TCL. Thereafter, you need only compile the files that have changed (or any files that depend upon the changes you have made). The VA provides an intimate interface with the Symantec development environment, allowing newly generated source modules to be added automatically to your project.

As an extra added bonus, the VA includes a simulator that allows you to view the final appearance of windows, dialogs, and alerts, from within the designer program. With windows or dialogs that

contain controls, you can even test the functionality of those controls in the context of the simulated display. This allows you to see the effect of any changes and to delay the generation of code until you are sure that the appearance of each feature is what you want in the final form of the application.

When you design the various user interface elements that make up your application's appearance, the VA writes the settings for each of these into special CVue resources. The CVue resources can not be edited by a normal resource editor (for example, ResEdit or Resorcerer) because the resources consist of a highly complex stream of data. When your application is launched, the CVue resources are read into memory, as needed, and are made available for use by the remainder of your application and the TCL, automatically. I will cover more of this topic when discussing the creation of windows and dialogs in later chapters.

Chapter Summary

As you may have guessed by this time, I will be using the VA to illustrate how various user interface features can be designed. The generated code, as well as custom code, will be used to elaborate on how the features are best implemented in the context of the TCL. In so doing, I hope to cover a number of popular user interface features and their alternatives.

I will, for example, cover application structures that consist of single or multiple standard document windows, multiple files, and the use of modal or modeless dialogs. I will do my best to present you with examples of how to create whatever you need to by using a combination of the VA, the TCL, and whatever custom code is needed.

This book does not come with a disk because I don't intend to show you a bunch of simple do-nothing applications or even one large application. Instead, I am including a large number of tutorials, code fragments, and detailed code descriptions that are organized in a fashion that you will be able to pick up and use without any additional assistance.

The adventure begins in the next chapter.

Chapter 2
Building the Application's Foundation

This chapter describes how to create a skeleton TCL application using the Visual Architect (VA) tool. The application will contain a single window, with a color PICT image created by the VA, and standard **Apple**, **File**, and **Edit** menus. In the course of examining the generated code, we will discover how the application is constructed, how it is initialized, how various important TCL objects are created, and how all of these work together with the generated code to produce a fully functioning application.

Even though the application I will create and describe in this chapter has limited functionality, it does serve as an excellent basis for illuminating the foundation on which all TCL applications are built.

Creating the Skeleton Application

To commence development of the skeleton application for use in this chapter, the first step is to launch the Symantec C++ development environment. The Project Manager will display a dialog that offers a choice of various types of new projects. We will choose to create a Visual Architect Project. This will result in the population of the Symantec C++ project window with all of the files that are necessary to the creation of this type of project. In addition to the files associated with the TCL, the project will include a file called **Visual Architect.rsrc** that contains the initial resources that are applicable to a VA project. That file will also contain the resources that are unique to your user interface when you choose to change the default design. For now, we will use the default design. With the foregoing as a backdrop, the procedure for creating the skeleton project is as follows:

1. Name the new project **Skeleton.** The Project Manager will automatically add all of the initial files needed to commence the development of the project.

2. After the project is created, double-click on the **Visual Architect.rsrc** file in the project window to launch the Visual Architect application. When you do so, you will see a menu bar with VA's standard menus and a main window titled "Visual Architect.rsrc" that contains a single entry called "Main." The main window appears as shown in Figure 2-1. The entry for "Main" is the default view that is supplied by the VA for all new projects. The VA supports various types of views, and the default view just happens to be a document window that contains both horizontal and vertical scroll bars, a size box, and a go-away box.

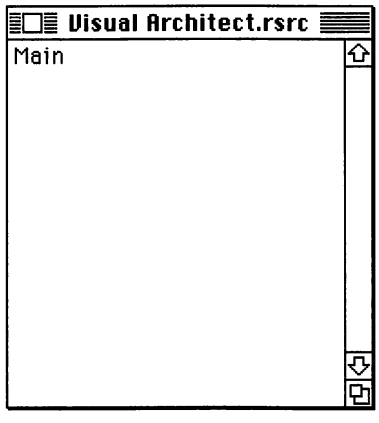

Figure 2-1
Visual Architect main window appearance

3. Double-click on the word Main in the Visual Architect.rsrc window to show the Main window's contents. What you'll see is a color version (if you are using a color monitor) of what is depicted in Figure 2-2. The default window contains a PICT image (named Pict1) that has been imported from the resource fork of the **Visual Architect.rsrc** file and also a static text field that contains the "hello world" text. After you have

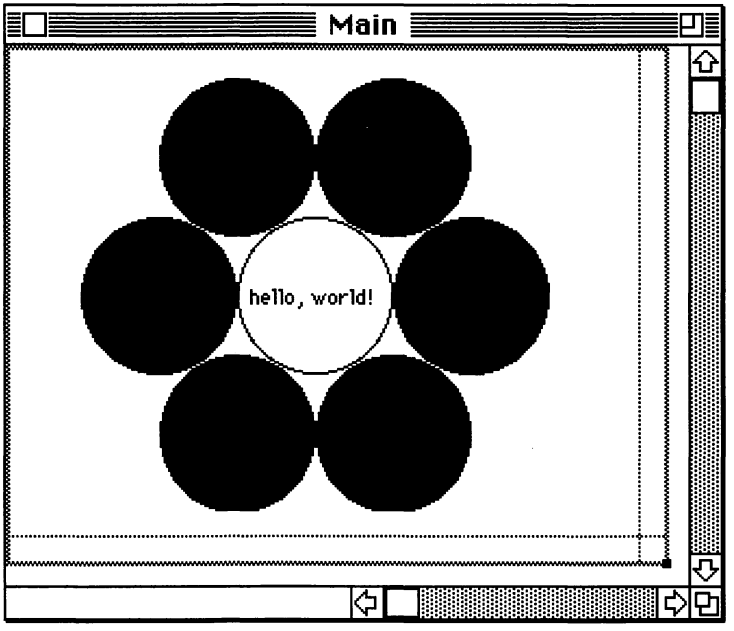

Figure 2-2
Contents of Main window supplied by the Visual Architect

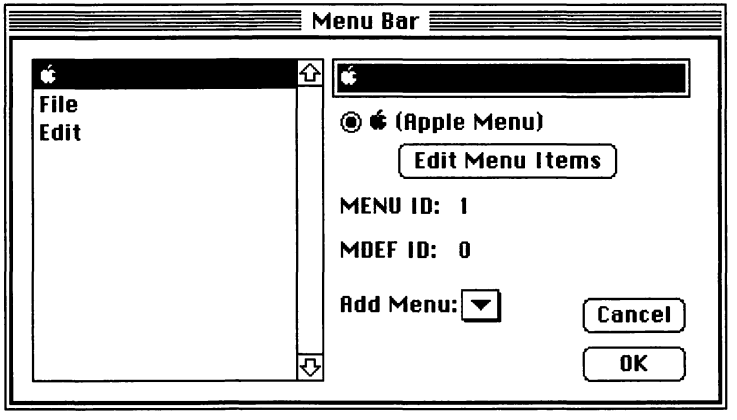

Figure 2-3
Menu Bar editor window in Visual Architect

finished looking at the contents of the window called Main, you can close that window.

4. To view and/or modify the menu bar, its menus, and the various menu commands, pull down the **Edit** menu and choose the **Menu Bar** command. A window displaying the current menu bar and its included menus will be shown, as illustrated in Figure 2-3. Note that **Apple**, **File**, and **Edit** menus are included in the default menu bar. If you wish to examine the

contents of any of the menus, select the menu in the list and then click the Edit Menu Items button. If you do that, you will be able to see all of the menu commands for the selected menu. When you are done, click Cancel in the Menu Items window, and then click Cancel again to dismiss the menu bar editor when you are finished examining the menu bar and its menus. We will not be modifying any of the menus in the skeleton application described in this chapter.

5. At this point we have looked at the user interface elements that are provided by default for all VA projects. The elements include a single window that contains a PICT image and a menu bar with **Apple**, **File**, and **Edit** menus. The next step in the creation of the skeleton application is to choose **Generate All** from the **Project** menu (shown immediately to the right of the Windows menu) in the VA, as an icon. When you perform this step, the VA will display first a window indicating the name of each file it is generating, then a message indicating that it is updating your project file.

6. After the code generation step is complete, you can quit the VA application by choosing **Quit** from its **File** menu. If the VA asks you whether you wish to save changes, click the No button in that dialog (this is just to ensure that the interface resources are all still in their default form).

After the foregoing steps are complete, your project has been updated with the files containing skeleton code to implement the basic features of the application. If you compile the files that comprise the project and then run the application, you will see the window and menu bar that were included in the default VA interface design.

Examining the Skeleton Application Code

The main purpose of creating a skeleton application is to produce a complete set of source code that we can dissect and analyze so that you will have a better understanding of the foundation on which all TCL applications are built.

The first step in analyzing the source code is to describe each of the source and header files produced by the VA when it generated code and added it to the project. The various files and their roles in the context of the application are as follows:

CApp.cp — This file is a subclass of the x_CApp file, which inherits its functionality directly from the CApplication class of the TCL. The **CApp.cp** file contains member functions that override or enhance the functionality of its base class and is the file that you would modify to include application-wide changes in the program. Once generated by the VA, the **CApp.cp** file will never be regenerated, unless the original file is deleted. The file is generated only once and you are free to modify this file.

CApp.h — This header file contains the declaration of the CApp class, its member functions, and member variables. It is generated only once, and you are free to add function and variable declarations to this file.

x_CApp.cp — This file contains the member functions that inherit and supplement the behavior of the functions in the CApplication class, from which it is derived. The x_CApp class is the direct ancestor of the CApp class and the direct descendent of the CApplication class. This file is regenerated whenever the VA deems it necessary to do so. You should not modify this file directly.

x_CApp.h — The declarations of the x_CApp class, its member functions, and member variables are contained in this header file. This file should never be modified directly. It will be regenerated by the VA whenever it is necessary to do so.

CMain.cp — This file contains the member functions for the CMain class. This class is derived from the x_CMain class and is the document subclass. The CMain class is where you add any code to implement functions that are associated with your document or its data. The file is generated only once.

CMain.h — This header file contains the declaration of the CMain class and its member functions and member variables. The file is generated only once. You are free to add function and variable declarations to this file.

x_CMain.cp — This file contains member functions that override and supplement the behavior of the CSaver class (from which it is derived), that, in turn, is derived from the TCL's CDocument class. If your application does not perform file I/O, then the x_CMain class

will be derived directly from the CDocument class. The file is regenerated by the VA whenever it is necessary to do so. You should not modify the contents of this file at any time.

x_CMain.h — This header file contains the declarations of the x_CMain class and its member functions and member variables. The file is regenerated by the VA whenever it is necessary to do so. You should not modify the contents of this file.

References.cp — This file includes most of the TCL header files that will be used by your project and also contains a function called ReferenceStdClasses that contains TCL_FORCE_REFERENCE macros to force the standard TCL classes to be linked into your final application. This file is regenerated each time you generate code with the VA. If you need to force class references for your own classes, there is a ForceClassReferences member function in the CApp class where you can add these function references. We will cover this topic in more detail later.

References.h — This header file includes the **TCLForceReferences.h** file that contains the definitions of the TCL_FORCE_REFERENCE and the TCL_FORCE_TEMPLATE_REFERENCE macros. It also contains the prototype for the ReferenceStdClasses function.

main.cp — Finally, the main module of your program is contained in this file. The main function in the file is given control when your application commences execution. The file is generated only once and is very short. You should not need to make changes to this file; however, you are free to do so.

Starting in the Main Function

I intend to examine the skeleton code (and therefore the TCL) by following the thread of execution of the application, starting with the **main.cp** module. The main function is not part of any class. It is not object-oriented and is merely the standard function that receives control when any C or C++ application is executed. The code for the main function is as follows:

```
/************************************************************
 main.cp

          Main Program

   Copyright © 1994 My Software Inc. All rights reserved.

   Generated by Visual Architect™ 2:56 PM Thu, Sep 22, 1994
************************************************************/
```

```
#include "CApp.h"

void main()
{
   CApp *application = TCL_NEW(CApp,());
   application->ICApp();
   application->Run();
   application->Exit();
}
```

As you can see from the foregoing code, the main function is very small. The code begins by creating a CApp object, storing its pointer into the temporary stack variable called `application`. Rather than using the C++ **new** operator, the foregoing code uses a macro called TCL_NEW that, by default, simply expands into the use of the **new** operator for the class. If the Bedrock Exception Library exception handling facilities are being used, then the use of the TCL_NEW macro will result in additional code being generated.

The point of bringing up the different possible expansions of the TCL_NEW macro is to advise you to use this macro when you create new objects in your own code. If you do so and later decide to use more extensive exception handling, you will be able to do so by including a single `#define` statement in the Prefix settings for your project.

When the CApp object is created, the constructor functions for all of its ancestors and then the CApp object itself are executed, in that order. The class hierarchy after the CApp object has been constructed is shown in Figure 2-4.

Constructing the CApp Object

Using the foregoing figure as a guide, you can see that the constructors for the CCollaborator, CBureaucrat, CDirectorOwner, CApplication, CAppleEventObject, x_CApp, and then the CApp class are executed. Some of these objects do not define default constructor functions; however, the compiler will generate code to perform any needed construction, even if an explicit constructor does not exist. The following sections describe the construction process and what is accomplished in each step.

CCollaborator Construction

The constructor for CCollaborator class is the first to be executed. In this case, a default constructor exists. The function performs the following actions:

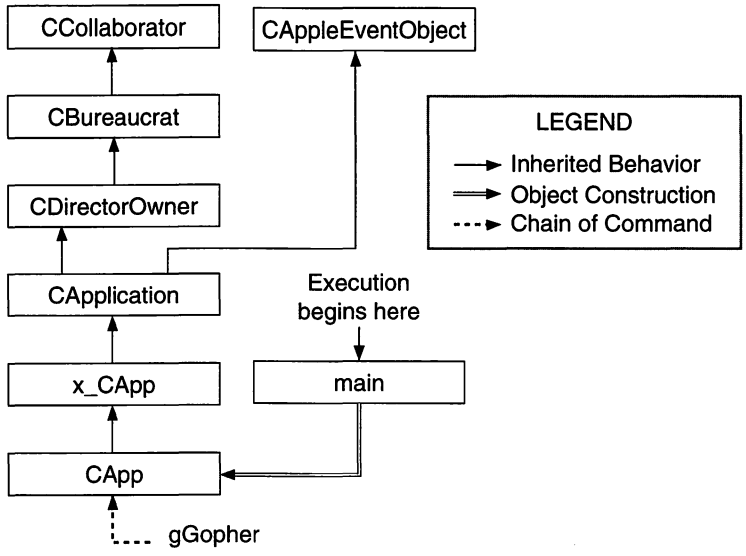

Figure 2-4
Class hierarchy after CApp object is constructed

1. The `itsProviders` member variable is set to `NULL`. This ensures that the list of providers for this CCollaborator object is empty.

2. The `itsDependents` member variable is set to `NULL`. This ensures that the list of dependents for this CCollaborator object is empty.

The function of CCollaborator objects will be discussed in a later chapter; however, simply put, CCollaborator objects provide the means to communicate changes to objects in the visual hierarchy with one another through a mechanism of providers and dependents that is managed entirely by the TCL.

CBureaucrat Construction

The CBureaucrat class has a default constructor with a single argument (`aSupervisor`) for which the CDirectorOwner derived class's constructor specifies an initializer value of `NULL`. Therefore, when the constructor executes, it sets the value of the application's `itsSupervisor` member variable to the `NULL` value.

CDirectorOwner Construction

The CDirectorOwner class has a constructor with a single argument (`aSupervisor`), for which an initializer in the CApplication derived class specifies a value of `NULL`. When the

CDirectorOwner constructor continues execution, it sets the value of the `itsDirectors` member variable (a list) to `NULL`, indicating that the list is empty.

CApplication Construction

The CApplication class contains a default constructor that provides initializers for the constructors of both the CDirectorOwner and CAppleEventObject base classes from which it is derived. When the CApplication constructor is executed, it performs the following functions:

1. The CApplication class inherits behavior from the CAppleEventObject class, so the constructor for that class is called with an argument of `FALSE`, which is taken to be the `isDisposable` argument for the constructor.

2. The CAppleEvent class's constructor takes the `FALSE` value and stores it into the `disposable` member variable. This ensures that the application object is not disposed when Apple Event handling is complete (I will discuss Apple Events in a later chapter). In addition, the `elementID` is a unique identifier for this object that is set to the value of the `lastElementID` member variable and then that member variable is incremented by one.

3. Finally, the pointer to this object (if you've lost track by now, we're still talking about the CApp object) is entered into a linked list of CAppleEventObject objects.

The last action of the constructor for the CApplication class is to call an additional member function of that class to complete the details of construction. This function is called CApplicationX and its code in the TCL is as follows:

```
void CApplication::CApplicationX()
{
   gApplication = this;        // initialize the global pointer

   phase = appInitializing;    // indicate initialization phase
   running = FALSE;
   urgentsToDo = FALSE;
   newWindowOnStartup = TRUE; // derived constructor can
                              // override

   recording = FALSE;
   factoring = kFactorWhenRecording;

   gDesktop = NULL;            // pointers to helpers
```

```
    gClipboard = NULL;
    itsSwitchboard = NULL;
    gDecorator = NULL;
    gBartender = NULL;
    gError = NULL;

    sfFileFilter = NULL;
    sfGetDLOGHook = NULL;
    sfGetDLOGFilter = NULL;

    lastTask = NULL;

    itsIdleChores = NULL;
    itsUrgentChores = NULL;

    rainyDay = NULL;

    urgentsToDo = FALSE;
    running = TRUE;
    lastTask = NULL;
    undone = FALSE;

    gInBackground = FALSE;
    gSignature = '???\?';
    gSleepTime = 0;              // We want an early first Idle

    gGopher = this;
    gLastViewHit = NULL;
    gLastMouseUp.when = 0L;
    gClicks = 0;

    InitToolbox();               // Initialize Toolbox Managers

    appResFile = CurResFile(); // Remember application resource
                                 // file
}
```

As you can see in the foregoing code, the CApplicationX function performs many initializations. These prepare the application for future operation with a "clean slate."

The `gApplication` global variable is set to point to the CApp object. This gives you the ability to access the application object from any other object in the program.

The `urgentsToDo` member variable is initialized to `FALSE`, indicating that there are no pending urgent chores.

The `newWindowOnStartup` member variable is set to `TRUE`, indicating that the default behavior for the application is to create a document and new windows when the application starts up. You can override this behavior in your own code.

The `recording` member variable is set to `FALSE` and the `factoring` variable is set to the value of the `kFactorWhenRecording` constant. These actions specify that the application is currently not recording and that the application is factored only when it is recording. You can override these settings in your own code.

A number of other variables are initialized to `NULL`. These include a number of global variables, such as `gDesktop`, `gClipboard`, `gDecorator`, `gBartender`, and `gError`. In addition, other variables, procedure pointers, and lists are set to `NULL`. These include the `itsSwitchboard` pointer and the `sfFileFilter`, `sfGetDLOGHook`, and `sfGetDLOGFilter` procedure pointers.

The `lastTask` (CTask object pointer) and the `itsIdleChores` and `itsUrgentChores` lists of chores are set to `NULL`, making sure these lists are empty.

The `running` variable is set to `TRUE`, indicating that the application is running. The `undone` variable is initialized to `FALSE`, indicating that the last task has not been undone. The `gInBackground` variable is set to `FALSE`, indicating that the application is running in the foreground. The `gSignature` variable (creator code) is set to '????'. The `gSleepTime` variable is set to 0, ensuring that when the application begins processing events, it will generate an early idle event. The `gGopher` variable is set to point to the CApp object, and the `gLastViewHit` variable is set to `NULL`, indicating that the last view in which the mouse was clicked has not been specified. The time at which the last mouse-up event occurred is set to 0 in the `gLastMouseUp` variable and the number of previous mouse clicks is set to 0 in the `gClicks` variable.

After the foregoing initialization steps are taken, the CApplicationX function calls the InitToolbox function to initialize the Mac toolbox managers. These are called in the following order: InitGraf, InitFonts, InitWindows, InitMenus, TEInit, InitDialogs, and InitCursor.

Finally, after initializing the toolbox managers, the CApplicationX function sets the `appResFile` variable to the current resource file by calling the CurResFile toolbox function.

X_CApp and CApp Construction

Neither the x_CApp nor the CApp classes have explicit constructor functions, so the construction of the CApp object is complete after default constructors are executed for these classes.

As you can see in Figure 2-4, the `gGopher` variable, which represents the beginning of the chain of command, is pointing to the CApp object after construction of that object is complete. What this means is that any events, such as mouse clicks or keystrokes, will be sent to the CApp object when the application begins processing events.

Initializing the Application Object

After the application object (CApp) has been constructed fully, the code in the main function continues execution by using the pointer to the CApp object (`application`) to call the ICApp member function of the CApp class.

Initializing CApp

Refer back to the code for the main function (see page 16). After the CApp object has been constructed, you will see that the function calls the ICApp member function of the CApp object to perform additional initialization.

The ICApp member function calls the Ix_CApp member function inherited from the x_CApp class. The arguments to the call to Ix_CApp are as follows:

1. The first argument (`extraMasters`) specifies the number of times that the MoreMasters toolbox routine will be executed. In the default case, the argument to the Ix_CApp function is the number 4, indicating that MoreMasters should be called four times.

2. The second argument (`aRainyDayFund`) specifies the size of the "Rainy Day" fund—an allocated block of memory that is held in reserve in case a low-memory situation requires it to be used. The default value is 24,000 bytes.

3. The third argument (`aCriticalBalance`) specifies the portion of the Rainy Day fund to be reserved for use in critical operations such as in execution of the **Save** or **Quit** commands. The value of this argument is 20,480 bytes.

4. The fourth argument (`aToolboxBalance`) specifies the portion of the Rainy Day fund to be reserved to allocate a "system bomb" dialog in the case where a toolbox call fails. The value given for this argument is 2,048 bytes.

Initializing CApplication

When the Ix_CApp function executes, it calls the IApplication function inherited from the CApplication class with the foregoing four arguments. The IApplication function begins by calling the IApplicationX function, which performs the following actions:

1. The first task is to call the InitMemory member function of the CApplication class to initialize the memory usage for the application. The function proceeds as follows:

   a. When the InitMemory function commences execution, it validates that the value of the `aRainyDayFund argument` is greater or equal to the value of the `aCriticalBalance`. In addition, the function validates that the `aCriticalBalance` is greater or equal to the value of the `aToolboxBalance` argument. If either assertion is 23`FALSE`, then the application will display an error dialog and terminate execution.

   b. After validating the arguments, the function calls the MaxApplZone toolbox routine to expand the application zone to include all available heap memory.

   c. When the application zone has been fully expanded, the MoreMasters toolbox routine is called to allocate the specified number of master pointer blocks, the default number of which is four.

   d. The InitMemory function calls SetGrowZone with a pointer to a function to be called when a memory request cannot be filled. This is specified, by default, to be a global function called GrowZoneFunc, which is in the **CError.cp** source file. The GrowZoneFunc, when called, attempts to release enough memory to satisfy a failed memory-allocation request.

   e. The function then stores the values in the argument list into member variables of the application object (CApp). The `rainyDayFund` variable is set to the value of the

aRainyDayFund argument, the criticalBalance variable is set to the value of the aCriticalBalance argument, and the toolboxBalance variable is set to the value of the aToolboxBalance argument.

f. In addition to storing the values of the function's arguments into the CApp object's member variables, the function initializes the values of other member variables for the application object. The rainyDayUsed variable is set to FALSE, the memWarningIssued variable is set to FALSE, the canFail variable is set to FALSE, the value of the inCriticalOperation variable is set to FALSE (indicating that a critical operation is not in progress), the tempAllocation variable is set to 0, and the value of the _MMPrimitiveAllocate variable is set to FALSE.

g. The final action of the InitMemory function is to allocate a handle in the heap that is the size specified in the rainyDayFund variable and then verify that the memory has been allocated. If the memory request fails, then the application will display an error alert and will stop execution.

2. After the InitMemory function completes its execution, the IApplicationX function resumes execution and calls the InspectSystem member function of the CApplication class. This function performs the following actions:

   a. The InspectSystem function determines whether the _GestaltDispatch trap is available in the user's system, whether the WaitNextEvent trap is implemented, and whether the _ScriptUtil (Script Manager) trap is available. It stores these values into a series of fields in the global gSystem structure. The foregoing values are stored into the hasGestalt, hasWNE, and hasScriptMgr fields. The scriptsInstalled field is set to 1, indicating that a single script only (assumed to be Roman) is installed.

   b. The SysEnvirons toolbox routine is called to determine whether the user's system has Color QuickDraw installed, whether it has a floating-point unit, and which version of

the operating system is being run. These values are stored into the `gSystem` structure in its `hasColorQD`, `hasFPU`, and `systemVersion` fields. If the call to SysEnvirons results in an error, then the values of the `hasColorQD` and `hasFPU` fields are set to `FALSE`, and the value of the `systemVersion` field is set to 0.

c. If the `hasGestalt` field of the `gSystem` structure contains a value of `TRUE`, then the Gestalt toolbox routine is called to initialize additional fields in the `gSystem` structure. These include the `hasAppleEvents`, `hasAliasMgr`, `hasEditionMgr` and `hasHelpMgr` fields. In addition, if the value of the `hasScriptMgr` field is `TRUE`, then Gestalt is called to access the number of scripts installed in the system and store this into the `scriptsInstalled` field. Finally, Gestalt is called to determine whether the Process Manager is installed and store the Boolean result into the `hasProcessMgr` field.

d. If the `hasGestalt` field has a value of `FALSE`, then the `hasAppleEvents`, `hasAliasMgr`, `hasEditionMgr`, `hasHelpMgr`, and `hasProcessMgr` fields are set to `FALSE`. In addition, if the value of the `hasScriptMgr` field is `TRUE`, then Gestalt is called to access the number of scripts installed in the system

3. After the InspectSystem function returns control to the IApplicationX function, that function resumes execution. When the **CApplication.cp** source file is compiled, if the developer's system is not a Power Macintosh, then a call to the InstallPatches member function is compiled and is executed next. If the developer's machine is a Power Macintosh, then the function call is not compiled. The Install Patches function patches the ExitToShell trap to reference the ETS_Patch code and patches the LoadSeg trap to reference the LoadSeg_Patch code. The IApplicationX member function continues execution by performing the following actions:

   a. The `gInEnvironment` global variable is set to `TRUE` if a `CODE` 0 resource is present; otherwise, the variable is set to `FALSE`. This indicates whether the application is executing within the Project Manager environment.

b. A CChoreList object is created and the value of the `itsIdleChores` variable is set to point to the empty list. Another CChoreList object is created and its pointer is stored in the `itsUrgentChores` variable.

c. The IBeam cursor resource is accessed and its handle is stored into the `gIBeamCursor` variable. The handle for the IBeam cursor is set to not purge the resource. The Watch cursor resource is also accessed and its handle is stored into the `gWatchCursor` variable. The handle is also set not to be purged.

d. A "utility" region is created and its handle is stored into the `gUtilRgn` variable. This region is used for various purposes by portions of the TCL.

e. If the `hasAppleEvents` field of the `gSystem` structure holds a `TRUE` value, then the process serial number is accessed and stored into the `psn` variable and an Apple Event descriptor is created and stored into the `self-PsnDesc` variable.

Making the Application Helpers

Although the following is part of the initialization process for the application object, it is convenient to treat these steps as separate tasks because they relate to the construction of other important objects that are unrelated to the application object, but nonetheless very important to the application as a whole.

The MakeHelpers member function is responsible for creating some of the more important objects that interact with and perform many tasks on behalf of your application. When execution of the MakeHelpers function is complete, the structure of your application will look like what is shown in Figure 2-5.

Loading Important Resources

One of the first tasks of the MakeHelpers function is to access a number of 'STR,' 'STR#,' and 'ALRT' resources so that these are located permanently in memory during the execution of the application. If any of the standard resources is not accessible, then the application will display an error alert and then terminate.

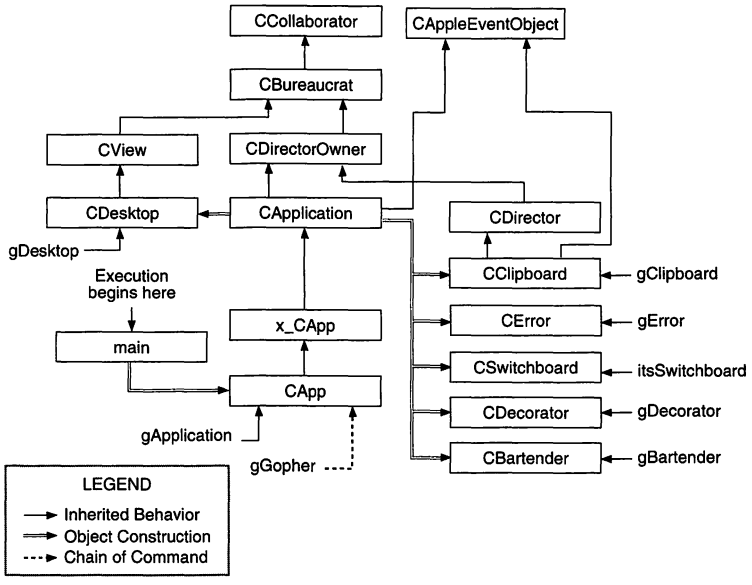

Figure 2-5
Application object after construction and initialization

Creating the CSwitchboard Object

After the resources have been loaded, the CSwitchboard object is created by calling the MakeSwitchboard member function. The CSwitchboard object is responsible for accessing the Macintosh event queue to acquire each event and then dispatch that event to the appropriate destination object. The MakeSwitchboard function creates the CSwitchboard object and stores its pointer into the `itsSwitchboard` member variable. After doing so, it calls the InitAppleEvents function using the pointer to the CSwitchboard object. The InitAppleEvents function performs the following tasks:

1. The AEObjectInit function is called to initialize the object support library.

2. The `hasAppleEvents` field of the `gSystem` structure is tested to determine whether the user's system is capable of processing Apple Events. If so, then the function continues by establishing a rather large number of event handlers, object accessors, callbacks, and coercion handlers. It does this by calling the InstallEventHandlers, InstallObjectAccessors, InstallCallbacks, and InstallCoercionHandlers functions.

Creating the CError Object

After the MakeSwitchboard function returns, the MakeHelpers function calls MakeError, which creates a CError object and stores its pointer into the `gError` variable. The CError object is used for reporting unrecoverable errors and must be constructed when the application is initialized so that it will be available whenever needed to post important error alerts.

Creating the CDesktop Object

A CDesktop object is next to be constructed by calling the MakeDesktop function. The pointer to the CDesktop object is stored into the `gDesktop` variable. The CDesktop object is responsible for managing all of the windows that are created during the execution of the application. During construction of the CDesktop object, the following actions are taken:

1. The CDesktop's list of windows, which are kept in a CWindowList object called `itsWindows`, is set to `NULL`, indicating that the list has not been allocated.
2. The CDesktop member variable called `topWindow` is set to `NULL`, indicating that no window currently exists.
3. The CDesktop's list of floating windows is created as a CWindowList object, whose pointer is stored into the `itsFloats` member variable. Although this list is allocated, it is initially empty.
4. The CDesktop's member variable called `topFloat` is set to `NULL`, indicating that no floating windows currently exist.
5. The CDesktop constructor then calls the IDesktopX member function to complete the initialization of the CDesktop object.
6. The IDesktopX function begins by testing whether the user's system contains Color QuickDraw. This is determined by checking the value in the `hasColorQD` field of the `gSystem` structure. If the test result is `TRUE`, then the function allocates a pointer that is the size of a color grafport (CGrafPort) and stores the pointer into the `macPort` member variable. It then opens the port by calling the OpenCPort toolbox routine. If the user's system does not support Color QuickDraw, then the IDesktopX function allocates a pointer the size of a nor-

mal grafport (GrafPort) and stores the pointer into the `macPort` member variable. It then opens the port by calling the OpenPort toolbox routine.

7. The IDesktopX function continues by getting the size of the "gray region" of the primary monitor (the screen size minus the space taken by the menu bar), sets the size of the port to include the entire gray region, and then sets the port's origin to the top-left corner of the region. The clipping region is set to the same size as the port and the gray region is copied to the `visRgn` field of the port defined by the `macPort` variable. The `visible`, `active`, and `wantsClicks` variables of the CDesktop object are set to `TRUE` values.

8. Finally, the IDesktopX function creates a CWindowList object, stores its pointer into the `itsWindows` member variable, and then sets the value of the `topWindow` variable to `NULL` to make certain that the CDesktop object is initialized to indicate that no windows currently exist.

Creating the CClipboard Object

The CClipboard object implements a standard Macintosh clipboard and supports a window to display the contents of TEXT or PICT images. You can derive your own clipboard class from the CClipboard to provide support for other clipboard data types. The CClipboard object is created by calling the MakeClipboard function from within MakeHelpers. When the CClipboard object is constructed and initialized, it will have a more detailed set of associated objects than are shown in Figure 2-5. The structure of just the CClipboard portion of the application under construction is shown in Figure 2-6.

The MakeClipboard function takes the following actions:

1. MakeClipboard creates a new CClipboard object, calling its constructor with a `TRUE` argument, and stores the pointer into the `gClipboard` global variable.

2. The CClipboard constructor uses the single argument to specify whether the clipboard has an associated window. The CClipboard class inherits functionality from both the CDirector and the CAppleEventObject classes. The CClipboard constructor supplies an initializer value of `gApplication`

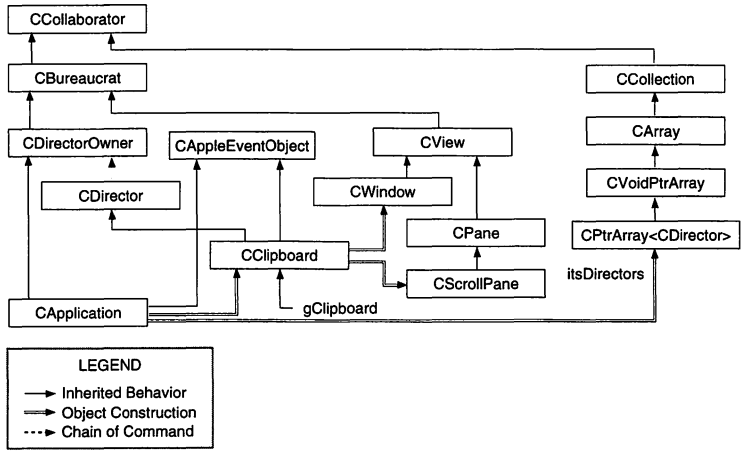

Figure 2-6
CClipboard object construction and initialization

for the CDirector constructor to use and an initializer of `FALSE` for the CAppleEventObject constructor to use.

3. The constructor for CDirector passes on the initializer value of `gApplication` for its CDirectorOwner base class to use as its `aSupervisor` argument. The CDirectorOwner constructor passes on the `gApplication` value as the initializer for its CBureaucrat base class, which uses that value as its `aSupervisor` argument. This results in the `itsSupervisor` member variable being set to the `gApplication` value. In essence, the CApp object is being made the supervisor (in the chain of command) of the CClipboard object. The CCollaborator constructor has no arguments, but does have a default constructor function.

4. The constructors for the CClipboard derived class are called in order, from CCollaborator, then CBureaucrat, then CDirectorOwner, then CDirector, CAppleEventObject, and then CClipboard.

5. When the constructor for CDirector executes, it initializes the values of several of its member variables. The `itsWindow` variable is set to `NULL`, indicating that no window currently exists. In addition, its `active` variable is set to `FALSE`. A director cannot be active without having a window. In addition, the `activeWindOnResume` variable is set to `FALSE`, the `alreadyClosing` variable is set to `FALSE`, and the `wasDirty` variable is set to `FALSE`. Finally, because the con-

structor is called with an argument of `gApplication`—the pointer to the application object—the constructor also calls the IDirector function in the CDirector class, passing on the `gApplication` argument to that function.

6. The IDirector function calls the IDirectorOwner function inherited from its CDirectorOwner base class, passing it the `gApplication` argument as the supervisor of the director. The IDirectorOwner function, in turn, calls the IBureaucrat function inherited from its CBureaucrat base class, that, once again, sets the value of the `itsSupervisor` member variable to the value held in the `gApplication` variable (that is, the pointer to CApp). When control is returned to the IDirector function in the CDirector class, that function calls the AddDirector function of the object pointed to by the `aSupervisor` argument (the pointer to CApp), with an argument that points to the CClipboard object (`this`). This action results in the supervisor of the CClipboard object being set to the CApp object. In essence, the application is the supervisor of the clipboard.

7. The AddDirector member function is found in the CApplication base class. When called with an argument (`aDirector`) specifying a pointer to a CDirector object (CClipboard in this case), it calls the AddDirector member function of its CDirectorOwner base class, that, in turn, determines whether a list of directors currently exists (evidenced by a valid pointer in the `itsDirectors` member variable). Because the list has not been allocated at this point (see page 19), a new CDirectorList object is created and its pointer is stored into the `itsDirectors` member variable.

   The AddDirector function of the CDirectorOwner class continues by calling the Add function for the CDirectorList object to add the CClipboard object pointer (`aDirector`) to the list as its first item.

8. When the AddDirector function of the CApplication class resumes execution, it stores the pointer to the new CDirector object into the application's `lastAdded` member variable.

9. At this point, the constructor for the CAppleEventObject class is called with an argument of `FALSE`, indicating that the

object is not disposable. The CAppleEventObject constructor function sets its `disposable` member variable to the `FALSE` value, sets its `elementID` variable to the current value of the `lastElementID` variable, and then increments that variable. The new CAppleEventObject (CClipboard) is linked into the list of CAppleEventObject objects.

10. When execution of the constructor for CAppleEventObject is complete, the body of the constructor for the CClipboard object is executed. That function initializes a number of its member variables. The `itsWindow` variable is set to `NULL`, the `itsScrollPane` variable is set to `NULL`, and the `itsContents` variable is set to `NULL`. The values of the `theLength`, `theOffset`, and `lastScrapCount` variables are all set to 0. The `privateNewer` and `windowVisible` variables are set to `FALSE`. Finally, the CClipboard constructor calls its IClipboardX function to finish up the initialization of the CClipboard object.

11. The IClipboardX function allocates a new CWindow object and stores the pointer into the `itsWindow` member variable. Then it constructs a CScrollPane object, making the window its enclosure and the CClipboard object itself (`this`) its supervisor, and stores the pointer to the CScrollPane object into the `itsScrollPane` member variable. Finally, the FitToEnclFrame member function of the CScrollPane object is called to size the object to the window's default size.

12. Finally, execution resumes in the MakeHelpers function in the CApplication class. At this point, the CClipboard object is fully constructed and initialized.

Creating the CDecorator Object

The MakeHelpers function of the CApplication class continues execution by calling its MakeDecorator function. The function creates a CDecorator object whose constructor sets the object's `wCount` variable to 0, its `index` variable to 1, and then computes the width and height of the main monitor using the bounds information for the screen, storing those values into its `wWidth` and `wHeight` variables.

Setting the File Parameters

The MakeHelpers function then calls the SetUpFileParameters function that is overridden in the VA-created code in the application's x_CApp class. That code is as follows:

```
void x_CApp::SetUpFileParameters()
{
   CApplication::SetUpFileParameters();

   // File types as defined in CApp.h

   sfNumTypes = kNumFileTypes;
   sfFileTypes[0] = kFileType1;
   sfFileTypes[1] = kFileType2;
   sfFileTypes[2] = kFileType3;
   sfFileTypes[3] = kFileType4;
   gSignature = 'cApp';
}
```

As is evident in the foregoing code, the first action of the override function is to call the SetUpFileParameters function in the base class to perform its initialization tasks. In the SetUpFileParameters function of the CApplication class, the `sfNumTypes` variable is set to -1, the `sfFileTypes[0]` entry is set to `'???\?'` (this syntax defeats the use of trigraph settings), the `sfFileFilter` pointer is set to `NULL`, the `sfGetDLOGHook` pointer is set to `NULL`, the `sfGetDLOGID` variable is set to the value in the `getDlgID` variable, and the `sfGetDLOGFilter` pointer is set to `NULL`.

When the SetUpFileParameters function in the CApplication class returns, the foregoing code continues to execute. The number of file types for *this* application is specified in the `kNumFileTypes` constant. In this case, the constant has the value 1 and all of the $\texttt{kFileType}_{1...4}$ variables are defined with `'TEXT'` as their type code. The default signature code of 'cApp' is stored into the global `gSignature` variable.

Forcing Class References

The MakeHelpers function calls a member function called ForceClassReferences at this point. The purpose of the function is to ensure that all of the necessary member functions are linked into the final application. When classes or their member functions are referenced in other than a direct call, it is possible for the linker to leave them out of the final binary file. The ForceClassReferences

function in the CApplication class is empty. The expectation is that you will provide an override for this function in your derived application class.

The x_CApp derived class has an override for the ForceClassReferences function, but the CApp class, derived from the x_CApp class, also contains that function in the VA-created code. The default code in the CApp class is as follows:

```
void CApp::ForceClassReferences(void)
{
   x_CApp::ForceClassReferences();

   // Insert your own class references here
   // by calling TCL_FORCE_REFERENCE for each class
   // See x_CApp.cp
}
```

As you can see, the foregoing code merely calls the ForceClassReferences function in its base class (x_CApp), whose code for the function is as follows:

```
void x_CApp::ForceClassReferences(void)
{
   CApplication::ForceClassReferences();

                              /* From References.c       */
   ReferenceStdClasses();     /*  See template file Ref  */
}
```

As you can see in the foregoing, the first act of the code is to call the function in the CApplication class that we have already indicated is empty. This is a good practice, however, because we have no way of knowing whether it will always be empty. The next statement in the foregoing code calls the ReferenceStdClasses function. This function is contained in the **References.cp** file and the code for the function is as follows:

```
void ReferenceStdClasses(void)
{
   TCL_FORCE_REFERENCE(CArray);
   TCL_FORCE_REFERENCE(CArrayPane);
   TCL_FORCE_REFERENCE(CArrowPopupPane);
   TCL_FORCE_REFERENCE(CBitMap);
   TCL_FORCE_REFERENCE(CBitMapPane);
   TCL_FORCE_REFERENCE(CButton);
   TCL_FORCE_REFERENCE(CCheckBox);
   TCL_FORCE_REFERENCE(CDialog);
   TCL_FORCE_REFERENCE(CDialogText);
```

```
    TCL_FORCE_REFERENCE(CEditText);
    TCL_FORCE_REFERENCE(CEnvironment);
    TCL_FORCE_REFERENCE(CIconPane);
    TCL_FORCE_REFERENCE(CIntegerText);
    TCL_FORCE_REFERENCE(CPane);
    TCL_FORCE_REFERENCE(CPaneBorder);
    TCL_FORCE_REFERENCE(CPanorama);
    TCL_FORCE_REFERENCE(CPicture);
    TCL_FORCE_REFERENCE(CPopupMenu);
    TCL_FORCE_REFERENCE(CRadioControl);
    TCL_FORCE_REFERENCE(CRadioGroupPane);
    TCL_FORCE_REFERENCE(CRunArray);
    TCL_FORCE_REFERENCE(CScrollBar);
    TCL_FORCE_REFERENCE(CScrollPane);
    TCL_FORCE_REFERENCE(CStaticText);
    TCL_FORCE_REFERENCE(CStdPopupPane);
    TCL_FORCE_REFERENCE(CStyleText);
    TCL_FORCE_REFERENCE(CTable);
    TCL_FORCE_REFERENCE(CTextEnvirons);
    TCL_FORCE_REFERENCE(CWindow);

    TCL_FORCE_REFERENCE(CColorTextEnvirons);
    TCL_FORCE_REFERENCE(CIconButton);
    TCL_FORCE_REFERENCE(CLine);
    TCL_FORCE_REFERENCE(CPICTGrid);
    TCL_FORCE_REFERENCE(CPictureButton);
    TCL_FORCE_REFERENCE(CPolyButton);
    TCL_FORCE_REFERENCE(CRectOvalButton);
    TCL_FORCE_REFERENCE(CRoundRectButton);
    TCL_FORCE_REFERENCE(CSubviewDisplayer);
}
```

The foregoing code uses the TCL_FORCE_REFERENCE macro to ensure that all of the standard TCL classes that perform Object I/O are included in the resulting application. This is necessary because the user interface is established by reading the 'CVue' resources generated by the VA and constructing all of the visual elements "on the fly." You can see in the code for the CApp override of the ForceClassReferences function (see page 34) that the VA has created the override function as a placeholder for you to add references to your own classes, for the same purpose.

Setting Up the Application's Menus

The last task of the MakeHelpers function is to call the SetUpMenus member function. The x_CApp derived class overrides this function to install additional menus, but the skeleton application does not contain any of these. The code in the x_CApp class for the SetUpMenus function is as follows:

```
void x_CApp::SetUpMenus()
{
    CApplication::SetUpMenus();
}
```

Note in the foregoing that the derived class merely calls the SetUpMenus function in the CApplication base class. The code in the CApplication class for that function begins by creating a new CBartender object, calling its constructor (with a single argument of the application menu bar identifier, `MBARapp`). During the course of execution of the CBartender constructor, the function sets the `numMenus` member variable to 0, sets the `theMenus` variable to `NULL`, and sets the `choreAssigned` and forceMBarUpdate variables to `FALSE`. The CBartender constructor then calls the IBartender member function to complete the initialization process as follows:

1. The IBartender function begins execution by accessing the 'MBAR' resource for the application menu, determines the number of menus in the menu bar, and then allocates a handle of the necessary size to contain all of the menus. The function then loops through the entries in the menu bar, allocating storage for each menu entry, and initializing the fields of each entry to default values, as follows:
    a. The `dimming` field is set to `DEFAULT_DIM`.
    b. The `unchecking` field is set to `FALSE`.
    c. The `hasHMenus` (hierarchical menus) field is set to `FALSE`.
    d. The `inMenuBar` field is set to `TRUE` indicating that the menu has been installed into the menu bar.
    e. The `isHier` field is set to `FALSE`, indicating that it is not a hierarchical menu.
    f. The `useCount` field is set to 1, indicating that it is currently in use and cannot be purged.
    g. The `lastEnable` field is set to the value of bit-0 of the `enableFlags` field in the menu entry.
2. The next step in the IBartender function is to extract the command code strings from the menu bar entry for the current menu and convert these to command numbers. This is carried out by the ExtractCommandCodes function. Entries that do not contain command codes are assigned a command

number of zero. The list of command codes is stored in the `theCommands` field of the menu entry.

3. The IBartender function continues by determining whether the current menu entry contains a hierarchical menu by calling the ExtractHierMenus function. If a hierarchical menu is found, it is inserted into the menu entry by calling the AddMenu function. The menu is initialized in a fashion similar to what was just described in steps 1 and 2; if the current hierarchical menu also includes hierarchical menus, these are recursively installed by the AddMenu/ExtractHierMenus functions.

4. After all of the menus have been installed and initialized, the `'MBAR'` resource is released, DrawMenuBar is called to draw the menu bar, and the `choreAssigned` and `forceMBarUpdate` fields are set to `FALSE`.

When the IBartender function completes execution, the MakeBartender function in the CApplication class resumes execution and stores the CBartender object pointer into the `gBartender` global variable.

Running the Application

Returning to the code for the main function, shown on page 16, we see that after the IApplication function is called, the Run function is called for the application object.

The Run function, once invoked, does not return control to the main function until the `running` member variable for the application is set to `FALSE`. This means that the TCL takes complete control of your application at this point. It will operate autonomously until it needs assistance to handle a particular event, in which case it will call a function that you must override in the appropriate derived class in your source code. The Run function commences its task by executing a loop that calls the HandleForeignExceptions function continually until the `running` variable is set to `FALSE`.

The HandleForeignExceptions function establishes the top-level error handler for exceptions not thrown by the TCL. The main body of the function is a single call to the DoRun function,

within a "try" block, which has a "catch all" block that catches all exceptions and calls ErrorAlert to display an alert to the user.

The DoRun function commences execution by testing the value of the current execution "phase," which is held in the `phase` member variable. If the phase is equal to the value `appInitializing` (which is the normal case at first), then the DoRun function performs the following tasks:

1. It determines whether the CSwitchboard object has been created (this is possible if the user created the initial CApplication object using a constructor with arguments for the various memory specification parameters). If the value of the `itsSwitchboard` variable is `NULL`, then the DoRun function calls MakeHelpers, which performs all of the tasks described on pages 26–37.

2. Regardless of whether the `itsSwitchboard` value was `NULL`, the DoRun function continues execution by calling the ShowSpashScreen function. This function is empty in the implementation of the CApplication class; however, you can display a splash screen for your application by overriding this function. The VA provides the means to create splash screens, so you can use that feature to create whatever you wish to display. This is described in a later chapter.

3. The DoRun application calls the Preload function at this point. When the user selects files in the Finder and then chooses either the **Open** or **Print** command from the **File** menu, the Preload function has the responsibility for opening the files. If the **Print** command was chosen and your application provides the feature of printing its files, the Preload function is used to initiate this process.

4. If your application has been compiled to execute on other than a Power Macintosh, then the Preload function commences execution by counting the number of files supplied at the time the application was launched and then initiates a loop to process each of the files according to the "open" or "print" request. The process for each file consists of the following steps:

    a. A `MacSFReply` record is constructed to contain the file type (`fType`), volume reference number (`vRefNum`), the

version number (versNum), the "good" flag (good), and the file name (`fName`).

b. The OpenDocument function is then called to open the file, using the `MacSFReply` record just constructed.

c. The `openOrPrint` variable is tested to determine whether the file is to be printed (`appPrint`). If so, the UpdateWindows function for the `gDesktop` object is called (causing the window's contents to be drawn), and then the print command (`cmdPrint`) is sent to the object pointed to by the `gGopher` variable (the initial object in the chain of command).

d. The current file name is removed from the list of files to process and then the loop continues for the next file.

The foregoing steps are repeated for each file in the set of initial files to open or print.

5. If the application was compiled to execute on a Power Macintosh, then it is known that the target computer will receive a series of Apple Events, one for each file to be opened or printed. In this case, the Preload function does not contain the code described in the foregoing step.

6. In either case (whether Power Macintosh or not), the Preload function executes another function called StartUpAction. This function is relatively simple in the default code for the CApplication class; however, it could be overridden to perform whatever startup action you require. The default code for the StartUpAction function performs the following tasks:

   a. The FlushEvents toolbox routine is called to flush every event *except* for disk-inserted events. Any such events are kept in the event queue, to be handled later.

   b. The `hasAppleEvents`, `numPreloads`, and newWindowOnStartup variables are tested to determine whether the system has no Apple Event handling, the number of preloads is 0, and the developer wishes a new window to be opened on startup. If these conditions are met, then the StartUpAction function calls the DoCommand function, with `cmdNew` argument, for the object whose pointer is contained in the `gGopher` global variable.

7. When the preceding operations are complete, then the `running` variable is set to `TRUE` and the DoRun function continues execution.

Whether or not the `phase` variable was set to `appInitializing`, the DoRun function sets that variable to `running` at this time. This ensures that the preceding steps are executed only once, in case the DoRun function is called a second time.

The DoRun function contains two nested loops at this point in the code. The first loop executes the following statements while the `running` variable is still `TRUE`. The statements within this loop are a top-level "try" block, which has a "catch" error handler, but for exceptions that are of class CException only. These are thrown by functions in the TCL or by your source code if you use the same mechanism. The try block contains another loop, but it is a do-while loop that also executes the enclosed statements while the `running` variable is `TRUE`. Within the do-while loop is a single statement that calls the Process1Event function. In other words, the DoRun function will continue to process events until some internal or external condition occurs that either throws an exception or sets the value of the `running` variable to `FALSE`.

Processing Events

At this point, the application is initialized fully and is ready to process events initiated by the user or, perhaps, by itself or another application (in the case of received Apple Events). The Process1Event function is responsible for calling the CSwitchboard object to process one event, processing urgent chores if any exist, handling the switching in or out of desk accessory windows, and then calling the ForceNextPrepare function to force the next Prepare call to perform a full prepare.

I intend to cover event processing in depth in a subsequent chapter; however, bringing you this far, I thought it necessary to describe the basic functioning of the DoRun function, its methodology for opening or printing files (if applicable), and then the structure of the code to handle the application's event loop. In case either of the catch handlers (for CException class exceptions or foreign exceptions) is invoked, the associated code displays an appropriate alert.

Customizing the Application Skeleton

The application skeleton can be customized in various ways, depending upon what features you wish to implement. I will cover a few obvious areas where you can add custom code at the application level in the sections that follow. Other chapters discuss custom code modifications that apply to other aspects of your program code. The main areas of focus for this section are depicted in Figure 2-7.

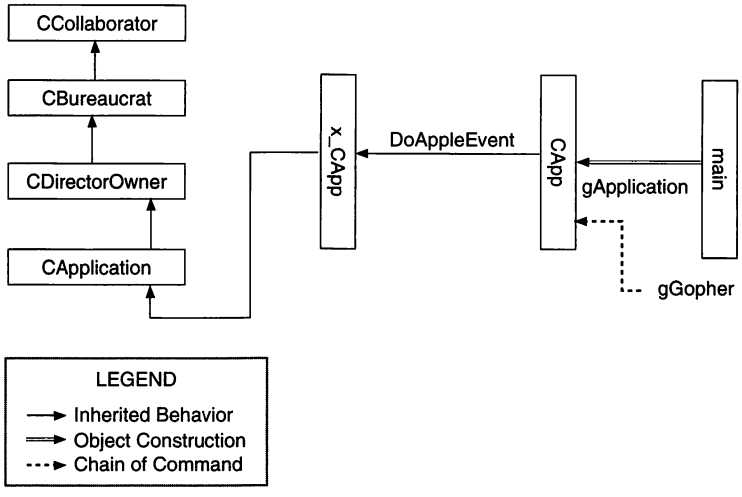

Figure 2-7
Custom changes to the application's structure

Examining the Application's Initialization Code

In the TCL, there are certain member functions that you *must* override when you create a derived class. This is the case for the x_CApp and CApp classes. The x_CApp class, consisting of the **x_CApp.cp** source file and **x_CApp.h** header file, is derived directly from the TCL's CApplication class. The SetUpFileParameters and SetUpMenus functions described previously are examples of functions that override those in the CApplication class.

The CApp class, consisting of the **CApp.cp** source file and the **CApp.h** header file, contains overrides for various functions that are either entirely empty or are only partially implemented in the CApplication and x_CApp classes. The ICApp function is one that *must* be overridden to provide application specific values for the various memory allocations described in the section titled "Initializing CApp," beginning on page 22.

The ICApp function is a very good place to put additional application-wide initialization code. If you need to load additional resources or create lists of objects that need to be referenced throughout the application, this is the place to put that code.

Modifying the Apple Event–Handling Code

If you intend for your application to handle Apple Events other than those that are built into the TCL, then you will need to install handlers for these. Before getting into the details of handlers and their installation, it is instructive to look at the list of handlers that are installed by the TCL in its InitAppleEvents member function of the CSwitchboard class.

Existing Handlers

A brief description of the InitAppleEvents function was provided in the "Creating the CSwitchboard Object," section beginning on page 27. That function calls the InstallEventHandlers member function to install Apple Event handlers for the following suites and event types:

- Handlers are installed for the *Required Suite*, including those for OpenApplication, OpenDocuments, PrintDocuments, and Quit Application events.
- Handlers are installed for the *Core Suite*, including those for Clone, Close, Count Elements, Create Element, Delete, Do Objects Exist, Get Data, Get Data Size, Move, Save, and Set Data events.
- Handlers are installed for some of the *Miscellaneous Standards Suite*, in the "Core Events" section, including those for Notify Start Recording, Notify Stop Recording, and Application Died events.
- Handlers are installed for other events in the *Miscellaneous Standards Suite*, including those for Begin Transaction, Copy, Cut, Do Script, End Transaction, Is Uniform, Paste, Redo, Revert, Transaction Terminated, and Undo events.

Events in the foregoing mentioned suites for which no handlers are installed, if sent to the application, will result in the return of an `errAEEventNotHandled` response from the AEProcessAppleEvent toolbox function.

Handling New Events

In order to handle new Apple Events, you must install a handler for each new event. This is accomplished by installing new handlers in the application's initialization code. A good place to accomplish this is in the ICApp member function of the CApp class. That code is as follows:

```
/************************************************************
 ICApp

     Initialize an Application.
************************************************************/
void CApp::ICApp()
{
   // The values below are:
   //
   //  extraMasters- The number of additional master pointer
   //                blocks to be allocated.
   //  aRainyDayFund- The total amount of reserved memory.
   //                When allocation digs into the rainy day
   //                fund, the user is notified that memory
   //                is low. Set this value to the sum of
   //                aCriticalBalance plus aToolboxBalance
   //                plus a fudge for user warning.
   //  aCriticalBalance- The part of the rainy day fund
   //                reserved for critical operations, like
   //                Save or Quit. Set this value to the
   //                memory needed for the largest possible
   //                Save plus aToolboxBalance. This memory
   //                will only be used if SetCriticalOperation()
   //                is set TRUE or if RequestMemory()/
   //                SetAllocation() is set FALSE
   //                (kAllocCantFail).
   //  aToolboxBalance- The part of the rainy day fund
   //                reserved for ToolBox bozos that bomb if
   //                a memory request fails. This memory is
   //                used unless RequestMemory()/
   //                SetAllocation() is set TRUE
   //                (kAllocCanFail). Almost all TCL memory
   //                allocation is done with kAllocCanFail,
   //                and yours should be, too. The default
   //                2K is probably enough.

   Ix_CApp(4, 24000L, 20480L, 2048L);

   // Initialize your own application data here.
   // We're installing a new Apple Event Handler.

   itsSwitchboard->InstallEventHandler (kMyAppSuite,
      kMyShowAboutBox, GenericAppHandlerUPP);
}
```

As you can see in the foregoing code, the VA has generated the call to initialize the base class (Ix_CAPP) and we have added the call to install the Apple Event handler for the "Show About Box" event.

You will want to call InstallEventHandler for each new event to be handled. If you wish to dispatch events to the application, you can refer to the global GenericAppHandlerUPP function pointer (defined in the CSwitchboard class) as the handler for your new events. The GenericAppHandler function itself is located in the CAppleEventObject class. When called, it in turn calls the GenericHandler function in that same class, passing it a pointer to a static function called DispatchApp. The GenericHandler begins execution by calling the application (via the `gApplication` global variable) object's PackageAppleEvent function. That function creates a new CAppleEvent object with the information from the current event. The event is passed to the DispatchApp handler, from which it is passed to the DoAppleEvent function associated with the application object. Don't worry about the details of all of this right now. I will cover all of the complexities of Apple Event dispatching in a later chapter.

In order to process (handle) new Apple Events, your CApp derived class should override the DoAppleEvent function inherited from the CApplication class. This is illustrated in Figure 2-7 by the DoAppleEvent label on the inheritance arrow from the CApp class to its x_CApp base class. The newly added code in the CApp class that overrides the DoAppleEvent function is as follows:

```
/************************************************************
DoAppleEvent (override)
   Overrides the DoAppleEvent function inherited from the
   CApplication class.

************************************************************/
void CApp::DoAppleEvent(CAppleEvent *theEvent, AEDesc *result)
{
   long       eventID = theEvent->GetEventID();
   Boolean    handled = TRUE;
   OSErr      err;

   switch (theEvent->GetEventClass())
   {
      case kMyAppSuite:
      {
         switch (eventID)
         {
            case kMyShowAboutBox:
            {
               gGopher->DoCommand(cmdAbout);
               break;
            }

            default:
            {
               handled = FALSE;
               break;
```

```
            }
         }
         break;
      }

      default:
      {
         handled = FALSE;
         break;
      }
   }

   if (!handled)
   {
      CApplication::DoAppleEvent(theEvent, result);
   }
}
```

The DoAppleEvent member function is the only one that we have added at this time. The code for DoAppleEvent references a new suite identifier called `kMyAppSuite`. Apple recommends placing all application-specific events into a separate suite, distinct from any of its predefined suites. I also use unique identifiers for the suite (class ID) and event type (type ID) to ensure that my choices won't conflict with future assignments by Apple. I have chosen to use the name `kMyAppSuite` to represent a suite whose code is `'Skel'` (for skeleton), and a type name of `kMyShowAboutBox` to represent a code of `'Bout'`. The code to process the event consists merely of a single statement that sends a command to the current gopher to display the "About Box" for the application (`cmdAbout`).

Because the CApp object is the current gopher, the DoCommand function in that class is first to be called. The source code for the DoCommand function is as follows:

```
void CApp::DoCommand(long theCommand)
{
   switch (theCommand)
   {
      // Insert your command cases here

      default:
         x_CApp::DoCommand(theCommand);
         break;
   }
}
```

As you can see, the foregoing code merely calls the DoCommand function in the x_CApp base class to handle every command; however, when you add code to process application-wide com-

mands, this function is where you should place that code. The code for the DoCommand function in the x_CApp base class is as follows:

```
void x_CApp::DoCommand(long theCommand)
{
   switch (theCommand)
   {
      case cmdAbout:
      {
         // Simple About alert. Subclasses will probably
         // trap this command to do something sexier.

         PositionDialog('ALRT', ALRTabout);
         InitCursor();
         ParamText("\pCApp","\pSkeleton Application","\p", "\p");
         Alert(ALRTabout, NULL);
         break;
      }
      default:
      {
         CApplication::DoCommand(theCommand);
         break;
      }
   }
}
```

The foregoing code handles the `cmdAbout` command by displaying an alert that contains the name of the application as a portion of its contents. After the user has dismissed the alert, processing of the Apple Event is complete. Apple Events, event handlers, coercion handlers, and other aspects of high-level events are discussed in detail in a subsequent chapter.

Other Application Services

The CApplication class contains many other functions that are called in the course of executing an application. Some of the important functions are as follows:

- The DoCommand function is the last in the chain of command to handle menu commands or keyboard shortcuts. The DoCommand function has the primary responsibility for performing the initial tasks in handling the following commands:
  - For the `cmdNew` command, it calls the CreateDocument function.
  - For the `cmdOpen` command, it calls the OpenDocument function.

- For the `cmdQuit` command, it calls the Quit function.

- The UpdateMenus function is responsible for enabling the appropriate menu commands just before a menu is displayed. Many other objects in the chain of command will also override this function to enable or disable various menus and their commands, as is appropriate.

- PackageAppleEvent is used to package an incoming Apple Event into a CAppleEvent object for dispatch to the appropriate handler.

- The Suspend function, by default, calls the Suspend function of CDirectorOwner and sets the `gInBackground` global variable to `TRUE`.

- The Resume function, by default, calls the Resume function of CDirectorOwner and sets the `gInBackground` global variable to `FALSE`.

- The Idle function handles periodic tasks. Idle calls the Dawdle function of the current gopher and of each of the gopher's supervisors.

- The Quit function is responsible for terminating execution of the application.

- The ChooseFile function displays a standard get file dialog and returns the SFReply pointer associated with the user's choice. This is a general purpose utility function.

- For periodic task management, the CApplication class provides AssignIdleChore to add a new chore to the list of idle chores, CancelIdleChore to remove a chore, and AssignUrgentChore to add an urgent chore to the list of urgent chores. Unlike Idle chores, urgent chores are removed from the list when they have been dispatched.

Chapter Summary

The focus of this chapter has been on the creation and initialization of the application and its related "helper" classes. The whole process of construction, initialization, and then execution of a TCL-based application commences in the global main function.

The main function creates the application object (which in our case is the CApp object) and then proceeds to execute the constructors for all of the ancestors of the CApp class, from the earliest ancestor to the CApp itself (which, by the way, doesn't have an explicit constructor function).

After the application has been constructed, the main function uses the global `gApplication` object pointer to initialize the application and all of its "helpers," including construction and initialization of the CSwitchboard (to handle events), CError (to handle errors), CDesktop (to manage windows and mouse events), CDecorator (to arrange windows on the screen), and the CBartender (to handle the menu bar). In addition, the IApp member function of the CApp class is responsible for setting the memory requirements for the application.

We have modified the ICApp function to show how a new Apple Event handler is installed into the code and have overridden the DoAppleEvents function inherited from the CApplication class so that we can process the newly installed event. Processing a new event is handled easily in the DoAppleEvents function. Events that we can't handle in our override function are passed on to the DoAppleEvents function in the CApplication base class.

When the initialization is complete, the main function uses the global `gApplication` pointer to call the Run function, which commences processing events and "running" the application. At this point in its execution, the application is fully functional; however, we have not yet opened a document and/or its associated file. The Run function does not return to the main function until the user chooses the **Quit** command or until a fatal error occurs.

If some other application (for example, AppleScript) sends the **Show About Box** command to the application, the newly added Apple Event handler will cause the DoAppleEvents override function to be invoked. This in turn, will call the DoCommand function to cause an alert to be displayed, just as if the user had chosen the **About application** command from the Apple menu.

In the next chapter, I will continue the examination of the VA-created code and look specifically at the creation of the document object and the features it provides.

Chapter 3

Creating and Managing Documents

The TCL, like most modern application frameworks, is structured according to what is called a document/view architecture. What this means is that the application object is responsible for creating one or more document objects, each of which is (normally) associated with a file and one or more views. A view is normally a window, but could be a modal or modeless dialog as well.

In this chapter, we will focus our attention on the role of the document object in the overall application. I will show how multiple document types can be created with the help of the VA and how these types manage different types of Macintosh files and the data they contain.

Creating the Default Document and Its Window

When we generated the skeleton code in Chapter 2, we generated code for a pair of document-oriented source and header files in addition to the application files we studied in that chapter. The VA generates files named **x_CMain.cp**, **x_CMain.h**, **CMain.cp**, and **CMain.h** as the base class and derived class source and header files for the default document objects.

When the application is being initialized, specifically when executing the StartUpAction function (see page 39, step 6), the code determines whether the user's system has the facilities to process Apple Events, whether the number of files pre-loaded was 0, and whether the value in the `newWindowOnStartup` variable is `TRUE` (which is the default case). If so, then the StartUpAction function calls the DoCommand function for the current gopher with a `cmdNew` command. Let's follow the execution path of that option, in which case the following steps occur:

1. Because the CApp object is the current gopher, it receives the DoCommand function call to handle the `cmdNew` command.
2. If you refer to the code for the DoCommand function in the CApp class (see page 45), you will see that it has some skeleton code, but simply passes on all of the commands to the function inherited from the x_CApp class. When you want to process application-wide commands, this is the place to add the code to do so.
3. By the same token, the code for the DoCommand function in the x_CApp class (see page 46) also passes on the `cmdNew` function, in this case to its CApplication base class.
4. The DoCommand function in the CApplication class handles the `cmdNew` command and calls the CreateDocument function to create a new document object.
5. The CreateDocument function is empty in the CApplication class and is one that our application code must override. The VA-generated code creates an override of the CreateDocument function in the x_CApp class.

Let's stop at this point and examine the code for the CreateDocument function in the x_CApp class, which is as follows:

```
void x_CApp::CreateDocument()
{
   CDocument* volatile theDocument = NULL;

   theDocument = TCL_NEW(CMain,());
   TRY
   {
      ((CMain*) theDocument)->ICMain();
      theDocument->NewFile();
   }
   CATCH
   {
      TCLForgetObject(theDocument);
   }
   ENDTRY
}
```

As you can see from the foregoing code, a new object of the CMain class is constructed, initialized, and then its NewFile function is called. Note that the `theDocument` variable is declared as "volatile." This ensures that the variable will be stored in memory, rather than in a register. This is important, should the process of initializing the object or the call to its NewFile function fail for

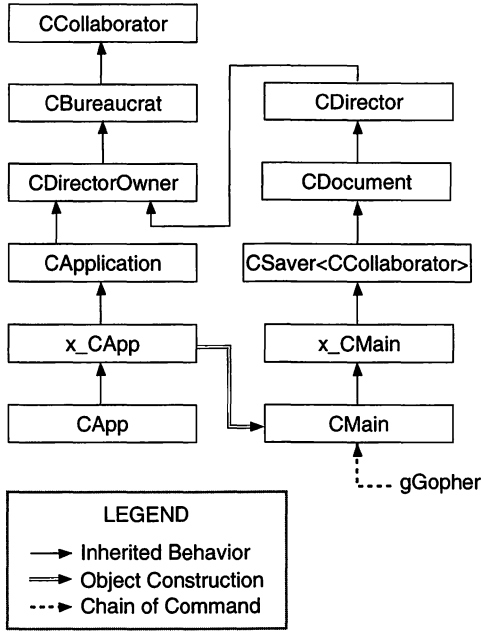

Figure 3-1
Construction of document object

any reason. For the `CATCH` exception handling code to work properly, the `theDocument` variable *must* be stored in memory.

Before examining the next series of steps, it will be instructive to refer to Figure 3-1, which shows the inheritance tree for the object that we are constructing. As you can see from the figure, I have chosen to show only the objects of interest in this case. The inheritance tree for the CApp object and also the new tree for the CMain object (and its base classes) are shown.

The construction and initialization of the document object is described in the steps that follow. Note the strange nomenclature in the case of the CSaver<CCollaborator> class. This indicates a variation of the CSaver class that operates on CCollaborator objects. It is constructed from a template (held in the **CSaver.tem** file in this case). Deriving a class from the CSaver base class enables the application to perform Object I/O, which is covered in a later chapter.

For now, let's continue the discussion of the creation and initialization of the CMain document derived class, for which the steps are as follows:

1. Referring to Figure 3-1, you can see that the first constructor to be called when the CMain object is created will be its earliest ancestor, which in this case is CCollaborator. The constructor for CCollaborator merely sets the `itsProviders` and `itsDependents` variables to the value `NULL`. The next younger ancestor in the inheritance tree is CBureaucrat, whose constructor is called with an initializer value of `gApplication`, which is supplied by its CDirectorOwner derived class. This results in the value of the `itsSupervisor` variable being set to `gApplication`. Next, the constructor for CDirectorOwner is called with the value of the `gApplication` initializer supplied by its CDirector derived class. This results in the application (CApp object) being made the supervisor of the document (CMain) object. The constructor for the CDirector class is next to execute and it performs several tasks, as follows:

   a. The `itsWindow` variable is set to `NULL`.

   b. The `active` variable is set to `FALSE`, indicating that it can't be active because there is no window yet.

   c. The `itsGopher` variable is set to `this`, indicating that when the document's window becomes inactive, the document object (CMain in this case) will become the current gopher in the chain of command.

   d. Several other variables are initialized. These include setting `activateWindOnResume` to `FALSE`, `alreadyClosing` to `FALSE`, `wasDirty` to `FALSE`, and `dirty` to `FALSE`.

   e. Finally, because the `aSupervisor` argument of the constructor is being initialized to the value held in the `gApplication` member variable, the IDirector function is called with the value of `gApplication` as its argument. The initialization sequence, commencing with the IDirector function, is described fully in Chapter 2, beginning on page 31, in steps 6–8. The end result of the foregoing steps is the installation of the document object (CMain) as a new director in the list of directors for the application object (CApp).

2. The constructor for the CDocument class begins by setting a number of member variables to the value NULL, including itsMainPane, itsFile, savePrintPane, savePrinter, askToSave, and itsPrinter. It then sets the pageWidth variable to the constant STD_PAGE_WIDTH and sets the pageHeight variable to STD_PAGE_HEIGHT. Then, because the definition of the constructor has a default argument of TRUE for printable, the constructor calls the SetupPrinter function, which creates a CPrinter object and stores its pointer into the itsPrinter variable.

3. The default constructor for the CSaver class merely sets the itsContents variable to NULL, indicating that there are no data currently associated with the document.

4. Neither the x_CMain nor CMain classes has a constructor function, so the construction of the document object is complete at this point, after the default constructors for those classes are executed.

5. When construction of the CMain object is complete, the foregoing CreateDocument function (see page 50) continues execution by calling the ICMain function of the object to initialize its contents. The code for the ICMain function in the VA-generated code is as follows:

```
void CMain::ICMain()
{
   Ix_CMain();

   // Initialize data members here
}
```

As is evident from the foregoing code, the function merely calls the Ix_CMain function in its base class. In addition, the VA has placed a comment into the code that indicates the place to add your own initialization code for the object. The code for the Ix_CMain function is as follows:

```
void x_CMain::Ix_CMain()
{
   IDocument(gApplication, TRUE);

   // Initialize data members below.
}
```

The foregoing code is written into the **x_CMain.cp** file by the VA and should not be modified manually. The VA may write additional code into this function in the future. And in any case, the file will be regenerated each time that you call upon the VA to generate code.

In the foregoing, the IDocument function is called with arguments of `gApplication` and `TRUE`. These correspond to the formal arguments of `itsSupervisor` and `printable` in the IDocument function of the CDocument class. Note that in calling the IDocument function, we have skipped over any potential initialization in the CSaver class.

6. When the IDocument function is called with the `gApplication` and `TRUE` arguments, the function calls the SetupPrinter function, which creates a new CPrinter object and stores its pointer into the `itsPrinter` variable for the document object. Some additional printer-related initialization is performed and the IDocument function returns.

The foregoing steps complete the initialization process for the document object. At this point, the CreateDocument function (refer to page 50) calls the NewFile function for the newly created document object.

It is worthwhile to note at this point that the NewFile function does not create a new file, but leaves the `itsFile` member variable for the document set to `NULL`. A file will be created only when (or if) the user performs a **Save** or **Save As** command. The primary function of the **NewFile** command is to create a new window that is populated with whatever contents were specified in the VA for the (default) Main window.

The NewFile function exists in the CSaver class, which is a template class that inherits behavior from the CDocument class and handles objects of the CCollaborator class by default. The NewFile function in the CSaver class performs the following steps:

1. The MakeNewWindow function is called. This function exists in the CDocument class, but is overridden by the VA-generated code in the x_CMain class. The code for the override of the MakeNewWindow function is as follows:

```
void x_CMain::MakeNewWindow(void)
{
   itsWindow = TCLGetNamedWindow("\pMain", this);
   itsMainPane = (CPane*) TCLGetItemPointer(itsWindow, 0);
   // Initialize pointers to the subpanes in the window
   fMain_Pict1 = (CPicture*) itsWindow->FindViewByID(kMain_Pict1ID);
  ASSERT(member(fMain_Pict1, CPicture));

  fMain_Stat2 = (CStaticText*) itsWindow->FindViewByID(kMain_Stat2ID);
   ASSERT(member(fMain_Stat2, CStaticText));
}
```

The foregoing code uses the facilities of the TCL to access the 'CVue' resource whose name is `Main`, to get the location and bounds of the window to be created. The window is created using the TCLGetNamedWindow function (located in the **ViewUtilities.cp** source file) and is assigned the current document object (`this`) as its supervisor in the chain of command. The GetNamedWindow function is described in a later chapter that covers Object I/O. It is important to point out that the functions in the **ViewUtilities.cp** source file are used only if windows and views are being constructed using the TCL's facilities for Object I/O.

The MakeNewWindow function continues by calling the TCLGetItemPointer function (in **ViewUtilities.cp**) to access the pointer to the CPanorama object associated with the window constructed by the VA. This pointer is stored into the `itsMainPane` variable, which is used by various functions in the application.

2. MakeNewWindow continues by calling the FindViewByID function for the CWindow object, using the value of the `kMain_Pict1ID` constant as the identifier of the view to be located. The values associated with the various mnemonic constants for the Main window are found in the **MainItems.h** header file and are defined as follows:

```
enum
{
   Main_Begin_,
   kMain_Pict1 = 1,
   kMain_Pict1ID = 1L,
   kMain_Stat2 = 2,
   kMain_Stat2ID = 2L,
   Main_End_
};
```

The VA assigns a value to the `ID` variable of each view and the foregoing enumeration indicates that the FindViewByID function will be used to search for a view that has an ID of 1 in the case of the `kMain_Pict1ID` constant. The FindViewByID function is located in the CView class of the TCL. That function calls an iterator function called MatchView that searches for a subview associated with the current window, whose ID matches what is being sought. In this case it is the CPicture view, illustrated in black and white in Figure 2-2, except for the words "Hello, world!" in that figure.

The pointer to the CPicture variable is stored into the `fMain_Pict1` member variable for the document and then is verified to be of the CPicture class.

3. The MakeNewWindow function then searches for the subview whose view-ID corresponds to the kMain_Stat2ID constant, stores the pointer into the fMain_Stat2 member variable for the document, and then verifies that the pointer is of the CStaticText class.

At this point, the construction of the window is complete; however, the MakeNewWindow function returns to the NewFile function in the CSaver class and that function continues execution by calling the MakeNewContents function. That function is intended to populate the window with new contents that are unique to the current application. Therefore, the MakeNewContents function is overridden in the CMain derived class, in the default VA-generated code. The code is as follows:

```
void CMain::MakeNewContents()
{
   // Initialize document contents and itsWindow here
}
```

Although the foregoing function is empty, the VA has indicated that this is the appropriate place to initialize the document's contents and also to perform any initialization that is appropriate to the window, by accessing the `itsWindow` member variable. The NewFile function continues by calling the ContentsToWindow function, which is also overridden in the VA-generated code for CMain. That code is as follows:

```
void CMain::ContentsToWindow()
{
   // Transfer data from itsContents to itsWindow.
   // See Chapter 8, Using Object I/O
}
```

The foregoing code for the ContentsToWindow function is also empty; however, the comments indicate that this function is the place to put custom code to transfer the contents of the document's data into the associated window. At this point, there is no data to transfer (as is normally the case with a new window).

The NewFile function continues execution by calling the PositionWindow function to position the window on the screen and then calling the MakeWindowName function to determine the appropriate name for the window in its title bar. New windows are always named with the document's name (for example, Main), with the window number (that is, 1) appended to create a window title of "Main 1" for the first new window, "Main 2" for the second new window, and so forth. After the window name has been concocted, it is written into the window's title bar.

The final task of the NewFile function is to call the SetChanged function with an argument of `FALSE`, indicating that the document has not been changed (that is, it is not "dirty" at this point). Following that, the Select function is called for the `itsWindow` pointer, which activates the newly constructed window.

Creating the Document with an Open Application Event

I began the previous section by making the assumption that the user's Macintosh was not capable of handling Apple Events. In this section, I'll cover the differences between what was presumed earlier and how the application behaves when Apple Events are handled by the user's machine.

In the first place, if Apple Event support is present, the StartUpAction function in the CApplication class (see page 39) does not call the DoCommand function for the current gopher with the `cmdNew` command. Instead, after the application has been initialized and is in a stable state, commencing to process events, it receives an Open Application event from the system.

When the Open Application event is received by the application, it calls the GenericAppHandler function in the CAppleEventObject class to handle the event. That handler calls the GenericHandler function to create and package the Apple Event, and then the DoAppleEvent function associated with the object pointed to by the `gApplication` variable is called (CApp in this case). As you may recall, the DoAppleEvent code in our CApp class handles only the new Show About Box event and passes all other events to the DoAppleEvent function in the CApplication class to handle. That function does handle the `kAEOpenApplication` event by first checking whether the value of the `newWindowOnStartup` variable is `TRUE`, and if so, it calls the application's DoCommand function with an argument of `cmdNew`. This brings us to the same point in the execution logic as described previously, beginning with step 1 on page 50.

If you choose *not* to open a new window when the document object is created, then you can put the following statement into the ICApp function in your application:

```
newWindowOnStartup = FALSE; // prevent creation of a new window
```

If the typical use of your application is to input a data file and display its contents, then you may wish for the creation of a new window to be an explicit function by the user's choice of the **New** command from the **File** menu and, instead, launch the application with no default window present on start-up.

Managing the Document's Data

The document object is the data server for most applications. It is responsible for initiating the reading and writing of data files and is also the manager for the document's data in most cases. A single document object is created, by default, each time the user chooses the **New** or **Open** command from the **File** menu. In the case of the **New** command, no file is associated with the document object initially. However, if the user chooses either the **Save** or **Save As** command, he or she will be prompted with a Standard File dialog to specify a path and file name for the data to be saved.

A given TCL-based application can arrange to handle one or more files and their data in various ways. The following sections illus-

trate common application structures, with regard to the individual document objects and file types supported.

Handling a Single File Type

The most common application structure is one that supports a single file type (for example, a 'TEXT' file) and one or more instances of the document object that handles data for that file type, depending upon how many such files are open concurrently.

When the user chooses the **Open** command from the **File** menu, the DoCommand function in the CApplication class handles the command by calling the ChooseFile function (which prompts the user for which file to open) and then the OpenDocument function is called, the code for which is as follows:

```
void x_CApp::OpenDocument(SFReply *macSFReply)
{
   CDocument*volatile theDocument = NULL;

   theDocument = TCL_NEW(CMain,());
   TRY
   {
      ((CMain*) theDocument)->ICMain();
      theDocument->OpenFile(macSFReply);
   }
   CATCH
   {
      TCLForgetObject(theDocument);
   }
   ENDTRY
}
```

The code creates a new document object (CMain in this case), initializes the object, and then calls the document's OpenFile function. Neither the x_CApp nor CApp derived classes contain overrides for the OpenFile function. In this case, that function is contained in the CSaver class (in the file **CSaver.tem**, the template file from which the CSaver<CCollaborator> object is constructed). The code for the OpenFile function is as follows:

```
template<class T>
void CSaver<T>::OpenFile(SFReply*macSFReply)
{
   CDataFile  *theFile = NULL;
   FSSpec     spec;

   theFile = TCL_NEW(CDataFile,()); // Try to open this file

   try_
   {
```

```
      theFile->IDataFile();
      theFile->SFSpecify(macSFReply);
      theFile->GetFSSpec(&spec);
      FailOpen(&spec);       // Fail if open in this application
      theFile->Open(fsRdWrPerm); // Fail if open in another app
                                 // (and lots of other reasons)
      itsFile = theFile;         // Promote to document file
   }
   catch_all_()
   {
      TCLForgetObject(theFile);
      throw_same_();
   }
   end_try_

   ReadDocument(); // Read document from file
}
```

As is evident in the foregoing code, the OpenFile function creates a CDataFile object, initializes the object, and then uses the information in the `macSFReply` variable to resolve the file specification and call the Open function to open the file.

Assuming that the Open operation is successful, the OpenFile function concludes by calling the ReadDocument function. This function is also contained in the CSaver class and its code is as follows:

```
template<class T>
void CSaver<T>::ReadDocument()
{
   CFileStream   *theStream = NULL;
   Str255        theName;

   // Create a file stream
   theStream = NewInputFileStream((CDataFile*) itsFile);

   try_
   {
      ReadContents(theStream);   // Read contents of document
      SetChanged(FALSE);         // Not dirty any longer
      delete theStream;          // Get rid of stream
   }
   catch_all_()
   {
      delete theStream;
      throw_same_();
   }
   end_try_

   if (itsWindow == NULL)        // If window wasn't created by
      MakeNewWindow();           //  ReadContents, make it now
   ContentsToWindow();           // Make contents displayable

   PositionWindow();             // Must do before making name
   MakeWindowName(theName);      // Change the window title
   itsWindow->SetTitle(theName);
   itsWindow->Select();          // Activate the window
}
```

As is evident in the foregoing code, the ReadDocument function creates a new input file stream to access the user-specified file and then calls the ReadContents function to read the contents of the file into memory. The process of reading the file involves the special Object I/O features of the TCL that I will be describing in a later chapter; however, it is fair to state that the contents of the file will reside in memory after the ReadContents operation is complete. After this, the document's SetChanged function is called to ensure that the document's contents is not "dirty" at this point (having been newly read) and then the stream used to input the file's contents is deleted.

The ReadContents function continues execution by testing whether the document's `itsWindow` variable has a `NULL` value. This is the case for a user-specified file, so the function calls the MakeNewWindow function to create the document's window (or windows). The MakeNewWindow function is overridden in the derived x_CMain class and its code is as follows:

```
void x_CMain::MakeNewWindow(void)
{
   itsWindow = TCLGetNamedWindow("\pMain", this);

   itsMainPane = (CPane*) TCLGetItemPointer(itsWindow, 0);

   // Initialize pointers to the subpanes in the window

  fMain_Pict1 = (CPicture*) itsWindow->FindViewByID(kMain_Pict1ID);
  ASSERT(member(fMain_Pict1, CPicture));

  fMain_Stat2 = (CStaticText*) itsWindow->FindViewByID(kMain_Stat2ID);
  ASSERT(member(fMain_Stat2, CStaticText));
}
```

As you can see in the foregoing code, the MakeNewWindow function calls TCLGetNamedWindow to construct the window whose name is "Main," as specified to the VA when the view was created. This is the default name for the main window in the skeleton project. You can change the name to whatever you wish and also specify additional windows within the VA that will also be created in the foregoing function, if they are specified. (The TCLGetNamedWindow function is contained in the **ViewUtilities.cp** source file.)

Returning to our examination of the MakeNewWindow function, we see that after the window has been created and its pointer has

been stored into the `itsWindow` member variable for the document object, the code calls the TCLGetItemPointer function (also contained in the **ViewUtilities.cp** file) to access item 0 for the newly created window. This item is the CPanorama object for the window (if any). The pointer to the panorama (or `NULL`) is stored into the `itsMainPane` member variable of the document object.

The code continues by creating objects for each of the panes in the view and storing their pointers into corresponding member variables of the document object. This practice allows you to write code in your document subclass that refers directly to the visual elements of the document's window. When this step is complete, the application has the structure shown in Figure 3-2.

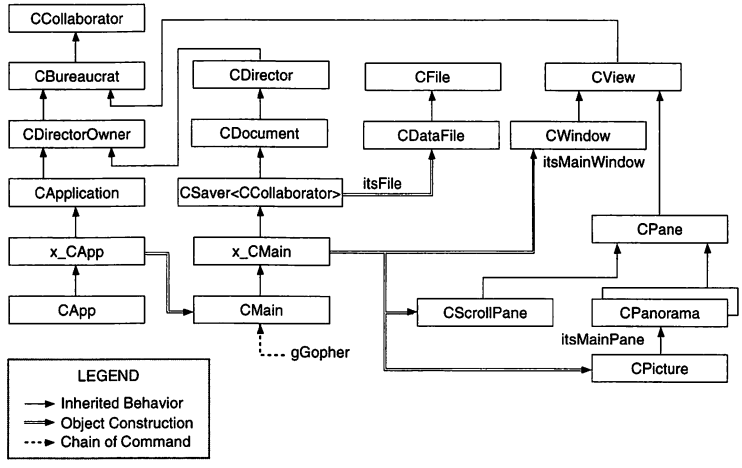

Figure 3-2
Application structure after the document's file and views have been created

After the window and all of its subviews have been created, the ReadDocument function of the CSaver class continues by calling the ContentsToWindow function to display the data read from the user-specified file in the window. The ContentsToWindow function is overridden in the VA-generated code for the CMain derived class and is as follows:

```
void CMain::ContentsToWindow()
{
   // Transfer data from itsContents to itsWindow.
   // See Chapter 8, Using Object I/O
}
```

As is evident, the VA has generated this function as a placeholder in which you can add the necessary code to perform the specified operation. I will be covering the whole topic of Object I/O in a later chapter.

After the contents have been added to the window, the ReadDocument function continues by calling the PositionWindow function. This function calls the CDecorator object (by means of the `gDecorator` global variable) to stagger the window with respect to other windows on the desktop for this application. Although the default-generated code doesn't do so, you can override this function to place the window wherever you wish.

After the window has been placed into position on the desktop, the ReadDocument function sets the window's title to the name of the associated file and a sequence number (by default) by calling the MakeWindowName and SetTitle functions. The ReadDocument function concludes execution by selecting the current window, making it the frontmost window on the desktop.

When the execution of ReadDocument is complete, the OpenFile function also concludes execution, returning control to the OpenDocument function in x_CApp (shown on page 59), and concluding execution of the **Open** command.

Handling Multiple File Types

If your application is able to open files of various types (for example both `'TEXT'` and `'PICT'` files), then you should create a separate document class for each file type. This is easily done using the VA. Double-click on the **Visual Architect.rsrc** file in your project window to launch the VA, and then choose **New View** from the VA's **View** menu. This will cause the VA to display a dialog that requests a name for the view and also its type. The type of a main document view is "Main Window," as shown in Figure 3-3, and as chosen from the dialog's pop-up menu.

Click OK in the dialog to dismiss it, and the VA will show a default view on the screen. At this point, you may want to change the characteristics of the view, in which case you should choose **View Info** from the **View** menu. This will cause VA to display a view information dialog.

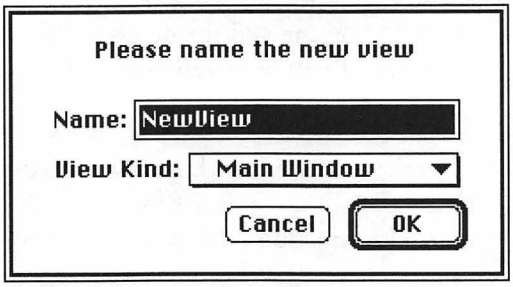

Figure 3-3
Visual Architect's New View dialog

I chose for the new view to have both a close box and a zoom box, so I changed the window type. In addition, I checked the Size Box checkbox to ensure that the new view can be resized and also checked the "Vert Scroll" checkbox to add a vertical scroll bar. Finally, I unchecked the Use File checkbox so that the CSaver-oriented files are not created for this document and its view. The final appearance of the View Info dialog is as shown in Figure 3-4.

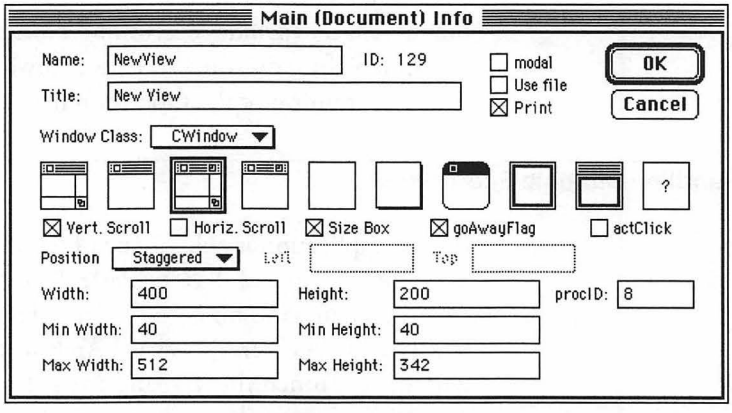

Figure 3-4
Visual Architect's View Info dialog

Dismiss the View Info dialog to save the selections we made. To see what the view will look like when it is created, choose the Try Out option. It will appear as shown in Figure 3-5.

Notice that the new view has the title we specified (New View) and that it has the close, zoom, and size boxes. Also note that it has only a vertical scroll bar. If we were going to use this view to

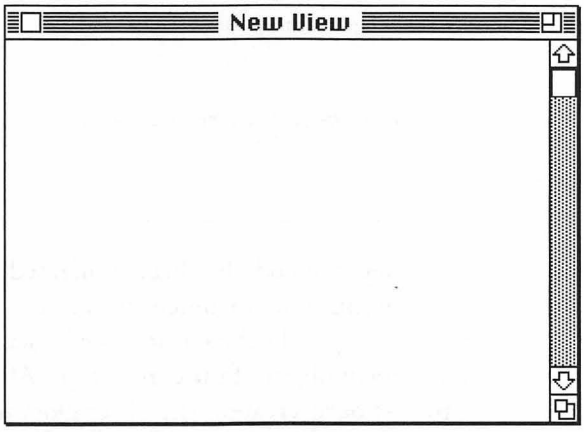

Figure 3-5
Appearance of the New View in the Visual Architect

display a particular type of data (for example, a 'PICT' file), we should create a CPicture or other subpane derived from CPanorama in which to display the contents of the 'PICT' file or resource. I have not added any subviews at this point.

After the new view has been specified, you will need to choose the **Generate** command from VA's **Project** menu (the menu immediately to the right of its **Windows** menu). If this is the first time that you are generating code for the project, then choose **Generate All** from the **Project** menu.

Opening a Document

After the code has been generated, open the **x_CApp.cp** file and examine the OpenDocument override function. You will see that the generated code is quite different from what was shown for that function on page 59. The earlier example corresponds to the code that VA generates for a single file document. The newly generated code for that function is as follows:

```
void x_CApp::OpenDocument(SFReply *macSFReply)
{
   CDocument  *volatile theDocument = NULL;
   FInfo      info;

   FailOSErr( GetFInfo(macSFReply->fName,
      macSFReply->vRefNum, &info) );

   TRY
   {
      theDocument = CreateTypedDocument(info.fdType);
```

```
      if (theDocument != NULL)
         theDocument->OpenFile(macSFReply);
   }
   CATCH
   {
      TCLForgetObject(theDocument);
   }
   ENDTRY
}
```

The foregoing code has been generated to take the prospect of opening multiple document files into account. Notice that the function now calls the CreateTypedDocument function to create a document object of the correct type. After the proper document type has been created, the OpenDocument function calls the OpenFile function, which is unchanged from what was presented on page 59. The code for the CreateTypedDocument function is as follows:

```
CDocument *x_CApp::CreateTypedDocument(OSType filetype)
{
   CDocument  *theDocument;

   if (filetype == CMainFType)
   {
      theDocument = TCL_NEW(CMain,());
      ((CMain*) theDocument)->ICMain();
   }
   else if (filetype == CNewViewFType)
   {
      theDocument = TCL_NEW(CNewView,());
      ((CNewView*) theDocument)->ICNewView();
   }
   else
   {
      ASSERT(!"\pNo document of the argument type");
      theDocument = NULL;
   }

   return theDocument;
}
```

The foregoing code determines whether the file that the user has just opened (via the ChooseFile call in the DoCommand function in CApplication) matches the `CMainType` or `CNewViewType` constants. These constants are defined in the header files of the most derived class for the corresponding view (that is, **CMain.h** and **CNewView.h**). You will have to edit the definitions in these files to correspond to the actual file types you expect to open. They are set to type `'TEXT'` by default.

After the appropriate file type match has been made, the CreateTypedDocument function creates and initializes the appropriate document class. The ICMain and ICNewView initialization functions are empty; however, you can add whatever code is necessary to perform additional initialization, after the statement that calls the initialization function for the immediate base class. For example, the code for the ICNewView functions is as follows:

```
void CNewView::ICNewView()
{
   Ix_CNewView();

   // Initialize data members here
}
```

And the code for the Ix_CNewView base class is as follows:

```
void x_CNewView::Ix_CNewView()
{
   IDocument(gApplication, TRUE);

   // Initialize data members below.
}
```

The IDocument function creates and initializes a CPrinter object for the document if the second argument is `TRUE` (as is the case in the foregoing code). If you had chosen to uncheck the "Print" checkbox in the View Info dialog (shown in Figure 3-4), then the VA would have generated a `FALSE` value for the `printable` argument of the IDocument function.

After the CreateTypedDocument function returns, the OpenDocument function continues execution by calling the OpenFile function for the selected document object. Because the "Use File" checkbox in the View Info dialog for the new view is unchecked, there is no generated code in either the CNewView or x_CNewView classes that overrides the OpenFile function in the CDocument class (which is an empty function). So to implement opening the specified file, you will have to add an override for the OpenFile function to the CNewView derived class in order to open and read the file's contents. The "Use File" checkbox, when checked, instructs the VA to generate code that references the Object I/O features of the CSaver class. When the checkbox is not checked, then it is up to you to write the necessary code to open

and read the specified data file. Instead of using the Object I/O features, you can derive your document class from the CSimpleSaver class (as described in the chapter concerning Object I/O) and benefit from the great deal of support it provides for reading and writing files with arbitrary formats.

Don't conclude from this that simply checking the "Use File" checkbox will solve your problems. If you do so, you will have to make use of a file that is written in a very specific format. The Object I/O facilities cannot handle plain text or other standard files. Only files that were written previously using the Object I/O facilities are suitable for input with the facilities of the CSaver class. I will cover the Object I/O facilities in a later chapter.

Creating a New Document

When the user chooses the **New** command from the **File** menu, the newly generated code must determine what type of file (that is, document type) should be created. Because the VA has no way of knowing how you might wish to specify the appropriate file type, it leaves much of the selection process up to your program code. The code for the CreateDocument function is as follows:

```
void x_CApp::CreateDocument()
{
   CDocument  *volatile theDocument = NULL;
   OSType     doctype;

   TRY
   {
      if (DoNewDialog(&doctype))
      {
         theDocument = CreateTypedDocument(doctype);
         if (theDocument != NULL)
            theDocument->NewFile();
      }
   }
   CATCH
   {
      TCLForgetObject(theDocument);
   }
   ENDTRY
}
```

The foregoing code commences by calling a function called DoNewDialog, which is intended to store the file type to create into the `doctype` argument and return `TRUE` to indicate that a file type was chosen, or `FALSE` if not. The default code for the

DoNewDialog is generated into the **x_CApp.cp** source file and is as follows:

```
Boolean x_CApp::DoNewDialog(OSType *filetype)
{
   Boolean haveType = FALSE;

   return haveType;
}
```

As you can see in the foregoing code, the function simply returns a `FALSE` result. You will have to override this function in your CApp derived class to provide the results that you desire. If, for example, you wish for the **New** command to create only a document of the `CMainFType` type, then your override function could be written as follows:

```
Boolean CApp::DoNewDialog (OSType *filetype)
{
   filetype = CMainFType;
   return TRUE;
}
```

This would cause the CreateDocument function to call the CreateTypedDocument function with the `CMainFType` argument. The code for that function has been shown previously (see page 66). If, instead of limiting the creation to the single document type, you wish for the user to be able to create either type of document, then your DoNewDialog override function should probably open a dialog that specifies a choice between the two file types.

Creating a File Type Dialog

Let's assume that you wish to create such a dialog to offer the user the choice between the two supported file types. This is easily accomplished by using the VA to design the dialog, generating new code, and then writing a new version of the DoNewDialog function. The steps for implementing this concept are as follows:

1. The first step in this process is to launch the VA by double-clicking on the **Visual Architect.rsrc** file name in the project window.

2. After the VA has been launched, pull down the **View** menu and choose the **New View** command. You will be presented with a dialog similar to what is shown in Figure 3-3. Enter "NewFile" for the view name and choose Dialog as the view type from the pop-up menu, as shown in Figure 3-6.

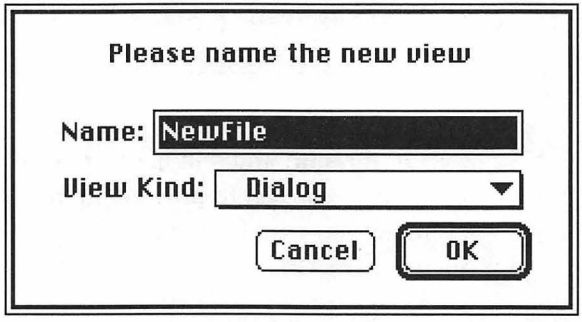

Figure 3-6
Creating a New File dialog view

3. After dismissing the dialog shown in the foregoing figure, the VA will display a new view. Choose **View Info** from the **View** menu and make the entries and change the settings to match what is shown in Figure 3-7.

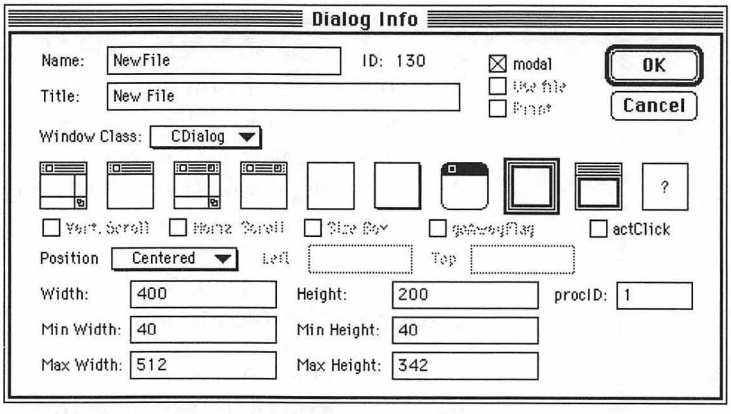

Figure 3-7
View Info settings for the NewFile view

4. Note in Figure 3-7 that we have selected the window type with the double border (third from the right in the pictures of the window types). We have assigned the name **NewFile** to the view, have given it a title of **New Name**, and have checked the **modal** checkbox, making it a modal dialog.

5. After making the foregoing selections in the **View Info** dialog, we clicked OK to dismiss the dialog and then added some simple controls to the view. The appearance of the completed dialog, as it appears in the VA, is shown in Figure 3-8.

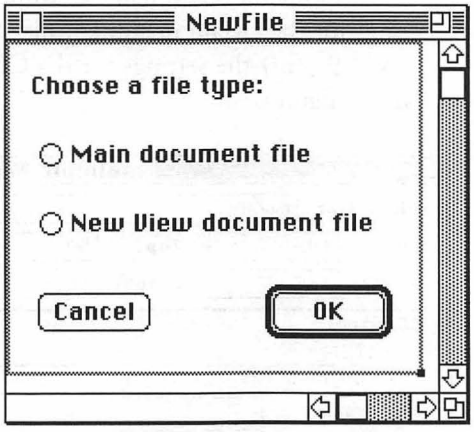

Figure 3-8
Completed NewFile dialog view

The creation of the view shown in Figure 3-8 is relatively simple and was accomplished as follows:

a. Choose the Static text tool from the Tools palette (the one that has the "A" icon), and type in the "Choose a file type:" text string.

b. Choose the Radio button tool from the Tools palette, and create the two radio buttons shown in the figure.

c. Choose the Push button tool from the Tools palette, and create first the OK and then the Cancel buttons by simply clicking the mouse in the positions shown in the figure.The first button you create will be created as a default OK button, the second will be created as a Cancel button, all automatically.

d. Resize the ScrollPane that encloses the controls to be just large enough to encompass the contents of the dialog.

6. After the dialog has been created, you will need to change the settings for the radio buttons so that the one corresponding to

the "Main document file" caption is turned on and the one corresponding to the "New View document file" caption is turned off. These changes are made to the parameters for the CControl class and are accessible by either double-clicking on the control or by selecting the control and choosing the **Pane Info** command (Command-L) from the **Pane** menu. The settings for the "Main document file" button are shown in Figure 3-9 (only the settings for the CControl class are pictured in the figure).

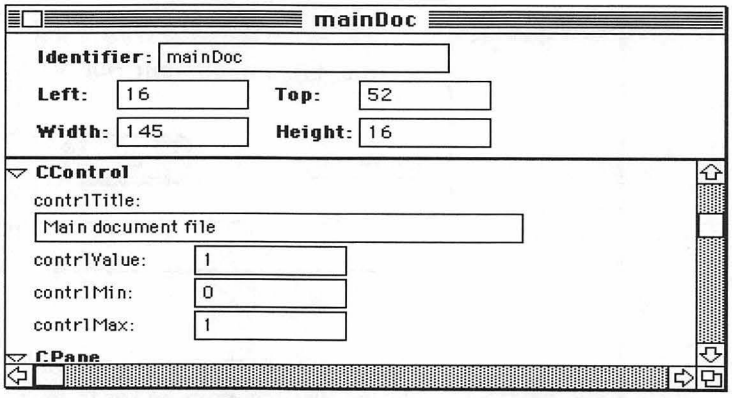

Figure 3-9
Settings for mainDoc radio button in NewFile dialog view

Figure 3-9 shows the results of making two changes to the settings for the "Main document file" radio button control. The **Identifier** has been changed to **mainDoc**, and then the triangle control next to the CControl class has been clicked to open up its settings, changing the **controlValue** setting to 1, instead of the default value of 0. After those changes have been made, the **Pane Info** window can be closed by clicking in its close box.

7. The settings for the "New View document file" radio button are changed in a similar fashion to those for the "Main document file" radio button and are shown in Figure 3-10. We have changed the **Identifier** to **newViewDoc** and have left the **controlValue** setting as 0 (the default value). After the change to the identifier has been made, the **Pane Info** window can be closed by clicking in its close box.

This completes the steps for creating and modifying the **NewFile** dialog view. If you choose the **Try Out** command from the **View** menu, you will see a dialog pictured in Figure 3-11.

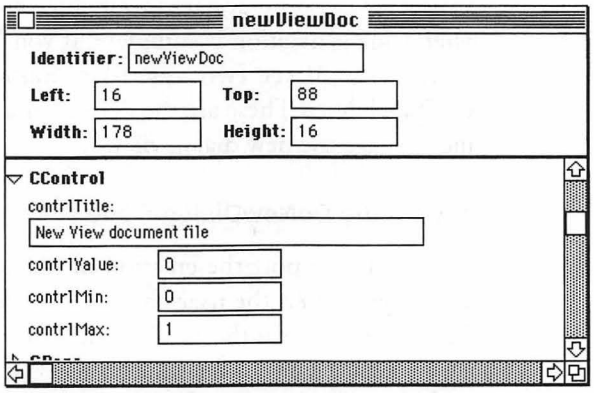

Figure 3-10
Settings for NewView radio button in New File dialog view

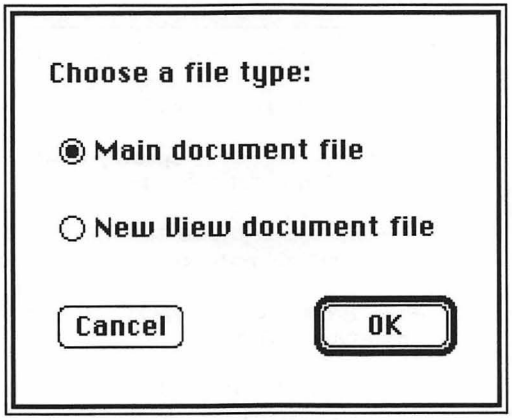

Figure 3-11
Final appearance of NewFile dialog within the VA

When you have finished testing the dialog shown in Figure 3-11, click either the OK or Cancel button to close the dialog.

Generating the Dialog Code

At this point, we have done everything necessary in VA to create the new dialog, so the next step is to generate code to support its creation and management. Pull down the **Project** menu in the VA, and choose the **Generate** command. This will cause the VA to generate the new files and modified versions of existing files to manage the new dialog view. The Symantec Project Manager

project will also be updated to include the newly generated files when code generation is complete. If you look in the project window, you will see two new files, named **CNewFile.cp** and **x_CNewFile.cp**. These are the derived and base class files to create and manage the new dialog view.

Adding the DoNewDialog Code

In order to support the creation of the appropriate view (document) type when the user chooses the **New** command from the **File** menu (or when the application starts up), the existing code in the CreateDocument function in the x_CApp class (see page 68) calls the DoNewDialog function to ascertain what type of document object to create. I have written an override DoNewDialog function in the CApp class whose code is as follows:

```
Boolean CApp::DoNewDialog(OSType *filetype)
{
   Boolean haveType = FALSE;

   *filetype = 0;     // specify a NULL file type
   itsNewDialog = TCL_NEW(CNewFile, ());  // create dialog
   itsNewDialog->ICNewFile(this);         // initialize dialog
   itsNewDialog->BeginModalDialog();
   if (itsNewDialog->DoModalDialog (cmdNull) == cmdOK)
   {
      *filetype = GetDocTypeFromDialog (itsNewDialog);
      haveType = TRUE;
   }
   ForgetObject (itsNewDialog);

   return haveType;
}
```

The foregoing code is not much more complex than what was presented earlier (see page 69) to handle the creation of a single document type. In the case of the foregoing code, I created the CNewFile dialog object and then invoked it to allow the user the opportunity to specify what document type should be created when the **New** command is chosen.

In addition to the source code for the DoNewDialog function, I added a declaration of the function and one new private member variable declaration into the **CApp.h** header file. These are as follows:

```
   virtual Boolean DoNewDialog(OSType *filetype);
private:
   CNewFile   *itsNewDialog;  // newly added
```

The DoNewDialog code is relatively simple. It creates a new CNewFile object, calls its INewFile initialization function, calls the BeginModalDialog function for that object, and then calls the DoModalDialog function (which actually "runs" the dialog). When DoModalDialog returns, the new code tests whether the function returned a value equal to the `cmdOK` constant. If so, then the code calls another function called GetDocTypeFromDialog to access the dialog object to extract the user's choice of document type. The type is stored into the `fileType` variable and the value of the `haveType` variable is set to `TRUE`. This indicates that a type was chosen (that is, the user didn't cancel the dialog). If the return value from the DoModalDialog function is not equal to the value of the `cmdOK` constant, then the user must have cancelled the dialog and the `haveType` variable will remain set to the `FALSE` value with which it was initialized prior to running the dialog. In either case, the code calls the ForgetObject function to delete the CNewFile object and then returns the value of the `haveType` variable (indicating success or failure—the `fileType` variable is passed back to the caller to use when a selection has been made).

Modifying the GetDocTypeFromDialog Code

When the VA generated code for the CApp class originally, it created the GetDocTypeFromDialog function, which was unused in the initial single document implementation of our skeleton application. Because the current version of the application requires the choice between various file types when a new file is created, we will use this function to access the selected file type from the CNewFile dialog director object. The modified code for the GetDocTypeFromDialog function is as follows:

```
OSType CApp::GetDocTypeFromDialog(CNewFile *dialog)
{
   if (dialog->endData.fNewFile_mainDoc == 1)
   {
      return CMainFType;
   }
   else if (dialog->endData.fNewFile_newViewDoc == 1)
```

```
    {
        return CNewViewFType;
    }
    else
    {
        return 0;
    }
}
```

The foregoing code accesses a data structure called `endData` from the `dialog` argument to the function and tests whether the `fNewFile_mainDoc` or `fNewFile_newViewDoc` field has been set to 1 (indicating that the associated radio button was clicked). Only one of the fields can have a value of 1. The function returns the file type (`CMainFType`) or (`CNewViewFType`) corresponding to the selected radio button. Because the default-generated code specifies a file type of `'TEXT'` for both file types, I changed the code for the `CMainFType` constant to `'Savr'`, indicating that the main document uses the CSaver class (Object I/O features) to read and write its data. This change is made to the **CMain.h** header file and is as follows:

```
// If you have multiple document classes, you must change
// the file type below to the appropriate type for this class.
// If not, this #define is not used.

#define CMainFType 'Savr'
```

Note in the foregoing code that the VA requests a definition that is appropriate to the class. (Although the header file for the CNewView class has a similar definition, I have not modified the default-generated definition of `'TEXT'` for that document type.)

Modifying the CNewFile Dialog Code

In order for the CNewFile dialog to operate properly, it's necessary to add some code to the **CNewFile.cp** derived class source file. The number of changes is minimal, and I will not attempt to explain the intricacies of the dialog's operation at this point.

I will explain dialogs in depth in a later chapter. The first change we made was to the **CNewFile.h** header file, in which we added a new instance of the CNewFileData structure member variable, as follows:

```
// Insert your own public data members here

CNewFileData endData;
```

The endData structure gives us a place to store the final results of the user's selections in the dialog. The only other change made was to the **CNewFile.cp** source file, in the EndData function, whose code is as follows:

```
void CNewFile::EndData(CNewFileData *final)
{
   // The values of all panes are returned by this function,
   // which is called just before Close for a modeless dialog,
   // or just before returning from DoModalDialog.

   // If DoModalDialog returns cmdCancel, EndData is called
   // with the values initially supplied to BeginData,
   // allowing you to back out any intermediate changes made
   // in response to UpdateData. If you do not use UpdateData,
   // you can test the value of dismissCmd to see whether to
   // respond to EndData.

   endData = *final;
}
```

As you can see in the foregoing code, a single statement was added to the otherwise empty EndData function. The assignment of the contents of the structure addressed by the `final` argument to the function to the newly added `endData` structure ensures that the structure contains the final values associated with the radio buttons when the dialog is dismissed.

The comments in the EndData function make reference to another function called BeginData. Just for your reference, this function stores the values associated with the dialog's controls (as defined in the VA) into another instance of the CNewFileData structure called `savedData`. It is a pointer to the `savedData` structure that is passed to the EndData function in the `final` argument if the user cancels the dialog. Because we are not using these values when the DoModalDialog function returns other than a `cmdOK` value, we don't care what they contain.

The foregoing code, along with the changes made to the CApp class to invoke the dialog and test the results, are all that are needed to implement the choice between two different document types when the **New** command is chosen. (Because the default behavior for TCL-based applications is for the `cmdNew` command to

be sent to the current gopher when the application is initialized, the new dialog will be displayed when the application is first launched. This behavior can be changed by initializing the application's `newWindowOnStartup` variable to `FALSE` in the IApp function of the CApp class.)

Performing Simple Text File Input and Output

When I described the default-generated code for the CMain class, I indicated that the input and output of data for that document type would be handled largely by the CSaver class, using the Object I/O facilities of the TCL.

In the case of the CNewView class, I glossed over the concept of input and output facilities. When the user chooses the **Open** command from the **File** menu and opens a file whose type matches the value of the `CNewViewFType` constant (which happens to be `'TEXT'` in our case), the OpenDocument function of the x_CApp class (see page 65) determines the appropriate document type to create (CNewView in this case), creates the document object, and then calls its OpenFile function. Because I chose to uncheck the "Use File" checkbox for the NewView view, no code was generated into either the CNewView derived class or the x_CNewView base class source files. Therefore, the OpenFile function is inherited from the CDocument class in the TCL, and is as follows:

```
void CDocument::OpenFile(SFReply*macSFReply)
{
}
```

As is painfully evident, the OpenFile function in the CDocument class is empty. This indicates that we must write the necessary code to perform the OpenFile task if we want to open a file and display its data in the associated view. In fact, if we intend to provide the ability to handle the **Save**, **Save As**, and **Revert** commands, we will also have to add functions to implement them. We have two possible ways to approach this task. These are as follows:

- Add the necessary functions directly to the CNewView class.

- Create a new class that is derived from the CDocument class and then derive our CNewView class from that one, implementing all of the input and output-related functions in the new class.

The first option is more appropriate if we want to handle data in a unique file format that would not likely be used in another document.

The second option provides a class that can be used for other applications (and other views). It is probably the best long-term choice. When we create a new view in the VA and wish to use the features of the new class for input and output, we would specify it as the "Library Class" for the new view. For example, if we launch the VA right now and choose the **Classes** command from the **Edit** menu, the opportunity to choose a class and specify its Library Class is provided, as shown in Figure 3-12.

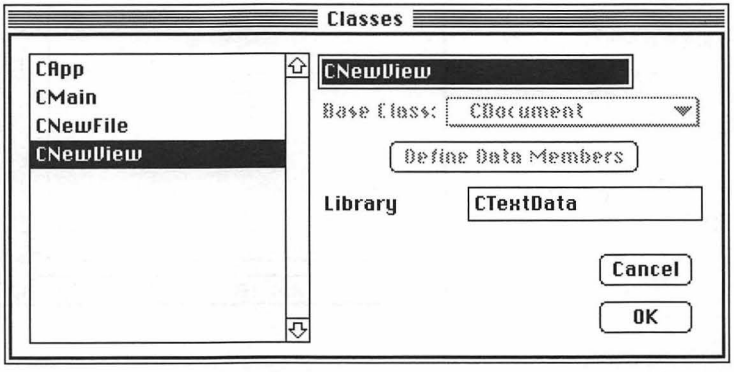

Figure 3-12
Specifying CTextData as the Library Class for CNewView

Creating a CTextEdit Panorama in the NewView Window

In order to illustrate how effective writing a separate "Library Class" will be for our purpose, we will add a CEditText panorama to the NewView window while we are still inside the VA. This will provide an editable text pane that we can use to demonstrate the features of the CTextData class. The steps for doing so are as follows:

1. Close the Classes window, after making the change shown in Figure 3-12, by clicking the OK button.

2. Choose the **View Info** command from the **View** menu and, referring to Figure 3-4, remove the check in the **Vert Scroll**

checkbox. This will ensure that the window itself does not contain a scroll bar. Click OK in the dialog to dismiss it.

3. Double-click the NewView name in the list of views to open the NewView window. Pull down the **Tools** menu and choose the Panorama tool (it looks like a window with horizontal and vertical scroll bars). Position the crosshairs of the Panorama tool at the top-left corner of the grey outline within the view, drag down and to the right until the dotted outline of the panorama just overlaps both the right and bottom edges of the grey outline, and then release the mouse button. The result should look like what is shown in Figure 3-13.

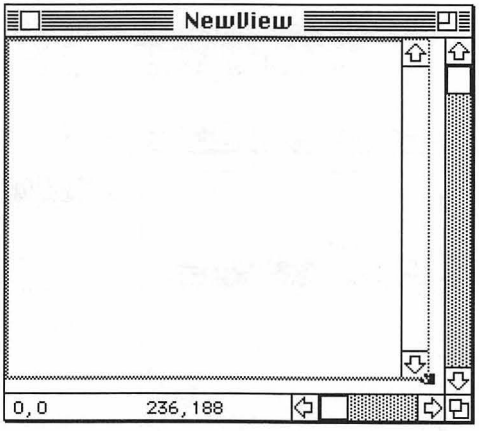

Figure 3-13
Panorama pane added to the NewView window

4. Notice that the panorama just created displays a vertical scroll bar. This is the default setting for that element within the VA. Double-click on the newly created panorama (or click to select it and choose the **Pane Info** command from the **Pane** menu) to open the Pane Info window.

5. Change the name of the pane at the top of the Pane Info window to TextPane, and then also change the sizing characteristics within the CPane object settings to correspond with what is shown in Figure 3-14.

6. Note in Figure 3-14 that the `hSizing` and `vSizing` settings are set to `sizElastic` from the corresponding pop-up

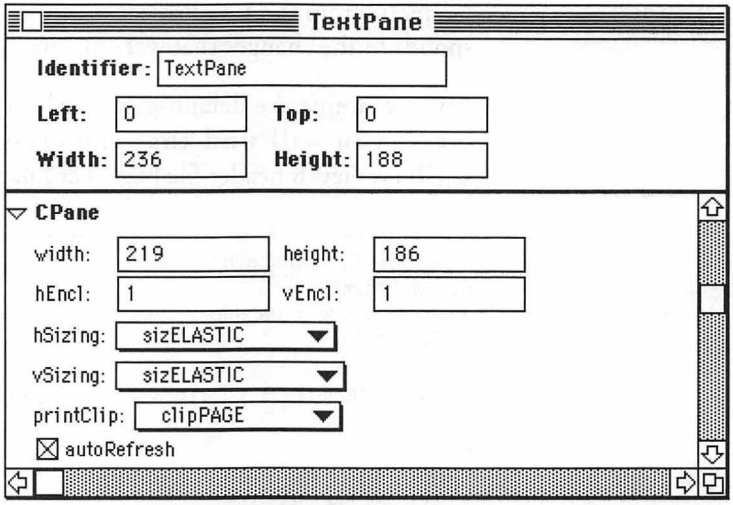

Figure 3-14
Pane Info window for the new TextPane panorama element

menus. These are all that are needed for the new pane. Close the window by clicking in its close box.

7. Choose the **Scrollpane Info** command from the **Pane** menu, and you should see that the settings are as shown in Figure 3-15. You don't have to change any of these settings, so you can close the window after looking at its contents.

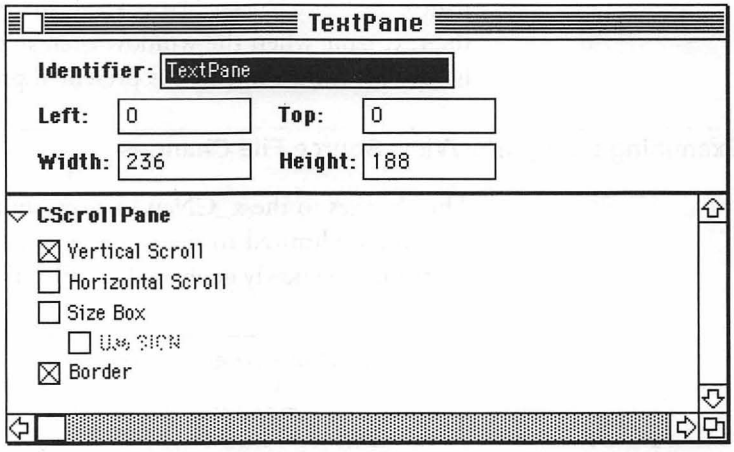

Figure 3-15
Scrollpane Info for the TextPane panorama

Examining the x_CNewView Header File Changes

The foregoing steps complete the changes that are necessary to provide the new panorama subview in the NewView view. Choose

Generate from the VA's **Project** menu to generate code that corresponds to the changes that were just made.

If you examine the default-generated code for the x_CNewView class, you will find that the class declaration in the **x_CNewView.h** header file begins as follows:

```
#include "CTextData.h"
class CEditText;
#define x_CNewView_super CTextData
class CFile;

class x_CNewView : public x_CNewView_super
{
public:

   TCL_DECLARE_CLASS

   // Pointers to panes in window
   CEditText  *fNewView_TextPane;
```

The foregoing code shows that the x_CNewView class is derived from the newly specified CTextData class. In addition to generating a new base class declaration for the x_CNewView class, the VA has also generated a new member variable for the panorama pane called `fNewView_TextPane` that will contain a pointer to the text pane when the window is created. The remainder of the header file is the same as was presented previously.

Examining the x_CNewView Source File Changes

The changes to the **x_CNewView.cp** source file are relatively minor and are limited to the statements in the MakeNewWindow function. The newly generated code for that function is as follows:

```
void x_CNewView::MakeNewWindow(void)
{
   itsWindow = TCLGetNamedWindow("\pNewView", this);
   itsMainPane = (CPane*) TCLGetItemPointer(itsWindow, 0);

   // Initialize pointers to the subpanes in the window
   fNewView_TextPane = (CEditText*) itsWindow
      ->FindViewByID(kNewView_TextPaneID);
  ASSERT(member(fNewView_TextPane, CEditText));
}
```

Examining the CTextData Header File

The **CTextData.h** header file defines the CTextData class and declares the member variables and functions that are needed to implement the simple text input and output features we require. The contents of the file are as follows:

```
/************************************************************
 CTextData

        Header File For CTextData Lower-Layer Data Handling

   Copyright © 1995 Richard O. Parker. All rights reserved.

   This file declares functions to override the empty functions
   in the CDocument class for simple TEXT file I/O.

************************************************************/

#pragma once

class CFile;
class CDataFile;

class CTextData : public CDocument
{
public:

   TCL_DECLARE_CLASS

   Handle itsData;    // place to put the document's data

protected:

   virtual void    OpenFile(SFReply *macSFReply) = 0;
   virtual Boolean DoSave();
   virtual Boolean DoSaveAs(SFReply *macSFReply);
   virtual void    DoRevert();
   virtual void    ReadData();
   virtual void    WriteData();
   virtual void    PositionWindow();
   virtual void    ContentsToWindow();
   virtual void    WindowToContents();
   virtual void    MakeWindowName (Str255 name);
};
```

The foregoing class declaration indicates that the CTextData class is derived directly from the CDocument base class in the TCL. This provides it with the opportunity to implement many of the functions that are empty in the base class implementation.

Examining the CTextData Source File

I have written the **CTextData.cp** source file so that it implements the features of the functions shown in the class definition. Many

of the functions are quite simple and all can be overridden for any view that wishes to perform input or output to other than simple text files. The following sections describe the code contained in the various functions of the CTextData class.

CTextData.cp Beginning Declarations

The first part of the CTextData.cp source file contains declarations and `#include` statements that are needed by the various functions. These are as follows:

```
/************************************************************
 CTextData.cp

          CTextData Document I/O Class

   Copyright © 1995 Richard O. Parker. All rights reserved.

************************************************************/

#include "CTextData.h"
#include <CApplication.h>
#include <CBartender.h>
#include <Commands.h>
#include <Constants.h>
#include <CDecorator.h>
#include <CDesktop.h>
#include <CFile.h>
#include <TBUtilities.h>
#include <CWindow.h>
#include <CDocument.h>

extern CApplication   *gApplication;   // The application
extern CDecorator     *gDecorator;     // The Decorator
extern CDesktop       *gDesktop;       // The visible Desktop
extern CBartender     *gBartender;     // Manages all menus
extern OSType         gSignature;      // Application's creator

TCL_DEFINE_CLASS_M1(CTextData, CDocument);
```

CTextData OpenFile Function Code

The OpenFile function gets called, eventually, as a result of the user's choice of the **Open** command from the application's **File** menu. When this occurs, the DoCommand function for the current gopher is called with a `cmdOpen` argument. The CApp object is still the gopher at this point (unless another document object is already open—in which case its "main pane" will be the current gopher). None of the objects that are both in the chain of command and also in the generated code handle the `cmdOpen` command in their DoCommand functions. As a result, that command gets passed up the chain until it enters the DoCommand

function for the CApplication object in the TCL. The code that handles the command in this function is as follows:

```
case cmdOpen:

     // Have to ask the user what document to open,
     // so do command normally and then sum up
     // for recording

   SFReply macSFReply;
   ChooseFile(&macSFReply);
   if (macSFReply.good)
   {
      SetCursor(*gWatchCursor);
      OpenDocument(&macSFReply);
      if (Factoring())
      {
         CDocument *doc = TCL_DYNAMIC_CAST(CDocument,
            lastAdded);

         TCL_ASSERT(doc);

         doc->SendOpen(kAEDontExecute
            | kAENeverInteract);
      }
   }
   break;
```

The first action of the foregoing code is to execute the ChooseFile function, which asks the user (via the standard open file dialog) what file is to be opened. The ChooseFile function is also implemented in the CApplication class, although it could be overridden in your application-derived class if you wish to use something other than the standard file dialog for choosing files. If the ChooseFile function returns a "good" result, then the user did not cancel the dialog and the foregoing code continues by calling the OpenDocument function, which is implemented in the x_CApp base class in the VA-generated code (see page 65). The OpenDocument function creates a document object that matches the type of the chosen file and then calls OpenFile to open the file that the user chose. The code for the OpenFile function in our new CTextData class is as follows:

```
void CTextData::OpenFile(SFReply *macSFReply)
{
   CDataFile  *theFile = NULL;
   FSSpec     spec;

   theFile = TCL_NEW(CDataFile,());// Create the file object
   try_
   {
      theFile->IDataFile();
      theFile->SFSpecify(macSFReply);
```

```
      theFile->GetFSSpec(&spec);
      FailOpen(&spec);      // Fail if open in this application
      theFile->Open(fsRdWrPerm); // Fail if open in another app
                                 //  (and lots of other reasons)
      itsFile = theFile;         // Promote to document file
   }
   catch_all_()
   {
      TCLForgetObject(theFile);
      throw_same_();
   }
   end_try_

   ReadData();         // Read data from file
}
```

The foregoing code creates a new CDataFile object, initializes the object, calls the SFSpecify and GetFSSpec functions in the CFile class of the TCL to fill in all of the fields of the FSSpec record, and checks to see whether the chosen file is already open in this application. The code continues by calling the Open function for the CDataFile object, which calls the HOpen toolbox function to open the file. If the file cannot be opened, an exception will occur and the `catch_all_` block of the OpenFile code will be executed. If the Open operation is successful, the `theFile` variable's contents are stored into the document's `itsFile` variable, and then the OpenFile function calls ReadData to read all of the data for the chosen file into an handle in memory.

CTextData ReadData Function Code

The ReadData function for our new CTextData class is responsible for reading the file's data, calling the document's MakeNewWindow function to create the appropriate window to hold the text data, and then calling the document's ContentsToWindow function to copy the data to the window. The code for the ReadData function is as follows:

```
void CTextData::ReadData()
{
   Str255  theName;

   try_
   {
      // read all of the file's data into a Handle

      itsData = ((CDataFile *)itsFile)->ReadAll();
   }
   catch_all_()
   {
      throw_same_();
   }
   end_try_
```

```
   if (itsWindow == NULL)          // If window wasn't created
   {
      MakeNewWindow();             //    create the window
   }
   ContentsToWindow();             // Make contents displayable

   PositionWindow();               // Done before making name
   MakeWindowName(theName);        // Change the window title
   itsWindow->SetTitle(theName);
   itsWindow->Select();            // Activate the window
}
```

The foregoing ReadData function reads all of the file's data into memory, storing it into a handle called `itsData`. The handle is defined in the **CTextData.h** header file shown on page 83. After reading the data, the function determines whether a window is already open (this can happen in case ReadData is called as a result of the execution of a `cmdRevert` command); if not, it calls the MakeNewWindow function in the document base class (in this case, x_CNewView). The CTextData class contains only an empty version of that function. ReadData continues by calling the ContentsToWindow function (also empty in the CTextData class, but fully implemented in the CNewView derived class). The ReadData function completes execution by calling the MakeWindowName function to create an appropriate name for the window, calling SetTitle to store the name into the window's title bar, and then calling Select to select and make the new window active.

CTextData DoSave Function Code

When the user chooses the **Save** command from the application's **File** menu, the `cmdSave` command travels up the chain of command until it reaches the DoCommand function in the CDocument class, where it is handled as follows:

```
   case cmdSave:
      Boolean specify = itsFile == NULL;
      SetCursor(*gWatchCursor);
      DoSave();
         // We send the Save event after doing the save to
         // give the user a chance to specify a save file
         // if one is not already open. If itsFile is NULL
         // after calling DoSave, the user canceled.

      if (factoring && itsFile)
         SendSave(specify, kAEDontExecute);
      break;
```

The foregoing code changes the cursor to the "watch" icon and then calls the DoSave function to perform the save operation. Our CTextData class contains an override for the DoSave function and its code is as follows:

```
Boolean CTextData::DoSave()
{
   if (itsFile == NULL)
   {
      return DoSaveFileAs();
   }
   else
   {
      WriteData(); // need to override this function
      return TRUE;
   }
}
```

As is evident in the foregoing code, if the `itsFile` variable has not yet been assigned (as would be the case when the user created a new CNewView window, entered some data, and then chose the **Save** command from the **File** menu), the function calls the DoSaveFileAs function to perform the save operation; otherwise, if the file has already been created, the DoSave function calls the WriteData function to write out the data and indicate its success in doing so.

CTextData WriteData Function Code

The WriteData function for the CTextData class operates in reverse to the ReadData function. The code is as follows:

```
void CTextData::WriteData()
{
   try_
   {
      gApplication->SetCriticalOperation(TRUE);
      WindowToContents();   // get data from window
      if (itsData && itsFile)
      {
         ((CDataFile *)itsFile)->WriteAll (itsData);
      }
      SetChanged(FALSE);    // Not dirty any longer
      gApplication->SetCriticalOperation(FALSE);
   }
   catch_all_()
   {
      gApplication->SetCriticalOperation(FALSE);
      throw_same_();
   }
   end_try_
}
```

The foregoing code commences by calling the SetCriticalOperation function of the application object with a TRUE argument that enables the function to use more memory from the memory reserve, if it becomes necessary to do so. The WindowToContents function is then called to move the data from the window to the itsData handle (perhaps creating a new handle in the process); if the itsData and itsFile variables are both nonzero, the WriteAll function of the CDataFile class is called to write out the contents of the itsData handle to the itsFile file. The document's SetChanged function is called with a value of FALSE, indicating that the document is no longer "dirty," and then the SetCriticalOperation function is called with a FALSE value to indicate that the critical operation is complete. If an exception occurs during any of the foregoing processes, then the catch_all_ block will be executed.

CTextData DoSaveAs Function Code

In the event that a file has not yet been assigned when the user chooses the **Save** command, the DoSaveAs function is called. Also, if the user chooses the **Save As** command from the **File** menu, the cmdSaveAs command is passed up the chain of command until it is handled by the DoCommand function in the CDocument class. The code for that section of the function is as follows:

```
case cmdSaveAs:
   DoSaveFileAs();
   if (factoring && itsFile)
      SendSave(TRUE, kAEDontExecute);
   break;
```

The foregoing code calls the DoSaveFileAs function that is also implemented in the CDocument class of the TCL. The code for that function is as follows:

```
Boolean CDocument::DoSaveFileAs()
{
   SFReply macSFReply;         // Standard File reply record

   PickFileName(&macSFReply); // Let user enter a file name

   if (macSFReply.good)
   {
      SetCursor(*gWatchCursor);
      return(DoSaveAs(&macSFReply) );
   }
```

```
    else
    {
        return(FALSE);          // User cancelled SaveAs dialog
    }
}
```

The DoSaveFileAs function displays the standard SaveAs dialog box and allows the user to specify the name and location of the file to be saved. If the dialog is not cancelled, the function continues by setting the cursor to the "watch" icon and then calls the DoSaveAs function to handle the remainder of the task. Our CTextData class overrides the DoSaveAs function, the code for which is as follows:

```
Boolean CTextData::DoSaveAs(SFReply *macSFReply)
{
    OSErr       theError;
    CDataFile   *theFile = NULL;
    FSSpec      spec;

    TCLForgetObject(itsFile);       // Delete the current file
    theFile = TCL_NEW(CDataFile,());// Create a new file

    try_                        // Most of the following operations
                                //  can fail for various reasons
    {
        theFile->IDataFile();
        theFile->SFSpecify(macSFReply);
        theFile->GetFSSpec(&spec);
        FailOpen(&spec);
        if (theFile->ExistsOnDisk())  // If file is on disk
        {
            theFile->ThrowOut();      //  dispose of it
        }
        // create and open a TEXT file
        theFile->CreateNew(gSignature, 'TEXT');
        theFile->Open(fsRdWrPerm);

        // Change window name to match the new file name
        if (itsWindow)
        {
            itsWindow->SetTitle(macSFReply->fName);
        }
    }
    catch_all_()
    {
        TCLForgetObject(theFile);
        throw_same_();
    }
    end_try_

    itsFile = theFile;              // Set instance variable

    // Let DoSave() write the document to disk.  Note
    // that DoSave() may have called DoSaveAs(), but
    // itsFile is now non-NULL, so the recursion stops.
    return DoSave() ;
}
```

The foregoing code implements the **Save As** command—or is executed when called by the DoSave function—by disposing of any CDataFile object that is currently allocated, creating a new object, initializing the object, and filling in the fields of the `macSFReply` record. Then, if the chosen file already exists on the disk, it is disposed and a new file is created and opened. The window title is changed to reflect the new file name, the document's `itsFile` variable is set to point to the new CDataFile object, and then the code calls the DoSave function to complete the operation of writing the contents of the `itsData` handle to the disk.

The comments at the end of the foregoing code point out that although the DoSaveAs function might have been called by the DoSave function it, in turn, calls the DoSave function to complete its task only after a file has been created to hold the data to be written. Therefore, any possible recursion is avoided.

CTextData DoRevert Function Code

The DoRevert function is called by the DoCommand function in the CDocument class when the user chooses the **Revert** command from the application's **File** menu. The section of code in the DoCommand function that handles the `cmdRevert` command is as follows:

```
case cmdRevert:
   PositionDialog('ALRT', ALRTrevert);
   if (CautionAlert(ALRTrevert, NULL) == OK)
   {
      SetCursor(*gWatchCursor);
      DoRevert();
   }
   break;
```

The foregoing code displays a dialog that asks whether the user really wants to revert to the version of the file that was previously saved. If the user clicks the OK button, indicating acceptance of the proposed action, then the code changes the cursor to the "watch" icon and calls the DoRevert function to perform the revert operation.

Our CTextData class overrides the DoRevert function for text files and the code for this is as follows:

```
void CTextData::DoRevert()
{
   if (itsData)
   {
      DisposHandle (itsData);
   }
   if (itsFile)
   {
      ReadData();
   }
}
```

The foregoing code is quite simple. It checks to make sure that the `itsData` handle has been allocated and disposes it if so. Then, if the file has been created, it calls the ReadData function to reread the contents of the file on disk. Both of the tests are provided to bulletproof the code, as both the data handle and the corresponding file should have been created before the **Revert** command is enabled in the **File** menu.

CTextData PositionWindow Function Code

The PositionWindow function is called by the ReadData function to position a newly created window on the desktop. Our CTextData class overrides this function to allow the CDecorator object to perform the default action of staggering the window. The code is as follows:

```
void CTextData::PositionWindow()
{
   gDecorator->PlaceNewWindow(itsWindow);
}
```

CTextData ContentsToWindow Function Code

When the ReadData function is called, after the data file has been read into the `itsData` handle and a new window is constructed, the ContentsToWindow function is called to transfer the data to the window in an appropriate manner. You must override the default code in your derived document class (for example, CNewView). The default code for the function is as follows:

```
void CTextData::ContentsToWindow()
{
   // you must override this function
}
```

CTextData WindowToContents Function Code

Before the WriteData function can write the contents of the `itsData` handle to the disk, the WindowToContents function is called to ensure that the `itsData` handle contains the latest copy of the window's data. You must override this function in your derived document class (for example, CNewView). The default code for the function is as follows:

```
void CTextData::WindowToContents()
{
   // you must override this function
}
```

CTextData MakeWindowName Function Code

When the ReadData function executes, after a new window is created, it calls the MakeWindowName function to create a name for the new window. The default code for the function is empty; however, the VA-generated code in the base class for the derived document class source file (i.e., **CNewView.cp**) overrides this function. The default code is as follows:

```
void CTextData::MakeWindowName (Str255 name)
{
   // you must override this function
}
```

The VA-generated code for the MakeWindowName override function in the x_CNewView class is as follows:

```
void x_CNewView::MakeWindowName(Str255 newName)
{
   Str31 count;

   if (itsFile != NULL)
      itsFile->GetName(newName); /* Return file name*/
   else
   {
      /* Append window count to window*/
      /*  title (from resource)        */
      itsWindow->GetTitle(newName);
      if (Length(newName) == 0)
         GetIndString(newName, STRcommon, strUNTITLED);
      NumToString(gDecorator->GetWCount(), count);
      ConcatPStrings(newName, (unsigned char *)"\p ");
      ConcatPStrings(newName, count);
   }
}
```

As is evident from the foregoing code, the window name is created by combining the name of the file that was opened with a sequence number supplied by the "decorator," which maintains a count of the windows under its management. You can override the MakeWindowName function in the CNewView derived class to return whatever sort of window name is appropriate.

Examining the CNewView Source File Changes

In order to implement fully the new text pane in the NewView window, I added some code to the **CNewView.cp** source file. I did not add any new functions to the file, so the header file is unchanged. The sections that follow describe the changes to the existing VA-generated functions.

CNewView ICNewView Function Code

I added a single statement to the ICNewView function to initialize the `itsData` handle to a `NULL` value, as follows:

```
void CNewView::ICNewView()
{
   Ix_CNewView();

   // Initialize data members here

   itsData = NULL;     // make sure data is NULL
}
```

CNewView ContentsToWindow Function Code

As you may recall, the ContentsToWindow function is called by the CTextData ReadData function, after the contents of the data file has been read, to install the data into the current window. Because that operation is application dependent, we must override the function in our CNewView class. The VA generates an empty function for this purpose; I added the following code to make it fully functional:

```
void CNewView::ContentsToWindow()
{
   // set the CEditText's Handle to our data

   fNewView_TextPane->SetTextHandle (itsData);
}
```

To display the data in memory in the CEditText pane in the NewView window, all we have to do in the foregoing code is call the SetTextHandle function for the CEditText pane, passing it the handle to our data. The TCL copies the data in our handle into a new handle and then manages its own copy, displaying the data using the TextEdit features of the toolbox.

CNewView WindowToContents Function Code

The WindowToContents function is called by the WriteData function of the CTextData class in order to store the latest version of the window's contents into the `itsData` handle for the document so that the data can be written to a file. The code for this function is as follows:

```
void CNewView::WindowToContents()
{
   Handle theData;

   // check whether there is existing data in the "itsData"
   // variable and dispose the data if so. Then, get the Handle
   // from the CEditText pane and copy it and the data into a
   // new Handle.
   if (itsData != NULL)
   {
      DisposHandle (itsData);
   }
   theData = fNewView_TextPane->GetTextHandle ();
   itsData = theData;
   FailOSErr(HandToHand (&itsData));
}
```

When the foregoing function is called, a file might not yet be assigned (that is, the user created a new window) and the `itsData` handle might be `NULL`. If not, then a file has been assigned and the contents of the handle may not reflect what is displayed currently in the window. In this case, we dispose of the handle and then, in either case, create a new handle by calling the GetTextHandle function of the CEditText class, and then copy the handle and its data using the HandToHand toolbox function.

CApp ICApp Function Code Addition

I have made one more small addition to the IApp function in the CApp class to suppress the display of the NewFile dialog on startup. Because I want the user to be able to either open an existing file or create a new file at his or her discretion, I will launch the

application without displaying a window at the outset. The code to accomplish this is as follows:

```
voidCApp::ICApp()
{
   Ix_CApp(4, 24000L, 20480L, 2048L);
   // Initialize your own application data here.
   newWindowOnStartup = FALSE;
}
```

The foregoing changes are all that are required to fully implement the newly defined view and its CEditText panorama. After the changes have been made and the application is recompiled and run, the user will have the option of opening an existing file or choosing the **New** command from the **File** menu. In the latter case, the user will see the dialog displayed in Figure 3-16:

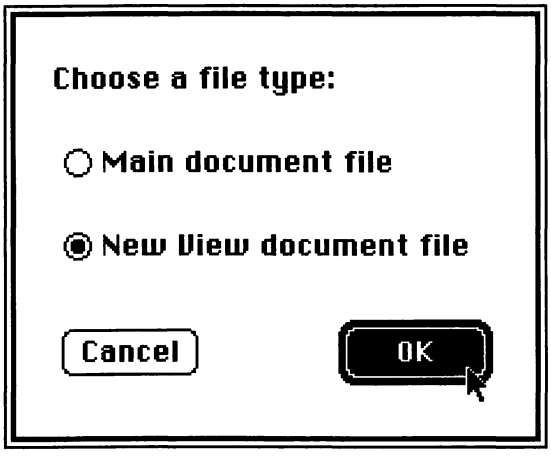

Figure 3-16
Choosing a New View file type

As is evident in the foregoing figure, the user has chosen a New View document file that results in the construction of a CNewView document object and its associated view. The user is able to enter data into the view by using the keyboard. After several paragraphs have been entered, the view may appear like what is shown in Figure 3-17.

We have entered enough data into the window to activate the vertical scroll bar, just to illustrate that it is possible to do so. We were also able to use the **Copy** and **Paste** commands in the **Edit** menu to duplicate the sentence that is displayed in the figure. These features are provided automatically by the TCL for CEditText objects.

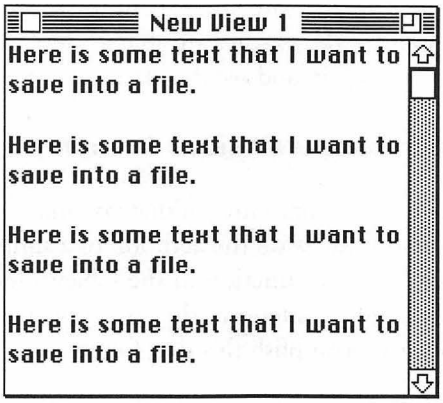

Figure 3-17
New View window containing newly entered text

One other thing to note in Figure 3-17 is that the window title is the default value constructed for a new window of the New View type. It includes the name of the view and the CDecorator's sequence number. After saving the foregoing window's contents into a file—by choosing the **Save** or **Save As** command from the application's **File** menu—the window's title is changed to reflect the file's name, as shown in Figure 3-18.

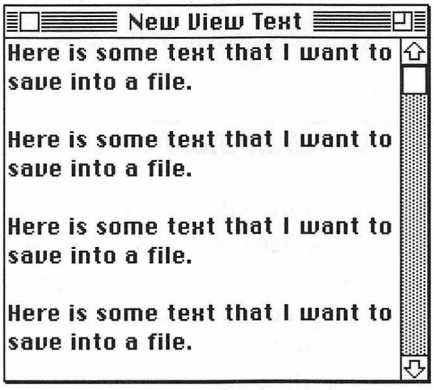

Figure 3-18
Window title after data has been saved to a file

In the case of the foregoing window, the data was saved to a file named **New View Text**. Using the facilities that have been de-

scribed in the foregoing sections, as well as the comprehensive support of the TCL, we are able to close that window and open the file once again and see the identical contents in the new window.

CApp MakeNewWindow Override Suggestion

There's one final addition that you may wish to make to the newly created view. With the addition of a simple override of the MakeNewWindow function in the CNewView class, you can arrange to print the contents of the text in the view. The override function code to accomplish this is as follows:

```
void CNewView::MakeNewWindow(void)
{
   x_CNewView::MakeNewWindow();

   // change itsMainPane to point to the CEditText panorama
   itsMainPane = fNewView_TextPane;
}
```

The foregoing code illustrates the simplicity of changing the value of the `itsMainPane` pointer to reference the text pane, rather than the window's panorama (as is the case with the VA-generated code). The base class function is called first, followed by the statement to change the value of the `itsMainPane` pointer.

Multiple Documents Versus Multiple Views

It is important to understand the distinction between multiple file types, their associated document objects, and simply the case of multiple views of the data for a single document and file type. In the foregoing sections, I described how to create separate document objects to handle single or multiple file types. The distinction between these two topics is very simple. If you need to open multiple files, then (with few exceptions) you should define a new "Main Window" view for each. If, instead, you wish to open a single file and show various views of its contents, then you will want to create a single "Main Window" view and then multiple plain "Window" views to display the data in different ways.

Application Document Summary

This chapter has described the role of the document object in the application. The document is the data server for the application and an application can have one or more document objects. Each document object has an associated file, and each file has an associated file type; therefore, each unique file type has an associated document object.

We also described the code for the CTextData class, which could be used to perform simple text input and output from/to a `'TEXT'` file. A more comprehensive simple file I/O functionality (CSimpleSaver) is described in a later chapter.

In the next chapter, I will discuss how to create multiple views on the data contained within one or more documents and their associated files.

Chapter 4

Creating and Displaying Views

This chapter is all about views of various types. The intention here is to present several different types of views so that you will get some idea of how they are created and what you need to do to implement them fully in your own applications.

I will be using the VA to create the various view types and I will also present its default-generated code, along with custom code and suggestions on how to complete the customization process.

All of the views in this chapter are based upon windows, rather than dialogs; however, I will show dialogs that are associated with the implementation of the new views, but will cover them in more depth in a later chapter. With that said, let's begin the chapter with a very interesting view.

Creating a Business Account View

You are probably very familiar with a class of commercial software that allows you to manage your money in various ways. These applications provide you with an account register view into which you enter your transactions for checking, savings, credit cards, and so forth. It is the account register view that you will learn how to create using the VA and the TCL.

Implementing the account view consists of many more elements than just the view itself because you will also want to include the following features with such a view:

- A main view that holds a list of account names, from which you can select the account to be viewed
- A dialog that allows you to name the account, choose its type, and store other pertinent information about the account

- An "Account" menu that offers the ability to create a new account, edit an existing account, delete an account, or simply select an account to view
- A separate view for each account, so that multiple account views can be displayed simultaneously

Creating the Main View

In order to satisfy the foregoing list of features, let's create the list of accounts (Main) view such that it displays the name of each account. Clicking on any entry in the list will immediately show its corresponding view. The steps to create this behavior are as follows:

1. Launch the Symantec Project Manager application and choose to create a new project using the "VA Application" project model.
2. Choose a new folder in which to store the project, name it "Business View *f*," name the project "Business View," and then let the Project Manager create the project file using the VA Application as its template.
3. When the project has been created, double-click on the "Visual Architect.rsrc" entry to launch the VA application.
4. When the VA launches, it displays a list of existing views that at the outset includes only the "Main" view (as shown in Figure 2-1). Double-click on the Main entry in that list and the predefined main view (as shown in Figure 2-2) will be displayed.
5. Choose the **Select All** command from the **Edit** menu and press the Delete key to delete the existing view components (a PICT image and a static text field).
6. Pull down the **Tools** menu and choose the List/Table tool (fifth tool down in the first column). You may also want to tear off the Tools palette so that it will be handy for use in some of the later steps in this section.
7. With the List/Table tool selected, position the cursor crosshairs at the top-left corner of the window boundary and

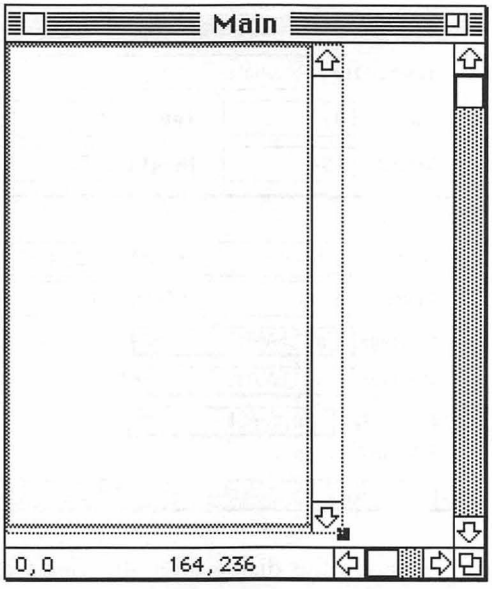

Figure 4-1
Scroll list created in Main view

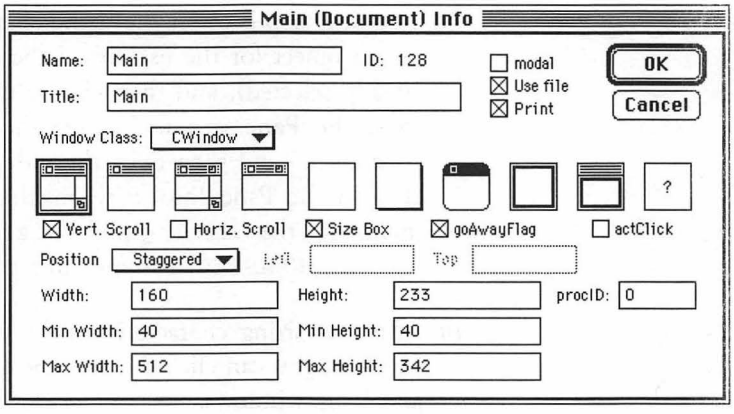

Figure 4-2
View Info settings for the Main view

drag down and to the right to create a list pane that looks similar to what is shown in Figure 4-1.

8. Now, pull down the **View** menu and choose the **View Info** command and change the settings to correspond with what is shown in Figure 4-2. Note in the figure that the view has a vertical scroll bar, a size box, a close box (go-away flag), and that the "Use File" checkbox is checked. You will probably want to use the Object I/O features of the TCL in the application that is built around this view. Click the OK button to

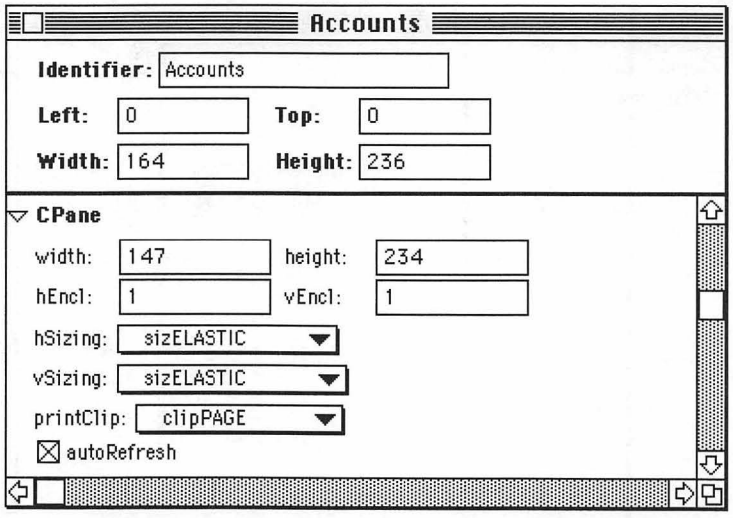

Figure 4-3
Setting hSizing and vSizing values for the Accounts list pane

dismiss the dialog after the view's characteristics have been specified.

9. The next step is to set the sizing characteristics of the CPanorama object for the list. Select the new list pane (if it isn't already selected), and then choose the **Pane Info** command from the **Pane** menu. Change the name of the pane to "Accounts," and then twist the indicator next to the CPane class in the Pane Info view, making sure that the pop-up menus for the `hSizing` and `vSizing` parameters are both set to `sizELASTIC`, as shown in Figure 4-3.

10. After the sizing characteristics for the Accounts pane have been set, you can click the close boxes on both the Accounts and Main windows.

11. At this point, you need to create a new class, derived from CArrayPane so that you can override the "GetCellText" function, in order to display the account names in the Accounts pane. Choose **Classes** from the **Edit** menu, and then choose the **New Class** command from the **Edit** menu. Create the CMainList class, as shown in Figure 4-4, and then dismiss the dialog.

12. After the CMainList class has been created, double-click on the Main view in VA's list of views, click to select the list pane,

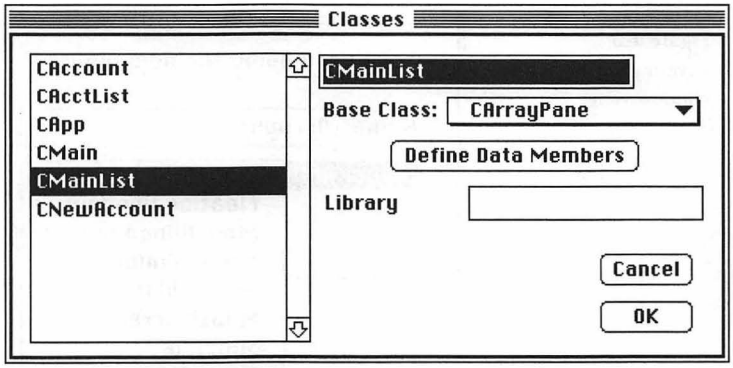

Figure 4-4
Creating the CMainList class

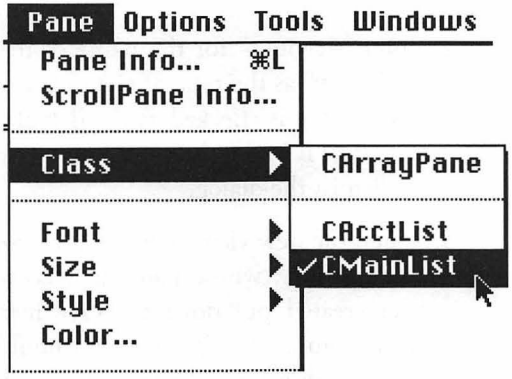

Figure 4-5
Changing the list pane class to CMainList

and then pull down the **Pane** menu and choose CMainList as the Class for that pane, as shown in Figure 4-5.

This concludes the procedure for creating the "Main" view. Its sole purpose is to display a list of accounts and their balances, from which an individual account view can be selected.

Creating the Account View

The next procedure concerns the creation of the Account view. The steps for doing so are as follows:

1. Assuming that the VA is still running and that you are continuing on from the previous tutorial, pull down the **View** menu and choose the **New View** command.

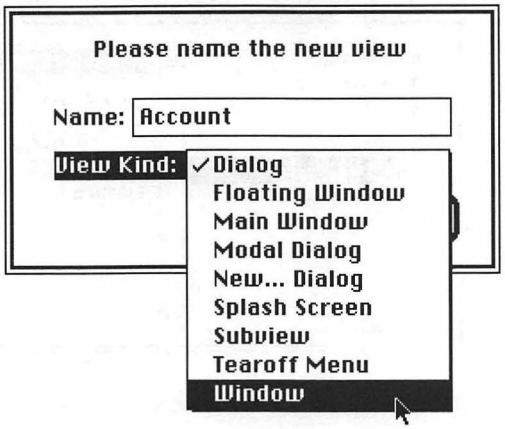

Figure 4-6
Creating the Account window view

2. Enter "Account" for the name of the view, and then choose "Window" as the type of view, as shown in Figure 4-6. (Note that Dialog is checked as the default selection, but the Window type is being chosen in the figure.) Click the OK button to dismiss the dialog.

3. When the new view dialog is dismissed, the VA will display a new window, whose name is "Account." After the view has been created, pull down the **View** menu, and choose the **View Info** command. The settings should be changed to correspond with what is shown in Figure 4-7.

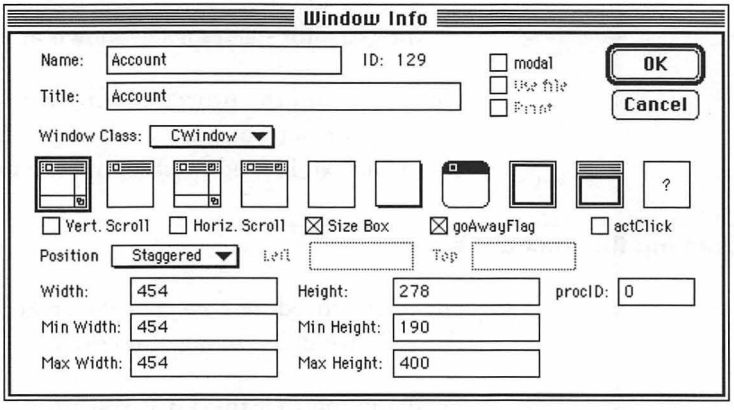

Figure 4-7
View Info settings for the Account window

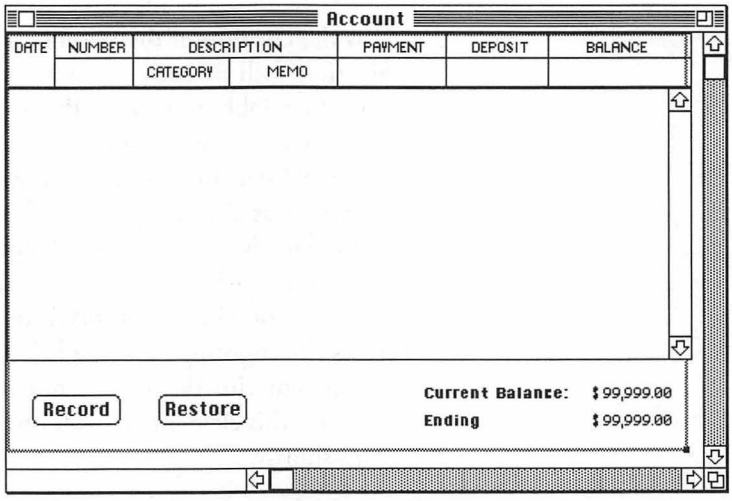

Figure 4-8
Finished appearance of Account view

Just to help you visualize what needs to be accomplished in the following steps, the final appearance of the Account view is displayed in Figure 4-8. The view consists of quite a number of elements, so you may wish to tear off the Tools palette so that the various tools are easier to reach.

4. The column heading section of the view shown in Figure 4-8 was created by choosing the Rectangle tool and drawing an outline large enough to surround all of the heading info at the top of the view (our outline is named "TitleBox" and it's 456x36 pixels in size). Both the `hSizing` and `vSizing` parameters in the CPane class are set to sizFIXEDSTICKY and the pen "h" and "v" settings in the CShapeButton class are set to 1 (indicating a 1-point pen width and height). All of the settings were made by selecting the rectangle and choosing **Pane Info** from the **Pane** menu. The various elements within the TitleBox were created as follows:

    a. The vertical and horizontal lines were drawn using the Line tool, and each of these is also a 1-point line.

    b. The text inside the pseudo columns of the TitleBox is in all capital letters in the Chicago 9-point font. Use the hierarchical **Font** menu, within the **Pane** menu, to select the font, and then the hierarchical **Size** menu, within the **Pane** menu to select the font size.

5. The next major element in the construction of the Account view is the creation of the scrolling list in which the account transactions will be displayed. Construct this pane by choosing the List/Table tool from the Tools palette, position the cursor crosshairs just below the bottom-left corner of the TitleBox construction, and drag down and to the right until you have created a list pane that is the same width as the TitleBox and is also an integral multiple of its height (the pane measures 456x216 pixels, which allows space for six entries, each of which is 36 pixels in height). Name the pane "Entries" by choosing the **Pane Info** command from the **Pane** menu and entering the name. The default assignment of CArrayPane for this element is fine for now, but we will be changing this shortly.

6. The next element to be created is the Record button at the bottom of the view. Note that we have left space below the scroll pane, created in the foregoing step, to place two buttons and four static text fields. To create the Record button (that when pushed saves the transaction just entered), simply choose the Button tool from the Tools palette, and click where you wish the button to be placed.

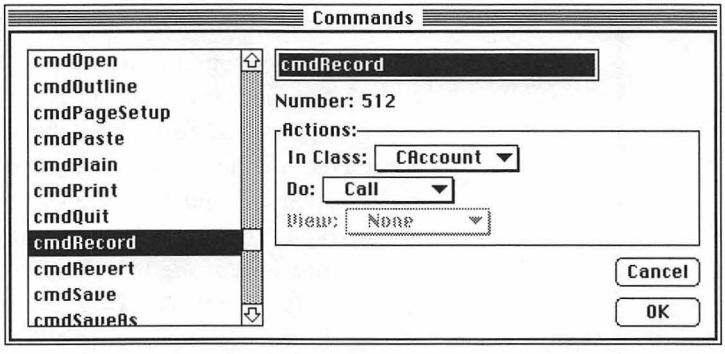

Figure 4-9
Record button command behavior

After placing the Record button and entering its name, you will want to assign a command to be executed when the button is clicked. This is called a "behavior." To assign the command behavior for the Record button, make sure the button is still selected, and then pull down the **Pane** menu and choose the **Pane Info** command. Twist the indicator next to the CButton class, click the pop-up menu next to the `Command` setting, and choose **Other** from the top of the menu of

available commands. This will display a Commands window with the `cmd1HalfSpace` command selected by default. Notice that the **Edit** menu is active when this dialog is being displayed. Choose the **New Command** command from that menu (or use the Command-K keyboard shortcut), type in a new command name (I used `cmdRecord`), choose CAccount as the class in which the command is to be executed (you want the Account class to handle commands associated with the Account view), and then choose `Call` as the action to perform when the command is executed. This will result in the DoCommand function for the CAccount class to contain code to call a function called cmdRecord. The completed dialog settings are shown in Figure 4-9.

7. The next step is to modify the settings in the CControl class for the Record button. Twist the indicator next to that class name, and change the settings to correspond with what is shown in Figure 4-10.

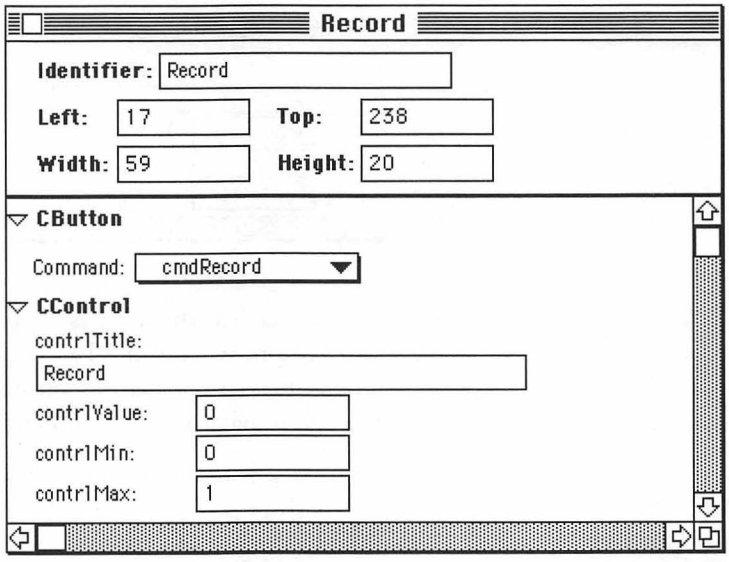

Figure 4-10
Record button settings

8. Next, just as we described in the steps pertaining to the Record button, create a new button called Restore, place it onto the view as shown in the depiction of the completed view in Figure 4-8, open the Pane Info dialog, and create a new command (I used `cmdRestore`) for the button, with the behavior shown in Figure 4-11.

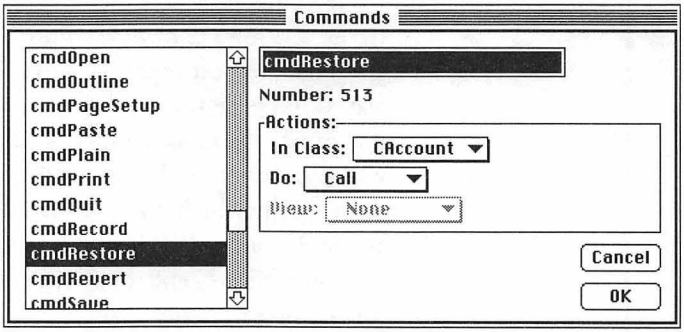

Figure 4-11
Restore button command behavior

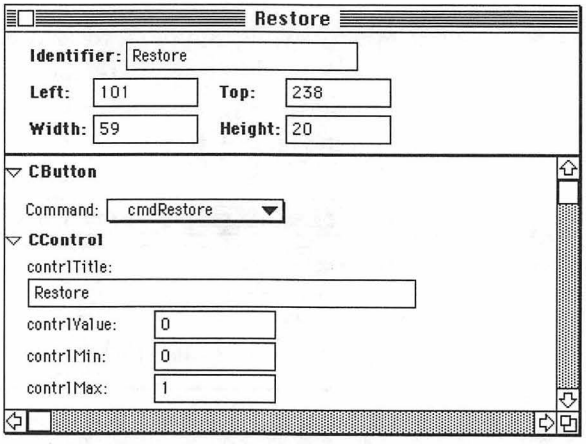

Figure 4-12
Restore button settings

9. Just as we specified the settings for the CControl class for the Record button, we also need to choose the **Pane Info** command and make sure that the settings for the Restore button correspond to what is shown in Figure 4-12.

10. The next step is to create a new class called CAcctList, which we will derive from the CArrayPane class to enable us to customize the appearance of the main (Entries) transaction list in the view. Pull down the **Edit** menu, choose **Classes**, and then choose **New Class** from the **Edit** menu, after the Classes window is displayed. Change the settings for the new class to match what is shown in Figure 4-13.

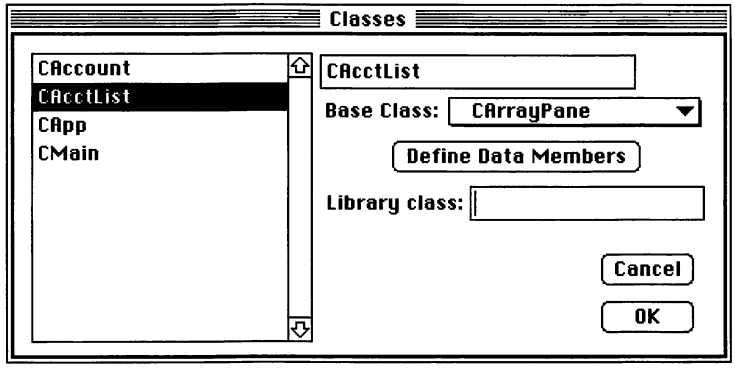

Figure 4-13
Creation of new CAcctList class

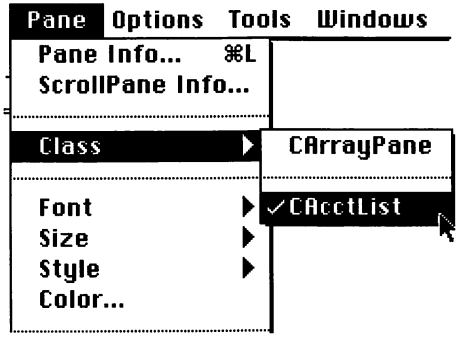

Figure 4-14
Changing the Entries pane to the CAcctList class

11. Then we need to change the class for the "Entries" pane to be CAcctList, instead of the default CArrayPane that the VA has assigned. This is easily accomplished by selecting the "Entries" list pane, pulling down the **Pane** menu, and selecting **CAcctList** in the list of acceptable classes in the **Class** hierarchical menu, as shown in Figure 4-14.

12. In addition to the foregoing elements, we have also created four static text fields, as follows:

    a. The "Current Balance" label field is created at the bottom of the Account view. It need not have a special name.

    b. The "Ending Balance" label field is created at the bottom of the Account view. It need not have a special name.

c. Another static text field that we have called "CurBal" is created at the bottom of the Account view, to the right of the "Current Balance" label. We have entered an initial value into the field, but this will be replaced by the program with the actual value when the account balance is computed.

d. The final static text field is called "EndBal," and it is created to the right of the "Ending Balance" label. We have also entered an initial value into this field, but it will be replaced by the program when the ending balance is computed.

This concludes the creation of the elements of the Account view. The positions and sizes of all of the foregoing elements are summarized in Table 4-1.

Table 4-1
Position and sizing of Account view elements

Element Name	Dimensions*	Sizing†
TitleBox	(0, 0, 456, 36)	H:F, V:F
Line11	(32, 0, 1, 36)	H:F, V:F
Line12	(84, 0, 1, 36)	H:F, V:F
Line13	(220, 0, 1, 36)	H:F, V:F
Line14	(292, 0, 1, 36)	H:F, V:F
Line15	(360, 0, 1, 36)	H:F, V:F
Line2	(32, 16, 424, 1)	H:F, V:F
Line19	(148, 16, 424, 1)	H:F, V:F
Entries	(0, 36, 456, 216)	H:E, V:E
Record	(17, 238, 59, 20)	H:F, V:B
Restore	(101, 238, 59, 20)	H:E, V:B
Stat23	(276, 232, 97, 16)	H:E, V:B
CurBal	(388, 232, 60, 12)	H:E, V:B
Stat24	(276, 250, 91, 12)	H:E, V:B
EndBal	(388, 250, 60, 12)	H:E, V:B

* Order is (Left, Top, Width, Height)

† F = sizFIXEDSTICKY, E = sizELASTIC, B = sizFIXEDBOTTOM

In the table, the "Line" elements are the vertical and horizontal lines; the "Stat" elements are static text, as are the special static text items named "CurBal" and "EndBal"; the "TitleBox" element

is the rectangle that encloses all of the other heading elements; the "Entries" element is the scrolling list; and the "Record" and "Restore" elements are the buttons at the bottom of the view.

Adding an Account Menu

The next few steps create a new Account menu and four new menu commands. Then the menu is added to the standard menu bar. The steps are as follows:

1. Choose the **Menu Bar** command from the **Edit** menu.
2. Choose the **New Menu** command from the **Edit** menu (or use the Command-K keyboard shortcut).
3. Enter the name **Account** into the Edit Text field at the top-right side of the Menu Bar dialog. The result is as shown in Figure 4-15. After the new menu has been added, click the OK button to dismiss the Menu Bar dialog.

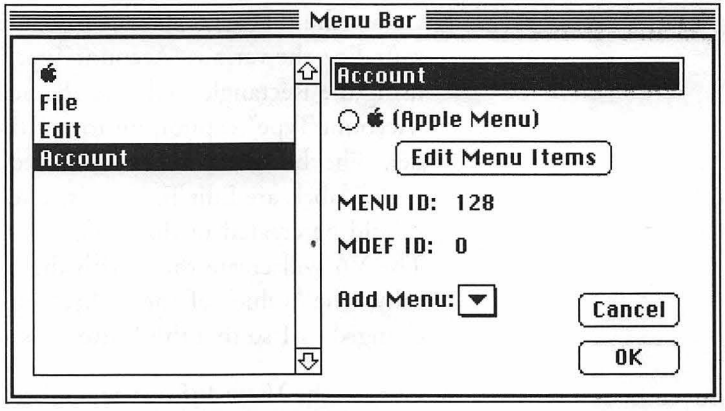

Figure 4-15
Account menu added to menu bar

This concludes the procedure for adding the Account menu to the menu bar. Before adding commands to the menu, we will construct a New Account dialog.

Creating a New Account Dialog

Two of the menu commands we want to add to the Account menu require that we display a dialog that lists the settings for an account. Before adding the menu commands, we will create the dialog so that the VA will generate code to open the dialog when

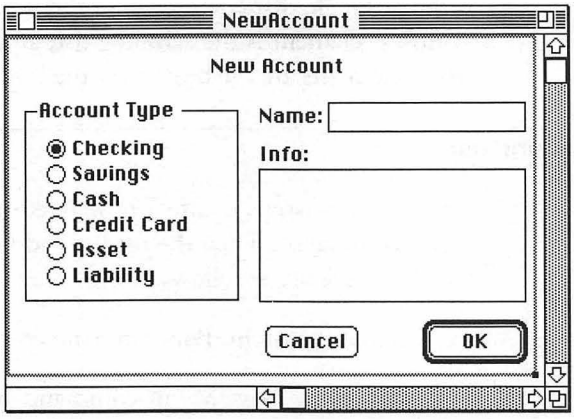

Figure 4-16
NewAccount dialog as displayed in the VA

the command is chosen. The steps for creating the New Account dialog are as follows:

1. Choose **New View** from the **View** menu, and then name the view "NewAccount." Select Dialog as the type of view.

2. Create the dialog as shown in Figure 4-16. The border surrounding the various "Account Type" radio buttons is created using the Rectangle tool. Use the Static Text tool to type the "Account Type" caption on top of the upper area of the border. The bordered fields associated with the "Name" and "Info" labels are Edit Text fields. The OK and Cancel buttons should be created in that order (first OK and then Cancel). The VA will create them with the proper settings automatically. The "value" of the "Checking" radio button has been changed to 1 so that this button is selected, by default.

3. Choose the **View Info** command from the **View** menu, and make sure that the "Modal" checkbox is checked and that the view type is the third picture from the right (standard bordered dialog). Click OK to dismiss the dialog, and then click in the close box of the NewAccount view to close the view's window.

This concludes the steps for creating the NewAccount dialog. I have purposely avoided going into much detail about the rationale for the dialog's construction, mainly because knowing this information is not essential to the explanation of how the Business View operates. All of the foregoing, and much of what follows, is

preliminary to the discussion of how to implement the Account view. Although it is necessary to show the various components that interact with the view, it isn't really necessary to describe why they have been implemented—at least, not at this point.

Adding Menu Commands to the Account Menu

After the NewAccount dialog has been constructed, the VA will have knowledge that files for a new class called NewAccount will need to be generated. It also knows that the class concerns a modal dialog view. With this in mind, we can now add commands to the **Account** menu we constructed earlier. The steps are as follows:

1. Choose **Menus** from the **Edit** menu, and select the Account menu in the list of available menus.

2. Click the Edit Menu Items button in the Menus dialog, and then choose the **New Menu Item** command from the **Edit** menu (or use the Command-K keyboard shortcut).

3. Enter the name **New Account**, followed by an ellipsis character (indicating that a dialog will open when the command is chosen. (*Note:* The ellipsis character is created by typing the semicolon key with the Option key held down.)

4. Click the pop-up menu at the bottom of the menu item dialog, next to the label "Command." Scroll up to the top of the list and choose "Other." This will cause the "Commands" dialog to open. Choose **New Command** from the **Edit** menu (or use the Command-K keyboard shortcut). Enter the name `cmdNewAcct` into the Edit Text box, and change the settings to match what is shown in Figure 4-17.

   Note that in the Actions settings for the `cmdNewAcct` entry we have chosen CMain for the class in which the code is to be generated; we have chosen the Open action for the code to "Do;" and we have chosen CNewAccount (the NewAccount dialog view) for the generated code to open.

5. Repeat the actions described in the foregoing steps 3 and 4 for a new command named `cmdEditAcct`, and set the Actions for that command to be as shown in Figure 4-18.

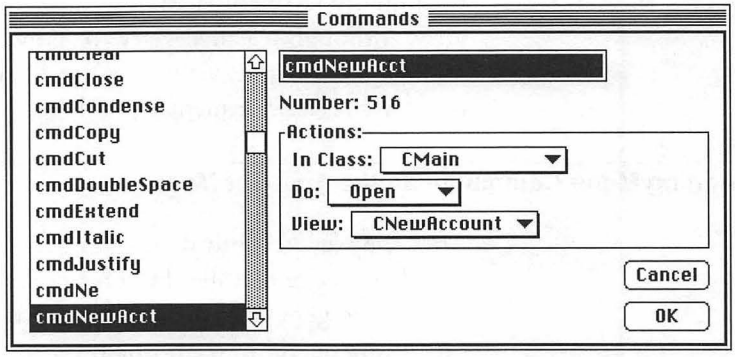

Figure 4-17
cmdNewAcct command added to Account menu

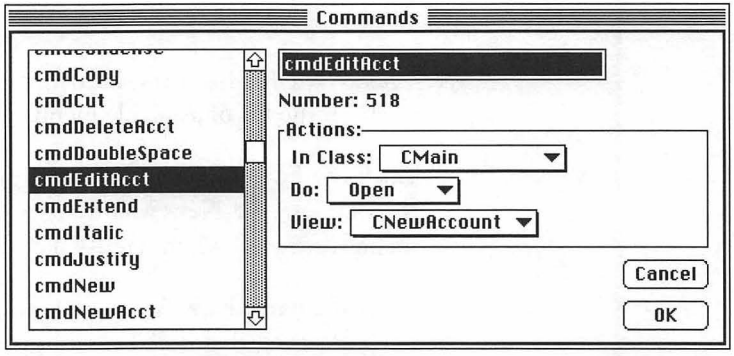

Figure 4-18
cmdEditAcct command added to Account menu

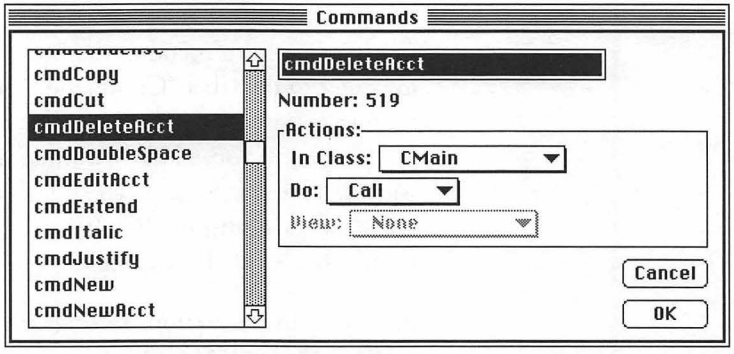

Figure 4-19
cmdDeleteAcct command added to Account menu

6. Repeat the actions described in the foregoing steps 3 and 4 for a new command named `cmdDeleteAcct`, and set the Actions for that command to be as shown in Figure 4-19.

7. Finally, create one more new command in the Account menu, and set its name to a single "dash" character (that is, '–'). This

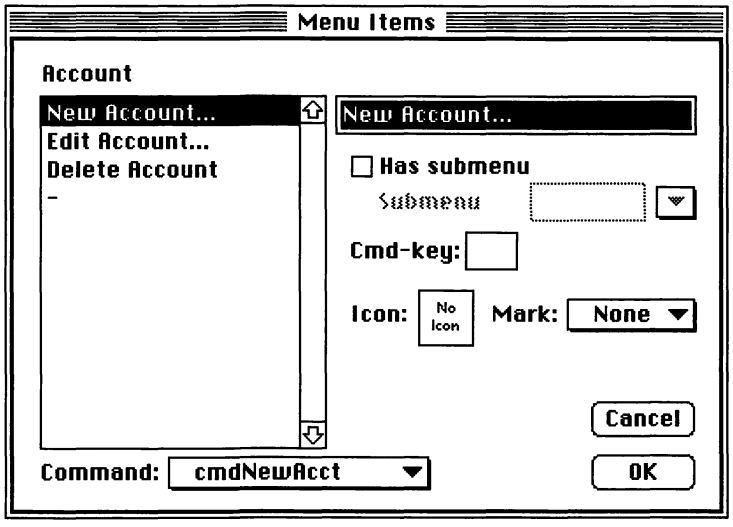

Figure 4-20
Completed list of menu commands for the Account menu

will cause a dividing line to be drawn in the menu at that point. No Actions are associated with a dividing line. The line will serve to separate the foregoing commands from the names of new accounts that are added to the menu when the user creates them. The final result is as shown in Figure 4-20.

8. When the foregoing steps are complete, click OK in the Menu Items dialog, and then click OK in the Menus dialog to complete the process of adding menu commands to the Account menu.

Generating Code and Viewing the Results

After creating the user interface elements described in the foregoing sections, choose the **Generate All** command from VA's project menu (the Symantec Project Manager "star field" icon), and let VA update your project with the generated files (it may have asked you to save the file before generating code and you should do so). You can now **Quit** from the VA.

The default-generated code for the CApp and CMain classes is the same as was described in Chapters 2 and 3. In this section, we will focus on the functionality of the newly generated code. A later section will discuss the code itself and what we need to do to implement the Account view fully.

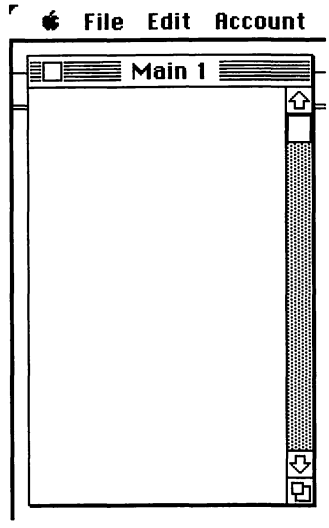

Figure 4-21
Application running, with Main window and menus shown

Figure 4-22
Choosing New Account from the Account menu

When the VA-generated code is compiled and executed, the structure of the application is as shown in Chapter 3, Figure 3-1. Only the application and main document objects have been created. In fact, you should see the main window and menus, as depicted in Figure 4-21 (the figure shows only a portion of the screen).

The VA-generated code also makes provision for opening the NewAccount dialog, so you can display the dialog if you pull down the **Account** menu and choose the **New Account** command, as shown in Figure 4-22.

When the **New Account** command is chosen, the VA-generated code (in the x_CMain class) creates a CNewAccount object, initializes the object, and then calls the DoModalDialog function to "run" the dialog. The result of doing so is shown in Figure 4-23 (with values entered into the Edit Text fields and the Savings radio button selected).

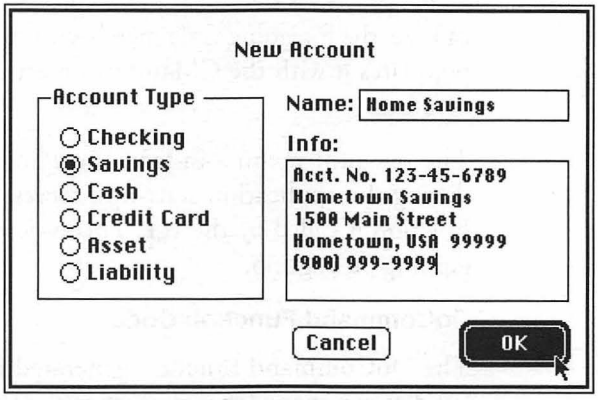

Figure 4-23
Running the New Account dialog

Although clicking the OK button in the foregoing dialog dismisses it, nothing happens after that. We have not yet added the functions that implement what to do with the data in the dialog once it has been dismissed.

Examining the Generated Code in x_CMain

Most of the functionality described in the foregoing section is implemented in the x_CMain class, including the display of the Main window and the handling of the **Account** menu commands. This section will concentrate on showing only the relevant sections of that code.

MakeNewWindow Function Code

The code for the MakeNewWindow function is as follows:

```
void x_CMain::MakeNewWindow(void)
{
   itsWindow = TCLGetNamedWindow("\pMain", this);

   itsMainPane = (CPane*) TCLGetItemPointer(itsWindow, 0);

      // Initialize pointers to the subpanes in the window

  fMain_Accounts = (CArrayPane*) itsWindow
     ->FindViewByID(kMain_AccountsID);
  ASSERT(member(fMain_Accounts, CMainList));
}
```

The MakeNewWindow function in the x_CMain class is called as a result of the `cmdNew` command being sent to the current gopher during the StartupAction function in the CApplication class. The

whole process is described in Chapter 3, on pages 49–57. As you can see, the foregoing code merely creates a new window and then populates it with the CMainList object that will hold our list of accounts.

The Account menu—in fact, the whole menu bar—is created during the application start-up process, when the SetUpMenus function is called by the TCL. This is described in Chapter 2, beginning on page 35.

DoCommand Function Code

The DoCommand function generated into the x_CMain class handles commands that are sent to the current gopher, including those that begin with the user's choice of one of the commands from the **Account** menu. The code for this function is as follows:

```
void x_CMain::DoCommand(long theCommand)
{
   switch (theCommand)
   {
      case cmdNewAcct:
         DoCmdNewAcct();
         break;
      case cmdEditAcct:
         DoCmdEditAcct();
         break;
      case cmdDeleteAcct:
         DoCmdDeleteAcct();
         break;
      default:
         CDocument::DoCommand(theCommand);
   }
}
```

The foregoing code has cases that handle the three new commands that we added to the **Account** menu. In each case, the DoCommand function calls a separate function to implement the command. If the command is not any of these, then the base class DoCommand function is called to handle the command.

DoCmdNewAcct Function Code

The DoCmdNewAcct function in the x_CMain class is called to handle the `cmdNewAcct` command. The code is as follows:

```
void x_CMain::DoCmdNewAcct()
{
   CNewAccount   *dialog;

   // Respond to command by opening a dialog
```

```
    dialog = TCL_NEW(CNewAccount,());
    dialog->ICNewAccount(this);

    dialog->DoModalDialog(cmdNull);
    TCLForgetObject(dialog);
}
```

The foregoing VA-generated code creates a new CNewAccount object, initializes the object, calls DoModalDialog to "run" the dialog, and then disposes of the object. So although the basic functionality of creating the dialog and running it is provided by this code, it does nothing to implement what should be done with the data once the dialog has been filled in and dismissed. We will need to override this function in our custom code for the CMain class.

DoCmdEditAcct Function Code

The VA-generated code for the DoCmdEditAcct function is identical to what was presented for the DoCmdNewAcct function. This is because we chose for this code to also open the NewAccount dialog. The code is as follows:

```
void x_CMain::DoCmdEditAcct()
{
    CNewAccount*dialog;

    // Respond to command by opening a dialog

    dialog = TCL_NEW(CNewAccount,());
    dialog->ICNewAccount(this);

    dialog->DoModalDialog(cmdNull);
    TCLForgetObject(dialog);
}
```

DoCmdDeleteAcct Function Code

The code generated for the `cmdDeleteAcct` command is somewhat different from the foregoing. We don't need to open the dialog to delete an account, so we chose simply to call a function to handle the command. Because the VA doesn't quite know what to do in this case, it indicates that the subclass (in our case, that's CMain) *must* override the function. The code is as follows:

```
void x_CMain::DoCmdDeleteAcct()
{
    // Subclass must override this function to
    // handle the command
}
```

UpdateMenus Function Code

In addition to the code to create the Main window and handle the menu commands that we just examined, the VA generates an override of the document's UpdateMenus function. When the user clicks on the menu bar, the default action by the CBartender object is to disable all menu commands and then call the UpdateMenus command for the current gopher (which, in this case, is the CMain object) to enable the menu commands that are appropriate for the current context. Because CMain does not contain an override for the UpdateMenus command, the version inherited from the x_CMain class is executed. The code for this function is as follows:

```
void x_CMain::UpdateMenus()
{
   CDocument::UpdateMenus();
   gBartender->EnableCmd(cmdNewAcct);
   gBartender->EnableCmd(cmdEditAcct);
   gBartender->EnableCmd(cmdDeleteAcct);
}
```

As is evident, the foregoing code first calls its CDocument base class function (to update menus that it and other members of the chain of command manage—this includes the **File** and **Edit** menus), and then it calls the EnableCmd function for the global gBartender object to enable all of the **Account** menu commands.

The foregoing is not quite the behavior we have in mind for our application. It doesn't make sense to edit or delete an account when none is selected in the Main window, so we must override the UpdateMenus function in our CMain class to enable only the commands that are appropriate in the current context.

What About the Rest of the Generated Code?

Although we created the NewAccount dialog in the VA and generated code to implement it, we will discuss the dialog and its associated (CNewAccount, x_CNewAccount) code later when we cover the custom code we need to add to implement the creation and editing of accounts fully. In addition, even though we spent a lot of time creating the appearance of the Account view, we will also wait to discuss of the CAccount, x_CAccount, CAccountList, and x_CAccountList classes until we have covered how the user interface features should function.

Making the Business View Fully Functional

The process of making a new view functional often consists of writing quite a bit of custom code in the various derived class modules that the VA generates. In a large sense, the VA has provided very little functionality in the derived class code, except for what is obviously necessary. Much of the content of both the CApp and CMain classes has been covered fully in Chapters 2 and 3. In this section, we will concentrate on describing the *additional* custom code that is needed to implement the Business View fully.

Before delving into the code, let's take a minute to discuss how the Business View should operate, from the user's perspective. Some features that we'd like to see are the following:

- Because the user will most likely want to open an existing file of accounts when the program is launched, the creation of a new Main window should be suppressed. If a new file of accounts is to be created, the user should choose the **New** command from the **File** menu to do so.

- When no accounts are present, or no account is selected in the Main window's account list, then the **Edit Account** and **Delete Account** commands in the **Account** menu should be disabled. The **New Account** command should always be enabled.

- When the **New Account** command is chosen and the NewAccount dialog is displayed and then dismissed, the application should determine whether the OK or Cancel button was clicked. If the dismisser was the Cancel button, then no action is required; however, if OK was clicked, then the application should create a new CAccount object, save the other features of the account in the CAccount object, open a window with the account's name in its title, and add the account name both to the list of accounts in the Main window and also to the bottom of the **Account** menu. Newly added accounts should also be associated with unique identifiers that can also serve as their command IDs in the **Account** menu.

- If an account name is selected (hilited) in the Main window, then both the **Edit Account** and **Delete Account** commands should be enabled when the UpdateMenus function is called.

- If the **Edit Account** command is chosen for a selected account, then the NewAccount dialog should be opened and its settings should be made to correspond with those saved in the CAccount object. When the dialog is dismissed with the OK button, the settings in the CAccount object should be changed to correspond with the settings in the dialog. If the dialog is dismissed with the Cancel button, the account's settings should remain the same as specified previously.

- If the **Delete Account** command is chosen for a selected account, an Alert should be displayed, allowing the user to decide whether deletion of the account object, its name in both the Main view and the **Account** menu, and its transactions is really wanted. If not, then the command should be ignored.

- Each entry in the Account view should indicate the appearance of having two rows of data, with multiple columns, as implied by the caption information that we designed to head the scrolling CAcctList pane in the view.

Ideally, we should carry the foregoing specifications further and design the behavior of making entries into the account and the loading and saving the account data; however, my intention in the sections that follow is to show the fundamental concepts behind the implementation of the foregoing-specified features—although I will show some of the necessary code, my main goal is to show how to use the features built into the TCL to support our efforts. I'll conclude this section with suggestions on what remaining features would be worthwhile and how these could be implemented.

Examining the Custom Code in the CApp Class

The CApp class is derived from the x_CApp class to hold the custom code for application-wide features. In this regard, we need to ensure that a new Main window is not created when the application is launched, and also that account IDs are unique when new accounts are created.

Adding a New Instance Variable

I have added a new instance variable to the CApp class, called `theNextAccount`, whose purpose is to hold the next available unique account identifier. The value also serves as the command

ID for accounts newly added to the **Account** menu. The additions to the **CApp.h** header file are as follows:

```
public:
   long  theNextAccount;     // the next available account number

#define kFirstAcctCmdID    2000
```

I have declared `theNextAccount` as a `long` value only because command IDs are passed around to various functions in that format. The `#define` statement—placed at the end of the file—provides an initial value for the `theNextAccount` variable. The value '2000' was chosen arbitrarily, so as not to conflict with any command numbers predefined for the TCL, or any that will be generated in the future by the VA.

Newly Added Initialization Features

Newly added code to set the value for `theNextAccount` variable and suppress the creation of the Main view on start-up is contained in the ICApp function, as follows (copious comments elided):

```
void CApp::ICApp()
{
   Ix_CApp(4, 24000L, 20480L, 2048L);

   // initialize the first account ID and
   // suppress the creation of the Main window
   // on startup.
   theNextAccount = kFirstAcctCmdID;
   newWindowOnStartup = FALSE;
}
```

The `kFirstAcctCmdID` constant is defined in the **CApp.h** header file, as discussed earlier.

Examining Custom Code in the CMain Class

The CMain class is the document object for the application. As such, it is both the "keeper" of the document's data and the manager of its views and the menu commands and associated actions for the views. When the user chooses the **New** command from the **File** menu, a new CMain document object is created and its view (list of accounts) is displayed.

CMain Header File Additions

I have added new member variables and function declarations to the **CMain.h** header file. These are as follows:

```
#define kAccountMenuID 128    // VA-assigned Account menuID
#define kOKCancelAlert 130    // Standard Yes/No/Cancel Dialog
#define kAlertOK         1    // OK button is control ID #1
```

The foregoing definitions are referenced in the source code of the CMain class.

```
public:
   CArray     *itsAccounts;      // list of CAccount objects
   CArray     *itsViewArray;     // account name pointer array
   CNewAccountData itsAcctData;  // new account dialog settings
```

The foregoing CArray member variables refer to the various arrays of data managed by the CMain object. The `CNewAccountData` variable is an instance of the structure defined in the CNewAccount class to hold the settings for the NewAccount dialog.

```
   virtual void MakeNewWindow(void);  // newly added override
   virtual void DoCmdNewAcct(void);   // newly added override
   virtual void DoCmdEditAcct(void);  // newly added override
   virtual void DoCmdDeleteAcct(void);// newly added override
   virtual void UpdateMenus(void);    // newly added override
   virtual void DoCommand (           // new override
                long theCommand);     //   command value
   virtual void GetAcctSettings (     // transfer to/from dlg
              Boolean fromDocument,   //  transfer direction
              CNewAccountData *data); // settings pointer
   virtual void MakeDefaultSettings(void); // default settings
```

The foregoing member function declarations were added to the **CMain.h** header file. All are public functions.

ICMain Function Code

After the CMain object is created, its ICMain function is called. The ICMain function is a good place to create the CArray object that will hold our list of CAccount (view) objects, as well as the CArray object to hold the strings that contain the names of the accounts that are displayed in the Main view. The newly modified version of this code is as follows:

```
void CMain::ICMain()
{
   Ix_CMain();

   // create the array of accounts (itsAccounts) and the array
   // of account names (itsViewArray)
   itsAccounts = TCL_NEW(CArray, (sizeof (CAccount *)));
   itsViewArray= TCL_NEW(CArray, (sizeof (Str255 *)));
}
```

In the foregoing code, the `itsAccounts` and `itsViewArray` variables are both declared as pointers to CArray objects in the **CMain.h** header file. Both arrays are setup to contain pointers to objects (CAccount and STR255).

MakeNewWindow Override Function Code

One of the first things I had to do to make sure that the `itsMainPane` variable was set properly was to override the MakeNewWindow function (generated by VA in the x_CMain class). The newly created code is as follows:

```
void CMain::MakeNewWindow()
{
   x_CMain::MakeNewWindow();
   itsMainPane = fMain_Accounts;
   fMain_Accounts->SetArray(itsViewArray, TRUE);
}
```

The foregoing code begins by calling the base class function to create the window, but then changes the value of `itsMainPane` to correspond to the CMainList object (`fMain_Accounts`). In addition, the array associated with the `fMain_Accounts` object is specified. We have indicated that the CMainList object is the owner of the array so that when the window is closed, the array will be deleted automatically.

DoCommand Override Function Code

I have overridden the DoCommand function to provide behavior that is specific to the user's choice of an account name command from the **Account** menu. The code for the override of this function is as follows:

```
void CMain::DoCommand(long theCommand)
{
   long nextAccount = ((CApp *)gApplication)->theNextAccount;
   if (theCommand >= 2000L && theCommand < nextAccount)
```

```
    {
       // it's one of our newly added Account menu commands.
       // Process the command by bringing the associated window
       // to the front, or by showing the window if it's hidden.

       CAccount *theAccount;
       CArrayIterator Iter (itsAccounts, kStartAtBeginning);
       while (Iter.Next (&theAccount))
       {
          if (theAccount->itsAcctID == theCommand)
          {
             // found the account, so get its window to the front

             if (!theAccount->itsWindow->IsVisible())
             {
                theAccount->itsWindow->Show();
             }
             theAccount->itsWindow->Select();
          }
       }
    }
    else
    {
       x_CMain::DoCommand(theCommand);
    }
}
```

When the user chooses an account name that has been installed into the **Account** menu, the window for that account has already been created, but may or may not be hidden from view. If the window is hidden, then the foregoing code locates and makes the window visible. If the window is already open, it is selected to bring it to the front.

DoCmdNewAcct Override Function Code

I have overridden the DoCmdNewAcct function that VA generated into the x_CMain class in order to add the functionality necessary to realize the objectives listed on page 123 when **New Account** is chosen. The code is as follows:

```
void CMain::DoCmdNewAcct()
{
   CNewAccount   *dialog;

   // override of DoCmdNewAcct function in x_CMain
   // initialize the "itsAcctData" to default values and then
   // run the dialog.

   MakeDefaultSettings();
   dialog = TCL_NEW(CNewAccount,());
   dialog->ICNewAccount(this);
   dialog->BeginDialog();
   if (dialog->DoModalDialog(cmdNull) == cmdOK)
   {
      // dialog was dismissed with OK button, so we need to create
      // a new CAccount object, initialize the object with the
      // document as its supervisor, transfer the characteristics
```

```
        // specified by the user, add the account to the document's
        // list, and then call the account's function to initialize
        // the array.

        CAccount *anAccount = TCL_NEW(CAccount, ());
        anAccount->ICAccount (this);
        anAccount->itsSettings = itsAcctData;
        itsAccounts->Add (&anAccount);
        anAccount->CreateNewEntries();

        // the next step is to allocate space for the new account
        // name string, store the string in memory, and then add
        // the string pointer to the document's list window array.

        short nNameLength = anAccount
           ->itsSettings.fNewAccount_AcctName[0];
        StringPtr aPtr = new unsigned char[nNameLength + 1];
        TCLpstrcpy (aPtr, anAccount
           ->itsSettings.fNewAccount_AcctName);
        itsViewArray->Add (&aPtr);

        // finally, we append the account name to the Account menu
        // by calling the Bartender's InsertMenuCmd function with
        // the next available account command ID and the address
        // of the name string.

        long itsID = ((CApp *)gApplication)->theNextAccount++;
        anAccount->itsAcctID = itsID;
        gBartender->InsertMenuCmd (itsID, aPtr, kAccountMenuID, 999);
    }
    TCLForgetObject(dialog);
}
```

The comments in the foregoing explain the functions performed by the various sections of code. Basically, if the NewAccount dialog is dismissed with the OK button, we need to create a new CAccount object, initialize the object, and add the object's pointer to our array of accounts (`itsAccounts`). Then we need to allocate memory for the name of the account, store the account name in the allocated area and add its pointer to the array associated with the Main view's list of account names (`itsViewArray`). Finally, we need to add the account name and ID to the **Account** menu by using the CBartender's InsertMenuCmd function.

DoCmdEditAcct Override Function Code

I have also overridden the DoCmdEditAcct function (generated by the VA into the x_CMain class) so that we can provide the additional functionality needed when the user chooses the **Edit Account** function. The newly added code is as follows:

```
void CMain::DoCmdEditAcct()
{
    CNewAccount   *dialog;          // new account dialog
    Cell          selAcct = {0, 0}; // selected account
```

```
    StringPtr     selName = NULL;    // name of selected account

    // begin by getting the name of the selected account

    TCL_ASSERT(((CAcctList *)itsMainPane)->GetSelect(TRUE, &selAcct));
    itsViewArray->GetArrayItem (&selName, selAcct.v + 1);
    TCL_ASSERT (selName);

    // next, find the matching account in the "itsAccounts" array

    Boolean found = FALSE;
    CAccount *anAccount;
    CNewAccountData acctData;
    CArrayIterator Iter (itsAccounts, kStartAtBeginning);
    while (Iter.Next (&anAccount))
    {
       acctData = anAccount->itsSettings;
       if (TCLpstrcmp (selName, acctData.fNewAccount_AcctName) == 0)
       {
          // account found, so use its settings for the dialog

          itsAcctData = acctData;
          found = TRUE;
          break;
       }
    }
    TCL_ASSERT (found);// fail if not found

    // respond to command by opening a dialog

    dialog = TCL_NEW(CNewAccount,());
    dialog->ICNewAccount(this);
    if (dialog->DoModalDialog(cmdNull) == cmdOK)
    {
       // assume that settings were changed and update the
       // existing account.

       anAccount->itsSettings = itsAcctData;
    }
    TCLForgetObject(dialog);
}
```

The foregoing code commences by locating the account name for the selected account in the `itsViewArray` array. Once the name is found, it is used to locate the CAccount object, using an instance of CArrayIterator to iterate through the objects in the `itsAccounts` array variable. When the matching account is found, its settings (`acctData`) are saved temporarily in the document's `itsAcctData` structure and the NewAccount dialog is displayed (the settings in the document's `itsAcctData` are used to preset the controls and values in the dialog). If the dialog is dismissed with the OK button, then the account's settings are replaced with the final values in the dialog object. (*Note:* The foregoing logic permits the user to change an account from one type to another without any quarrel. In a robust version of the

application, it might be useful to restrict the type of changes that the user can make to the various accounts.)

DoCmdDeleteAcct Override Function Code

I have also overridden the DoCmdDeleteAcct function (generated by the VA into the x_CMain class). The newly added code provides the functionality we require when the user chooses the **Delete Account** command from the **Account** menu. The code is as follows:

```
void CMain::DoCmdDeleteAcct()
{
   Cell          selAcct = {0, 0};  // selected account
   StringPtr     selName = NULL;    // name of selected account
   CTransaction  *aTrans;           // one transaction
   Boolean       found;             // search result

   // to delete an account means to remove all of its
   // transactions, the account entry, and also the menu command.
   // Start by finding the name of the account.

   TCL_ASSERT(((CAcctList *)itsMainPane)->GetSelect(TRUE,&selAcct));
   itsViewArray->GetArrayItem (&selName, selAcct.v + 1);
   TCL_ASSERT (selName);

   // next we inquire of the user whether he/she really wants to
   // delete the account and all of its transactions.

   ParamText (selName, 0, 0, 0);
   if (CautionAlert (kOKCancelAlert, NULL) == kAlertOK)
   {
      // user really wants to delete the account, so find the
      // account and delete all of its transactions.

      CNewAccountData acctData;
      CAccount *anAccount;
      long acctID;

      CArrayIterator Iter (itsAccounts, kStartAtBeginning);
      found = FALSE;
      while (Iter.Next (&anAccount))
      {
         acctData = anAccount->itsSettings;
         if(TCLpstrcmp (selName,acctData.fNewAccount_AcctName) == 0)
         {
            // account was found, so get its ID, delete it from
            // the list, remove the menu command that relates to
            // the account, and then delete the selected name.

            itsAccounts->DeleteItem (Iter.GetCursor());
            acctID = anAccount->itsAcctID;
            gBartender->RemoveMenuCmd (acctID);
            itsViewArray->DeleteItem (selAcct.v + 1);
            found = TRUE;

            // also, because the account is being deleted, we need
            // to delete all of its transactions.

            CArray *theTransactions = anAccount->itsTransactions;
            int numItems = theTransactions->GetNumItems();
```

```
            for (int count=0; count < numItems; count++)
            {
               theTransactions->GetArrayItem (&aTrans, 1L);
               TCLForgetObject (aTrans);
               theTransactions->DeleteItem (1L);
            }
            break;// break out of the while loop
         }
      }
      TCL_ASSERT (found);


      // finally, we get the account's window, make sure it's
      // visible, select it, and then send it the cmdClose
      // command. Note that this also disposes of the CAccount
      // object.

      TCL_ASSERT (anAccount->itsWindow);
      if (!anAccount->itsWindow->IsVisible)
      {
         anAccount->itsWindow->Show();
      }
      anAccount->itsWindow->Select();
      gGopher->DoCommand (cmdClose);
   }
}
```

The procedure for deleting an account, as demonstrated by the foregoing code, is not simple. The first step is to display a Caution Alert to the user, indicating that if the deletion process is allowed to proceed, the account and all of its transactions will be deleted. Only if the user clicks the OK button in that alert do we continue the process; otherwise, the function just returns at its bottom.

The Caution Alert (`kOKCancelAlert`) was created with ResEdit and the `kOKCancelAlert` symbol was added to the header file for the CMain class (see page 126). The appearance of the alert is shown in Figure 4-24. Note that the alert contains a single parameter (indicated by the ^0 placeholder in the text).

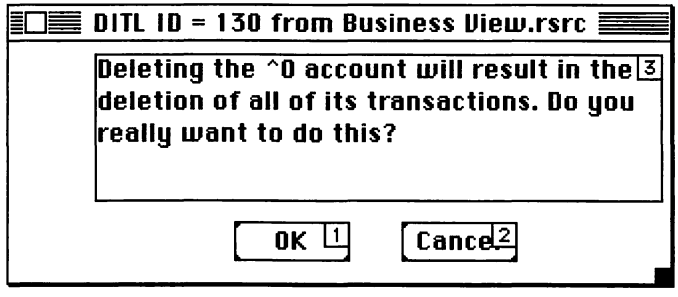

Figure 4-24
ALRT 130 resource appearance

If the user chooses to proceed with the deletion of the account and its transactions, then the foregoing code locates the CAccount object corresponding to the selected account name and proceeds to delete its entry from the `itsAccounts` array; then the menu command corresponding to the account is deleted by calling the Bartender's RemoveMenuCmd function with the account's ID (that is also its command ID); then the account's name is deleted from the list of accounts in the Main view by removing the entry from the `itsViewArray` array; then all of the account's transactions are deleted; and finally the account's window is made visible (if not) and is activated and closed. Closing the account's window results in deletion of the CAccount object.

MakeDefaultSettings Function Code

The DoCmdNewAcct function (in response to the user's choice to create a new account) calls the MakeDefaultSettings function to create the default settings for the NewAccount dialog. The code for this function is as follows:

```
void CMain::MakeDefaultSettings ()
{
   itsAcctData.fNewAccount_Rect2 = 0;
   itsAcctData.fNewAccount_CheckingRadio = 1;
   itsAcctData.fNewAccount_SavingsRadio = 0;
   itsAcctData.fNewAccount_CreditRadio = 0;
   itsAcctData.fNewAccount_CashRadio = 0;
   itsAcctData.fNewAccount_AssetRadio = 0;
   itsAcctData.fNewAccount_LiabilityRadio = 0;
   TCLpstrcpy (itsAcctData.fNewAccount_AcctName, "\p");
   TCLpstrcpy (itsAcctData.fNewAccount_InfoText, "\p");
}
```

The foregoing code fills the various fields in the itsAcctData structure with default settings.

ProviderChanged Override Function Code

The functional specifications for the application stated that when the user clicked on an account name in the Main view, then the corresponding account window should be brought to the front. To accomplish this task, I have created an override for the ProviderChanged function (inherited from the CDirector base class) and have written code to test whether the selection in the Main view has changed. If the user clicks the name of a new account, then the SelectRect function of the CTable class calls the BroadcastChange function, that results in (eventually) the

CDirector-derived ProviderChanged function being called. The code for the ProviderChanged function is as follows:

```
void CMain::ProviderChanged(CCollaborator *aProvider, long
reason, void* info)
{
   Cell aCell = {0, 0};
   StringPtr selName;

   if (aProvider == fMain_Accounts)
   {
      if (reason == tableSelectionChanged)
      {
         if (fMain_Accounts->GetSelect (TRUE, &aCell))
         {
            itsViewArray->GetArrayItem (&selName, aCell.v + 1);
            Boolean found = FALSE;
            CAccount *anAccount;
            CNewAccountData acctData;
            CArrayIterator Iter(itsAccounts, kStartAtBeginning);
            while (Iter.Next (&anAccount))
            {
               acctData = anAccount->itsSettings;
               if (TCLpstrcmp (selName,
                  acctData.fNewAccount_AcctName)==0)
               {
                  // account found, so use activate the view

                  if (!anAccount->itsWindow->IsVisible())
                  {
                     anAccount->itsWindow->Show();
                  }
                  anAccount->itsWindow->Select();
                  found = TRUE;
                  break;
               }
            }
            TCL_ASSERT (found);
            return;
         }
      }
   }
   x_CMain::ProviderChanged(aProvider, reason, info);
}
```

The foregoing code first checks whether the "provider" of the message is the `fMain_Accounts` object (that is the CMainList pane of our Main view). If so, then the code continues by checking whether the reason for the call is `tableSelectionChanged`. If so, then the code continues by locating the account whose name is selected, showing its window if it is hidden, and then selecting the window to make it frontmost. If the provider or reason arguments to the function do not satisfy our criteria, then the base class ProviderChanged function is called.

Examining the Custom Code in the CMainList Class

The CMainList class is generated by the VA to be derived from the CArrayPane class so that we can override the GetCellText function to supply the account names for display in the Main view. The generated code for this class contains only GetFrom and PutTo functions in both the base class (x_CMainList) and derived class (CMainList) source files.

GetCellText Function Code

I have added the GetCellText (override) function to the CMainList class so that we can supply the text that is written into the scroll list in the Main view. The custom code is as follows:

```
void CMainList::GetCellText(Cell aCell, short width,
   StringPtr itsText)
{
   StringPtr aPtr;
   CArray *itsArray = GetArray();
   itsArray->GetArrayItem (&aPtr, (long) aCell.v + 1);
   TCLpstrcpy (itsText, aPtr);
}
```

The foregoing code accesses the CArray object associated with the CMainList (CArrayPane-derived) object and then retrieves the string that is associated with the `aCell` argument of the call. The string is copied to the `itsText` pointer to complete the task. This is all that is required to cause the account names to appear in the scroll list of the Main view.

Examining Newly Added CTransaction Class Code

When we started customizing the CAccount class, we realized that we needed to create some entries in each new account's view so that its scroll bar would be active and the separators for the entries would be drawn. We designed a CTransaction class to hold the transactions for an account and we wrote some code to implement access functions for all of the member variables in that class.

CTransaction Class Header File

The class declaration for the CTransaction class is contained in the **CTransaction.h** header file, whose contents are as follows:

```
/************************************************************
 CTransaction.h

   Header File For CTransaction Class

   Copyright © 1995 Richard O. Parker. All rights reserved.


************************************************************/
#pragma once
#include <OSUtils.h>

typedef enum
{
   EMPTY,
   ACTIVE,
   MODIFIED,
   COMPLETE

} AcctStatus;

class CTransaction TCL_AUTO_DESTRUCT_OBJECT
{
public:

   TCL_DECLARE_CLASS

   CTransaction (long theAccountID);// constructor w/account ID
   CTransaction (void);             // default constructor

   short        GetAccountID (void);
   AcctStatus   GetStatus (void);
   CString      GetDate (void);
   CString      GetNumber (void);
   CString      GetDescription (void);
   CString      GetCategory (void);
   CString      GetInfo (void);
   long         GetPayment (void);
   long         GetDeposit (void);
   long         GetBalance (void);

   void         SetStatus (AcctStatus n);
   void         SetDate (DateTimeRec dt);
   void         SetNumber (CString& s);
   void         SetDescription (CString& s);
   void         SetCategory (CString& s);
   void         SetInfo (CString& s);
   void         SetPayment (long v);
   void         SetDeposit (long v);
   void         SetBalance (long v);

private:

   short        accountID;
   AcctStatus   status;
   DateTimeRec  date;
   CString      number;
   CString      description;
   CString      category;
   CString      info;
   long         payment;
   long         deposit;
   long         balance;
};
```

The foregoing class declaration shows that all of the member variables for the class are private. "Access functions" were declared to get and set the values of all of the variables and to provide the greatest flexibility for future changes.

CTransaction Class Source Code

The **CTransaction.cp** source file implements the access functions discussed in the foregoing as follows:

```
/************************************************************
 CTransaction.cp

   Transaction Data For Accounts

   Copyright © 1995 Richard O. Parker. All rights reserved.

   There are a lot of "access functions" in this class. Rather
   than publish its internal structure, we publish the functions
   by which the member variables are accessed.
************************************************************/
#include "CTransaction.h"

#include <TCLpstring.h>

TCL_DEFINE_CLASS_M0(CTransaction);

/****CONSTRUCTION/DESTRUCTION  FUNCTIONS****/

CTransaction::CTransaction(long theAccountID)
{
   accountID = theAccountID;
   TCL_END_CONSTRUCTOR
}

CTransaction::CTransaction (void)
{
   accountID = -1;
   TCL_END_CONSTRUCTOR
}

/****ACCESS FUNCTIONS FOR GETTING VALUES****/

short CTransaction::GetAccountID (void)
{
   return accountID;
}

AcctStatus CTransaction::GetStatus (void)
{
   return status;
}

CString CTransaction::GetDate (void)
{
   CString dateString;
   char s[10];

   sprintf (s, "%02d/%02d/%02d", date.month, date.day,
      date.year - 1900);
   dateString = s;
   return dateString;
```

```
}

CString CTransaction::GetNumber (void)
{
   return number;
}

CString CTransaction::GetDescription (void)
{
   return description;
}

CString CTransaction::GetCategory (void)
{
   return category;
}

CString CTransaction::GetInfo (void)
{
   return info;
}

long CTransaction::GetPayment (void)
{
   return payment;
}

long CTransaction::GetDeposit (void)
{
   return deposit;
}

long CTransaction::GetBalance (void)
{
   return balance;
}

/****ACCESS FUNCTIONS FOR SETTING VALUES****/

void CTransaction::SetStatus (AcctStatus n)
{
   status = n;
}

void CTransaction::SetDate (DateTimeRec dt)
{
   date = dt;
}

void CTransaction::SetNumber (CString& s)
{
   number = s;
}

void CTransaction::SetDescription (CString& s)
{
   description = s;
}

void CTransaction::SetCategory (CString& s)
{
   category = s;
}

void CTransaction::SetInfo (CString& s)
{
   info = s;
```

```
}

void CTransaction::SetPayment (long v)
{
   payment = v;
}

void CTransaction::SetDeposit (long v)
{
   deposit = v;
}

void CTransaction::SetBalance (long v)
{
   balance = v;
}
```

The foregoing functions are each very simple; however, rather than have full access to the transaction variables, programmers will have to use the access functions. Developing new classes in this way—especially classes that have no inheritance relationship to the remainder of the code—is a good approach, and one that isolates the data structures from the remainder of the user's code and allows us the flexibility to change these, should we decide to do so.

Examining Custom Code in the CAccount Class

When the user chooses to create a new account, the DoCmdNewAcct function (described on page 120) creates a new CAccount object and then initializes that object. Each new view in the application has a director that is responsible for managing the view's window and handling commands that pertain to various elements of the view.

The CAccount class is the CDirector-derived class for the Account view. The CMain (CDocument-derived) class is the "supervisor" of the CAccount director, which means that functions in the CAccount class can access public member functions and member variables in the document. Also, because code has been added to the CMain class to keep a record of the CAccount objects, the document can communicate with any of the public member functions and variables in the CAccount class. This provides two-way communication between the objects.

CAccount Class Header File

The **CAccount.h** header file contains the declarations for the CAccount class. The contents of that file are as follows:

```
/************************************************************
 CAccount.h

   CAccount Window Director Class

   Copyright © 1994 Richard O. Parker. All rights reserved.

   Generated by Visual Architect™ 10:58 AM Mon, Oct 31, 1994

   This file is only generated once. You can modify it by filling
   in the placeholder functions and adding any new functions you
   wish.

   If you change the name of the document class, a fresh version
   of this file will be generated. If you have made any changes
   to the file with the old name, you will have to copy those
   changes to the new file by hand.
************************************************************/

#pragma once
#include "x_CAccount.h"
#include "x_CNewAccount.h"

class CDirectorOwner;

class CAccount : public x_CAccount
{
public:

   TCL_DECLARE_CLASS

   // Insert your own public data members here

   CNewAccountData itsSettings;       // account settings
   long            itsAcctID;         // account command ID
   CArray          *itsTransactions;  // list of transactions

   void ICAccount(CDirectorOwner *aSupervisor);

   void ProviderChanged(              // newly added override
      CCollaborator *aProvider,       //   who's the provider?
      long reason,                    //   for what reason?
      void* info);                    //   other info.
   void DoCommand(long theCommand);   // override

   void CloseWind (CWindow *theWindow);// newly added override
   void CreateNewEntries (void);      // create account entries
};
```

As is evident in the foregoing class declaration, I have added three new "public" variables. The `itsSettings` variable holds the settings for each CAccount object that is created (account name, type of account, and miscellaneous information). The `itsAcctID` variable serves double duty; it is both a unique identifier for the account and the command ID used to address the account in the **Account** menu. The third public variable is a CArray object that is intended to hold all of the transactions for the CAccount object.

In addition to adding new public member variables, I have also added public member functions to override functions in the base classes from which the CAccount class inherits its behavior. The following sections display and discuss all of the newly added source code.

Preprocessor and Compiler Directives

I have added some preprocessor and compiler directives to what the VA has generated automatically into the **CAccount.cp** source file. The newly added code is as follows:

```
#include "CTransaction.h"                // newly added
#include "CAcctList.h"                   // newly added

extern CApplication   *gApplication;     // The application
extern CDecorator     *gDecorator;       // The Window Decorator
extern CDesktop       *gDesktop;         // The visible Desktop
```

In addition to including the header files for the newly added CTransaction class, I have also specified that the application, decorator, and bartender objects are defined externally as global object pointers to the specified classes.

ICAccount Member Function Code

The ICAccount function is called by the DoCmdNewAcct function in the CMain class when a new CAccount object has been created. The function of the ICAccount code is to perform any additional initialization functions, prior to performing any other operations on the CAccount object. The code is as follows:

```
void CAccount::ICAccount(CDirectorOwner *aSupervisor)
{
   // Initialize data members here that must be set up
   // before BeginData is called

   x_CAccount::Ix_CAccount(aSupervisor);

   // create the array to hold the CTransaction objects and
   // associate it with the CAcctList object.

   itsTransactions = TCL_NEW(CArray, (sizeof (CTransaction *)));
   fAccount_Entries->SetArray (itsTransactions, TRUE);
}
```

The VA-generated code for the ICAccount function includes only the call to the Ix_CAccount function in the x_CAccount base class. That function performs the task of creating the CAccount

window (by calling its MakeNewWindow function) and then returns. I have added the code to create the `itsTransactions` CArray object and then call the SetArray function to specify the array for the `fAccount_Entries` object (CAcctList), which is the list pane for the CAccount window. By making the connection between the list and its array, any changes made to the array will cause the list to be redrawn (this is by virtue of the collaboration mechanism in the TCL, which converts the BroadcastChange function call by the CArray object to a ProviderChanged call to the CArrayPane-based object).

The VA-generated code for the CAccount class includes functions that override both the ProviderChanged and DoCommand functions of the base class. I have not modified any of the code in these functions to implement the Account view.

CloseWind Override Function Code

In order to keep the TCL from disposing of CAccount objects when the user clicks in the window's close box, I have chosen to hide the window in that case. The window is shown when the corresponding account is selected in the Main window or by choosing the account, by name, from the **Account** menu.

When the user clicks a window's close box, the CloseWind function for the window's director is called. I have overridden this function to provide the desired behavior. The code is as follows:

```
void CAccount::CloseWind (CWindow *theWindow)
{
   // overrides the CDirector::CloseWind function to hide the
   // window rather than closing it and disposing of the
   // window's director.

   TCL_ASSERT (theWindow);
   theWindow->Hide ();
}
```

CreateNewEntries Function Code

When a new CAccount object is created by the DoCmdNewAcct function in the CMain class, that function also calls the CreateNewEntries function in the CAccount class to populate the list for the Account view with enough "empty" entries to cause the vertical scroll bar to activate and the blank entries to be drawn. The code for the CreateNewEntries function is as follows:

```
void CAccount::CreateNewEntries ()
{
   CTransaction *anEntry;
   DateTimeRec  date;
   CString s("\p");

   for (int i=0; i < 10; i++)
   {
      anEntry = TCL_NEW (CTransaction, (itsAcctID));
      anEntry->SetStatus (EMPTY);
      GetTime (&date);
      anEntry->SetDate (date);
      anEntry->SetNumber (s);
      anEntry->SetDescription (s);
      anEntry->SetInfo (s);
      anEntry->SetPayment (0);
      anEntry->SetDeposit (0);
      anEntry->SetBalance (0);
      itsTransactions->Add (&anEntry);
   }
}
```

The foregoing code creates ten—an arbitrary number—entries in the `itsTransactions` array. Each entry is initialized with default values that indicate it is empty and has no useful information in any of its fields. As indicated previously, the intention of this code is to populate the array with enough transactions to cause the scroll pane's vertical scroll bar to activate and the separators and other graphic components of the entries to be drawn.

It is important to point out that adding CTransaction objects to the `itsTransactions` array causes the CAcctList object (which implements the transaction list display) to receive a succession of calls to draw the contents of each of the newly entered entries. This occurs because of the collaboration mechanism that ties the `itsTransactions` array to the `fAccount_Entries` CAcctList object. The CArrayPane (base class) contains the primary code in its ProviderChanged function to cause the drawing operation to occur when an entry is added to its corresponding array object. It then calls the AddRow function in the CTable class to cause the entry to be drawn. At the most fundamental level, the DrawCell function is responsible for drawing a cell in any object that inherits functionality from the CTable class (that is, CArrayPane and its derived class, CAcctList).

The foregoing is the last function to be customized in the CAccount class; however, doing so does not implement the ability to input text into the entries, add or remove new entries, or perform any other related functions. I will leave this for you to implement,

but will provide some advice in how to accomplish these tasks at the end of the section. The next area to discuss is the code that implements drawing the pseudo fields of the CAcctList entries.

Examining Custom Code in the CAcctList Class

We created the CAcctList class with the VA (described beginning on page 110, in step 10) and generated code. The VA generated both an x_CAcctList base class and a CAcctList derived class. Both source files contain only GetFrom and PutTo functions, for compatibility with the Object I/O features of the TCL.

CAcctList Header File Contents

The header file for the CAcctList class is named **CAcctList.cp** and contains the class declaration, with the addition of our single, newly added, override function. The contents of the header file is as follows:

```
#pragma once
#include "x_CAcctList.h"

class CAcctList : public x_CAcctList
{
public:

   TCL_DECLARE_CLASS

   // Object I/O functions

   virtual void    PutTo(CStream &aStream);
   virtual void    GetFrom(CStream &aStream);

   // override of function in CTable class
   virtual void    DrawCell (Cell theCell, Rect *cellRect);
};
```

The foregoing class declaration shows the prototypes for the PutTo and GetFrom member functions generated by the VA, as well as the prototype for the DrawCell function that we have added to override that function in the CTable class.

Global List Border Array

To support drawing the separators for individual cells in the CAcct-List subview, I have created an array of `listBorders` elements (each of which is a variable of type Rect), whose contents are as follows:

```
Rect listBorders[] =
{
   {0, 31, 36, 31},{0, 83, 36, 83},{0, 219, 36, 219},
   {0, 291, 36, 291}, {0, 359, 36, 359},{17, 31, 17, 439},
   {17, 147, 36, 147}
};
```

As is evident in the foregoing definitions, there are seven distinct elements whose top, left, right, and bottom coordinates are specified. These definitions are used by the DrawCell function to render the separators for each entry. The coordinates are relative to the top-left corner of the cell's rectangle.

DrawCell Override Function Source Code

The DrawCell function is called by various functions in the CTable class to render the contents of a single cell. The function is called as many times as necessary to render the appearance of each visible cell in the subview. Because the table contains but a single column, in which each cell occupies a single row, custom rendering code is needed to give the appearance of two rows and multiple columns in each cell, according to the prototype heading for the table shown in Figure 4-8. As you can see from the figure, a simple table, with multiple rows and columns wouldn't be appropriate for this application.

When the DrawCell function is called, the first task is to draw the outline and separators that make up an entry in the table. The code to accomplish this task is as follows:

```
void CAcctList::DrawCell(Cell theCell, Rect *cellRect)
{
   PenState savePen;
   short cTop, cLeft, cBottom, cRight;
   Rect r, rB;

   // save the current pen state structure
   GetPenState(&savePen);

   // draw the border with a black pen
   PenPat (&qd.black);
   PenSize (1,1);
   FrameRect (cellRect);

   // next draw the cell dividers using a blue pen

   ForeColor (blueColor);
   r = *cellRect;
   for (int i = 0; i < 7; i++)
   {
      rB.top = listBorders[i].top;
```

```
        rB.left = listBorders[i].left;
        rB.bottom = listBorders[i].bottom;
        rB.right = listBorders[i].right;
        cTop    = r.top    + rB.top;
        cLeft   = r.left   + rB.left;
        cBottom = r.top    + rB.bottom;
        cRight  = r.left   + rB.right;
        MoveTo (cLeft,  cTop);
        LineTo (cRight, cBottom);
    }
    ForeColor (blackColor);

    // now draw the contents of the individual fields

    CTransaction *anEntry;
    itsArray->GetArrayItem (&anEntry, theCell.v + 1);

    // reset the pen state
    SetPenState (&savePen);
}
```

The foregoing code uses a black pen to outline the rectangular border of the cell (whose height and width were specified to the VA when the table was designed). Then a blue pen is used to draw the separator lines between the pseudo fields in the entry.

After the entry's outline and its field separators have been drawn, the foregoing code accesses the CTransaction object that corresponds to the cell being drawn. The `itsArray` variable refers to the member variable associated with the CArray object called `itsTransactions` in the CAccount class. While executing code inside the CAcctList object, that same array can be referenced by using the `itsArray` name. I have also included a function call to restore the pen that was used upon entry to the DrawCell function. This should be the last statement in the function, to ensure that the environment is restored.

I have purposely stopped writing code at this stage, before attempting to draw the contents of the CTransaction object inside the corresponding fields. I will leave that for you to complete. All of the data needed to draw the fields are accessible from the CTransaction object using its access functions (shown in the source code listing for the class beginning on page 137).

Recommended Tasks for Completing the Business Account View

As indicated previously, I have not implemented the Business Account View completely. I have shown how to create the components of the view and how to render the individual cells, but have stopped short of showing the complete functionality of what

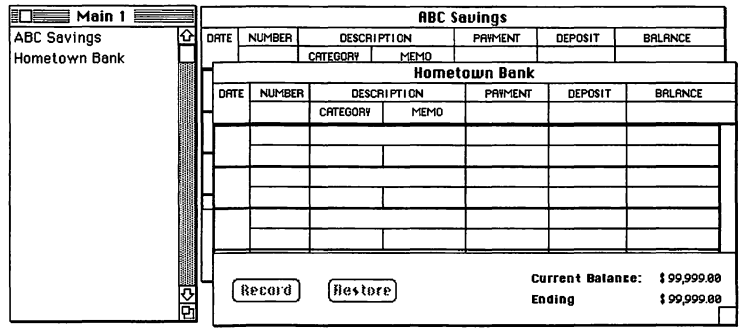

Figure 4-25
Final appearance of the Business Account view and its windows

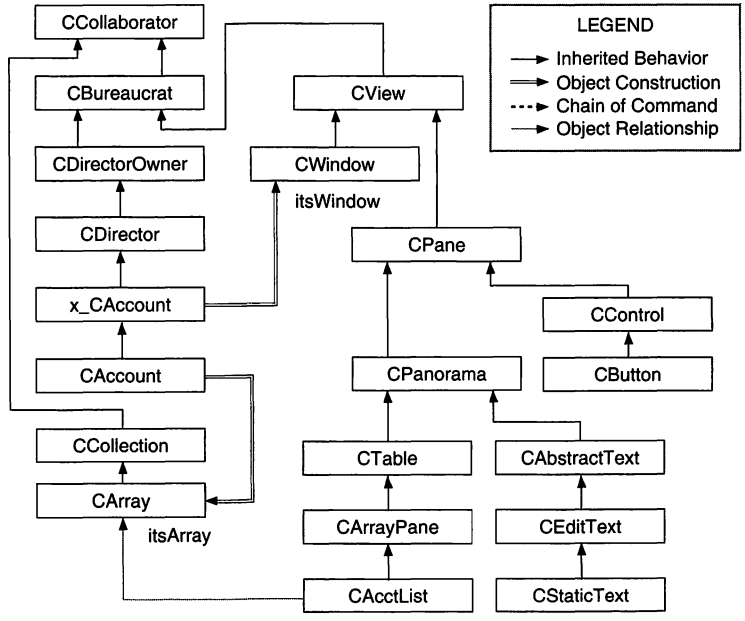

Figure 4-26
Structure of Business Account View and its component objects

could be a good home accounting application. Executing the foregoing code produces what is shown in Figure 4-25.

Figure 4-25 shows both the Main view (with two accounts listed) and also the CAccount views for those two accounts. As you can see, no data has been entered into the individual entries, but the figure gives you an idea of the final appearance of the user interface (of course, when a file of data is read, the name of the Main view will reflect the file's name, rather than "Main 1").

The structure of the application (beginning in the CAccount class, which is created in the DoCmdNewAcct function of the

CMain class), is shown in Figure 4-26. The figure illustrates most of the major objects making up the Account view, their interrelationships, and their inheritances.

In order to complete the functionality of the Business Account View application, here are some suggestions that may be of help:

- Because data cannot be keyed directly into a CTable-based object, you should create a CEditText object in a newly added IAcctList function that can be called by the ICAccount function when the account is initialized. The window will have been created at this time. You can create an array of pane locations and sizes, much like the array of `listBorders` elements (shown on page 145). The individual elements might be specified something like the following:

```
Rect listPanes[] =
{
   {1, 1, 16, 31},{18, 1, 35, 31},{18, 33, 35, 83},
   {1, 85, 16, 219}, {1, 221, 16, 291},{1, 293, 16, 359},
   {18, 85, 35, 147},{18, 149, 35, 220}, {1, 361, 16, 455}
};
```

  As with the `listBorders` Rect definitions, the `listPanes` specify the top, left, right, and bottom locations of the individual fields that occupy each entry in the CAcctList object. They are also relative to the top-left corner of a cell rectangle in the subview. You might have to play around with the settings of the foregoing to get them exactly right. Each is inset one pixel on each side from the corresponding border locations of the field.

- The CEditText object should be sized and positioned on top of the date field of the first empty cell in the array and the list should be scrolled to make the CEditText object visible. The CEditText object should display a blinking cursor, indicating that the user can begin entering text at that location of the entry.

- The application should make provision to recognize the TAB and arrow keys and should position and resize the CEditText object such that it occupies the next (or previous) position in

the entry when these keys are pressed. Before you move the object, its text should be verified (if desired) and be stored into the corresponding CTransaction entry, using the access functions provided for this purpose. Then the CEditText object can be cleared of its contents, be resized, and repositioned.

- The DrawCell function of the CAcctList class should be improved to draw the contents of the individual fields of a CTransaction entry. It should also keep the current balance of the account up to date.

- If the user decides to change the date of an entry, the entry should be moved to a new position in the `itsTransactions` array so that the entries are always kept in chronological order.

- The original contents of an entry being edited should be kept in a temporary CTransaction object so that it can be restored should the user desire to click the Restore button in the view. If the user clicks the Record button, the saved object can be disposed, and then the CEditText object can then be positioned at the date field of the next entry in the subview (scrolling the view to display the next entry if it is necessary to do so).

- The CAccount class can be enhanced in many ways, allowing the user to delete an entry in the list, create printed reports, and perform many other chores that are associated with a fully capable home or business accounting application.

- Each time an entry is made or changed, you should call the document's SetChanged function with a TRUE argument. (The document (CMain) is accessible to the CAccount class by referring to the `itsSupervisor` variable.)

- Providing Object I/O for the application is the topic of a later chapter and you will be able to get some ideas of how to implement it from that material.

The main focus in this section was to show how to create and implement the special view, and although the foregoing suggestions are just an outline of what you need to do to complete the application, the work that we have done should give you a good start.

Creating a Splash Screen View

Let's turn our attention to a different sort of view—one that is a standard component of shipping applications, but also one that developers are constantly asking how to create. The splash screen that appears briefly when a program is launched is a simple view that is also very simple to implement with the facilities of the VA and the TCL.

Creating the Splash View

We've created a new VA project, called "Splash," to illustrate how easy it is to add a splash screen to any TCL application. The steps to accomplish this are as follows:

1. After the project and its files have been created, launch the VA by double-clicking on the Visual Architect.rsrc entry in the project's list of files.

2. When it starts up, you will see that the VA has created a Main view that you can use as the basis for the application's primary view. In order to create a splash screen view, pull down the **View** menu and choose the **New View** command. This will cause the VA to display a dialog, allowing you to enter the name of the view and its type. Enter "Splash" as the name of the view, choose **Splash Screen** from the pop-up menu, and then click OK, as shown in Figure 4-27.

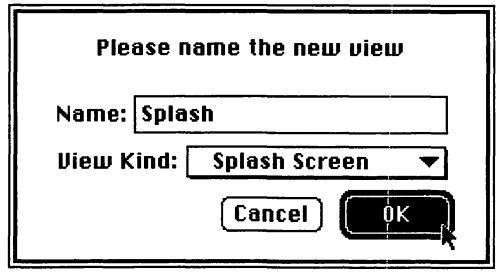

Figure 4-27
Choosing to create a Splash Screen view called "Splash"

3. The VA will create a new "Splash" view and will display a blank window in which you can place the content of the splash screen. For our purposes, using the color wheel PICT image built into the project resource file is fine. Choose the

Picture tool from the Tools menu and drag a frame in which to display the picture. The standard PICT image will be displayed automatically in the frame.

If you place a number of PICT images into the **Visual Architect.rsrc** resource file, you can choose the one you want to use for the splash screen by double-clicking on the PICT image (or select it and choose **Pane Info** from the **Pane** menu), twisting the control at the side of the CPicture class to display the settings (showing the default PICT image), and then clicking on the image. This will display a dialog that contains thumbnail views of all of the PICT images in the **Visual Architect.rsrc** file. Click to choose the one you want to use and click OK to dismiss the dialog. Then in the settings for the CPicture object, you can choose for the picture to be scaled to the size you've specified when creating the picture frame. If you do not choose the "Scaled" option, then the image will be clipped to the boundaries of the frame you've drawn.

4. At this point, we added a single static text field, with the word "Splash!," to add something to make the view unique. You have complete freedom to put whatever content you wish in the splash screen view. Our example screen appears in the VA as shown in Figure 4-28.

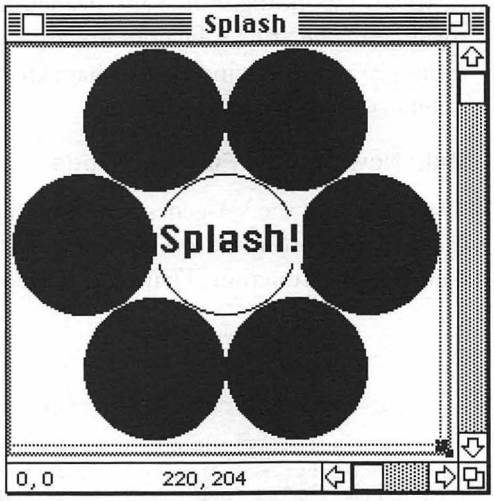

Figure 4-28
Completed splash screen view inside the VA

5. Choose **Generate All** from the **Project** menu in the VA to generate all of the files that are necessary to create the application and its Main and Splash Screen views.

You can compile and execute the application at this point. When you do, you'll see that the splash screen that you designed will magically appear when the application commences execution. I think you'll agree that nothing could be easier than the foregoing steps for creating splash screen views.

Examining the Splash View Code

You are probably wondering what code is generated to display the splash screen we designed in the foregoing section, and I'm going to show it to you in what follows; however, you should first refer back to Chapter 2, where the start-up actions of the TCL are described (beginning on page 38, in step 2). If you recall, the TCL calls a function called ShowSplashScreen and allows you to override that function to display a custom splash screen. The built-in functionality of the TCL is the basis for the code generated by the VA.

When the VA generates code for the project, in addition to the skeleton code for the Main view (described in detail in previous chapters), the VA generates an x_CSplash and CSplash classes (with corresponding source and header files). These classes are derived from the TCL's CDialogDirector class. The splash screen is really just a dialog window that happens to be created and shown briefly as the application starts up.

MakeNewWindow Function Code

The portion of the VA-generated code that creates the Splash view is contained within the **x_CSplash.cp** base class file, in the MakeNewWindow function. That code is as follows:

```
void x_CSplash::MakeNewWindow(void)
{
   itsWindow = TCLGetNamedWindow("\pSplash", this);

   // Initialize pointers to the subpanes in the window
   fSplash_Pict1 = (CPicture*) FindPane(kSplash_Pict1ID);
  ASSERT(member(fSplash_Pict1, CPicture));
   fSplash_Stat2 = (CStaticText*) FindPane(kSplash_Stat2ID);
  ASSERT(member(fSplash_Stat2, CStaticText));
}
```

The foregoing code illustrates that creating the Splash view is no different from creating any other window. The MakeNewWindow function accesses the 'CVue' resource named "Splash" to create the individual components of the view. In this case, the window contains a CPicture resource (kSplash_Pict1ID) and a CStaticText resource (kSplash_Stat2ID). Each of these resources is reified and its pointer is stored into a corresponding member variable of the CSplash object. That process completes the creation of the view.

ShowSplashScreen Function Code

The VA-generated code for the x_CApp class contains the override of the TCL's CApplication class ShowSplashScreen function (which is empty in that class). The newly generated code for invoking the Splash dialog is as follows:

```
void x_CApp::ShowSplashScreen()
{
   long dummy;
   CSplash *splash;

   /* Run splash screen for a short while */
   splash = TCL_NEW(CSplash,());
   splash->ICSplash(this);
   splash->BeginDialog();
   splash->itsWindow->Update();

   // If you want a longer delay, change the _App.cp
   // template below. If you want entirely different
   // splash screen handling, override ShowSplashScreen
   // in your application class and do your own thing.

   Delay(60, &dummy); /* Delay long enough for user to see */
   TCLForgetObject(splash);/* Get rid of splash screen */
}
```

The foregoing code creates the CSplash object, initializes the object, calls BeginDialog to select and display the window's contents, and then calls Update to update the screen. Following those steps, the code calls the Delay toolbox function with a duration of 60 "ticks" (60ths of a second), causing the splash screen to be displayed for approximately one second. After the delay has elapsed, the TCLForgetObject function is called to dispose of the object and its window.

That's all there is to creating a splash screen for your applications. You will probably spend a lot more time designing the PICT image than it took for me to write this section of the chapter.

Creating a Floating Palette View

Many applications use floating "palette" windows to contain tool boxes or other items that are referenced frequently. The VA incorporates the design and code generation tools and the TCL contains all the functions you will need to create and use this type of view in your applications.

I'm going to build a Tool palette in a floating window to show you how it's done. Tear-off menus will be described shortly. These are also floating windows after they have been torn off.

Creating the View

The palette is built in a new VA project by choosing **New View** from the **View** menu and then filling in the dialog, as shown in Figure 4-29.

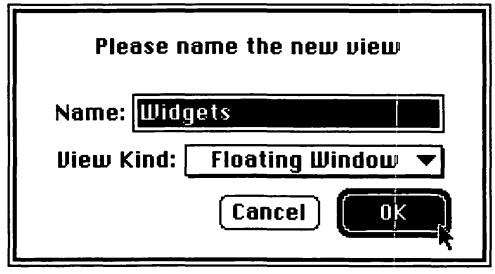

Figure 4-29
Creating the Widgets floating window

The "View Kind" is set to "Floating Window" from the pop-up menu, and the view is named "Widgets." After you click OK, the VA will create a blank window, whose characteristics are set by choosing **View Info** from the **View** menu. The settings for our Widgets window are shown in Figure 4-30.

I have used the default settings, except that I have set the width of the window to 95 pixels and its height to 255 pixels. This is to allow enough space for a palette that has 8 rows of cells, each of which has 3 columns, for a total of 24 individually selectable "widgets" from the palette.

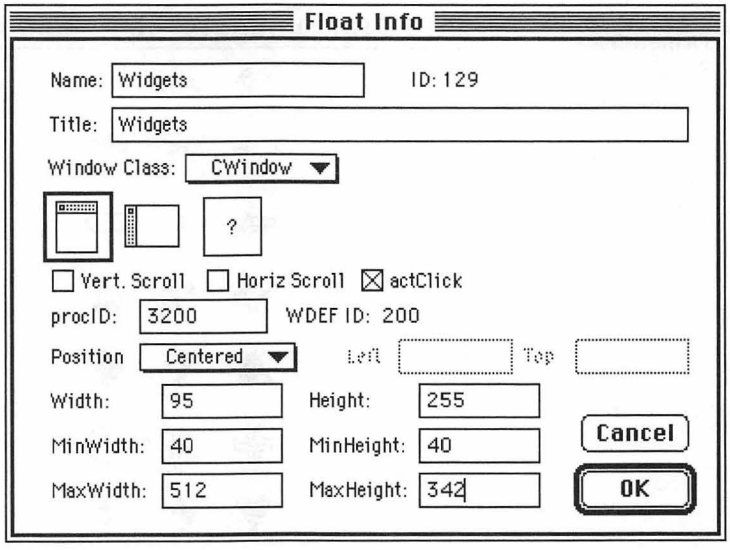

Figure 4-30
View Info for the Widgets window

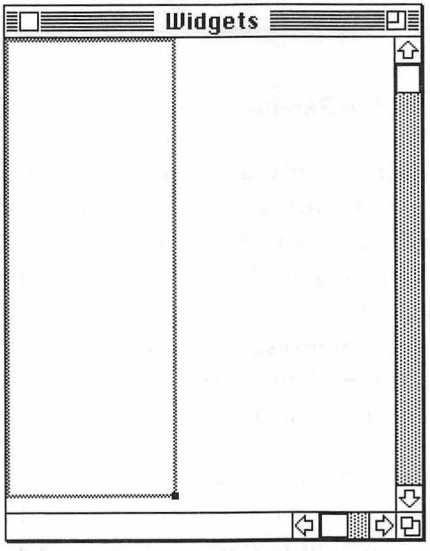

Figure 4-31
Widgets floating window as displayed by the VA

The final appearance of our (unpopulated) floating window appears as shown in Figure 4-31. The window is tall and thin and includes only a CPanorama object.

After the Widgets window has been created, you can tell the VA to generate code to create source and header files for the CWidgets and x_CWidgets classes.

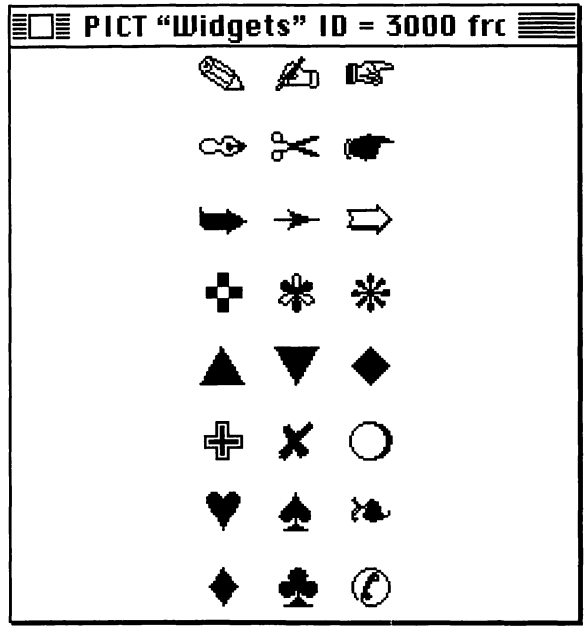

Figure 4-32
Widgets PICT
resource ID 3000

Creating the PICT Image for the Palette

In order to display a palette of tools, one good approach is to create a single PICT resource that contains all of the tools, located in proper relationship to one another, and in the proper size. I used a drawing program (Canvas™, by Deneba Software) to create the image, then copied it to the clipboard and then added it to the **Project Resources.rsrc** file for our project, by creating a 'PICT' resource in ResEdit™ (Apple Computer's resource editor) and then pasting the contents of the clipboard.

I gave the resource an ID of 3000 (large enough not to conflict with any of the TCL or VA resources). The PICT resource image, as displayed in ResEdit, is shown in Figure 4-32. (Note that the resource is only 96 pixels wide by 256 pixels high—it is centered in the PICT image editor by ResEdit.)

The image itself is formed by selecting individual characters from the Zapf Dingbats™ font. Of course, you will want to draw a picture of "real" tools, patterns, colors, or whatever you wish for your palette to display. Be sure that each tool's image has uniform horizontal and vertical spacing.

Creating the PICT Grid Resource

In order to display the PICT image shown in Figure 4-32 as a palette of individually selectable images, you will also need to create a 'PcGd' resource to specify to the TCL's CPICTGrid class, which PICT image is to be used for the palette and how the image is to be partitioned into rows and columns. I created a separate file called **widgets.r** to add to our project file. The contents of that file are as follows:

```
/*
   PcGd resource type

   creates a 'PcGd' resource for use with
   the TCL's CPICTGrid class.

*/
type 'PcGd'
{
   integer;                /* rows          */
   integer;                /* columns       */
   integer;                /* boxWidth      */
   integer;                /* boxHeight     */
   integer                 /* hSizing       */
      sizFIXEDLEFT   = 0,
      sizFIXEDRIGHT  = 1,
      sizFIXEDSTICKY = 4,
      sizELASTIC     = 5;
   integer                 /* vSizing       */
      sizFIXEDTOP    = 2,
      sizFIXEDBOTTOM = 3,
      sizFIXEDSTICKY = 4,
      sizELASTIC     = 5;
   integer;                /* hLoc          */
   integer;                /* vLoc          */
   integer;                /* commandBase   */
   integer;                /* PICTid        */
};

resource 'PcGd' (3000, "Widget Grid", appheap, preload)
{
   8,                  /* rows          */
   3,                  /* columns       */
   32,                 /* boxWidth      */
   32,                 /* boxHeight     */
   sizFIXEDSTICKY,     /* hSizing       */
   sizFIXEDSTICKY,     /* vSizing       */
   0,                  /* hLoc          */
   0,                  /* vLoc          */
   3000,               /* commandBase   */
   3000,               /* PICTid        */
};
```

The first section of the **widgets.r** file contains the definition of the format and contents of a 'PcGd' resource. The second section of the file defines the "Widget Grid" resource, whose ID is 3000.

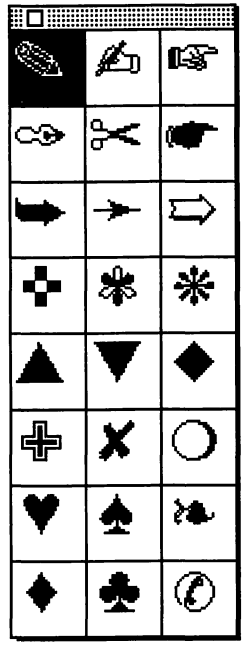

Figure 4-33
Final appearance of Tool palette when the window is opened

I have specified that the resource is to be stored into the application heap and is to be preloaded when the resource file is opened. The resource specifies values for each of the required fields. Notice that I have referenced our `PICT` resource in the last field, by specifying its resource ID (3000). The next to last field in the `PcGd` resource contains a value called the `commandBase`. This value is used as the basis for a *command number* for each of the palette's tools, when it is selected. Figure 4-33 shows the Tool palette as it appears when the floating window is opened in the application. The first tool is selected by default.

Examining the Floating Palette View Code

The VA-generated code for the floating palette is held in two classes. The x_CWidgets class contains the code to create the appearance of the floating window in its MakeNewWindow function. The CWidgets class contains only a constructor function and a ProviderChanged override function. A small amount of additional code is generated into the CApp class. We'll examine that code first.

SetUpMenus Function Code

As you may recall from the discussions about the application object in Chapter 2, beginning on page 35, you will recall that when the application is in the process of creating and initializing its "helper" objects (CBartender, CSwitchboard, and so on), it also calls the SetUpMenus function. The VA-generated code for our application overrides this function in the x_CApp class to create the floating window at the same time that the application's menus are created. The code for the function is as follows:

```
void x_CApp::SetUpMenus()
{
   /* Create floating windows*/

   gCWidgets = TCL_NEW(CWidgets,());

   CApplication::SetUpMenus();
}
```

The foregoing code creates the CWidgets object (causing its constructor function to execute, which we will examine shortly) and stores its pointer into a global variable called `gCWidgets`. By making the pointer to the CWidgets object global, the VA has allowed the floating window to be accessed from anywhere in the application. It is simply not known how you intend to use the window, or whether it should be associated with the application object, the document object, or some other object in the program.

CWidgets Constructor Function Code

The next series of events take place in the constructor function for the CWidgets object. Its code is as follows:

```
CWidgets::CWidgets()
{
   // create the new window and then also create the
   // CPICTGrid object that implements the palette.

   MakeNewWindow();
   widgets = TCL_NEW(CPICTGrid,(kWidgetPaletteID, itsWindow, this));
}
```

The constructor for the CWidgets object calls the MakeNewWindow function in its base class to create the floating window. I have added a statement to create a new member variable, whose type is CPICTGrid, specifying the resource ID for the `'PcGd'` resource

we created, as well as specifying that the CWidgets' window (`itsWindow`) is the enclosure for the palette and the CWidgets object (`this`) is the supervisor for the palette. Accordingly, I have added new declarations to the **CWidgets.h** header file for both the resource ID (`kWidgetPaletteID`) and the `widgets` member variable. That code is as follows:

```
#pragma once
#include "x_CWidgets.h"
#define kWidgetPaletteID 3000 // rsrc ID for PcGd resource
#define kBaseWidgetCmd 3000   // base command for selection

class CDirectorOwner;
class CPICTGrid;

class CWidgets : public x_CWidgets
{
public:

   TCL_DECLARE_CLASS

   // Insert your own public data members here
   CPICTGrid *widgets;

   CWidgets();

   virtual void ProviderChanged(CCollaborator *aProvider,
      long reason, void* info);
   virtual void DoCommand(long theCommand);
};
```

The only additions to the foregoing code are shown in a bold typeface. Note that I have defined symbolic constants for both the 'PcGd' resource and the command base value.

MakeNewWindow Function Code

The constructor for the CWidgets object calls the MakeNewWindow function to create the floating window. The code for that function is as follows:

```
void x_CWidgets::MakeNewWindow(void)
{
   itsWindow = TCLGetNamedWindow("\pWidgets", this);
   HideInitially();

   // Initialize pointers to the subpanes in the window
}
```

After the window is created, the VA-generated code calls the HideInitially function to hide the window. In order to use the window, we must first open and then show it.

CMain MakeNewWindow Override Function Code

In order to show the floating palette, I decided to override the MakeNewWindow function for the CMain (document) class object. If you want the window to be associated with a particular document, then you can do it as I did. The code for the override function is as follows:

```
void CMain::MakeNewWindow(void)
{
   Point where = {50, 350};

   //
   // call the base class function to create the Main view
   // and then open the Widgets palette.
   //
   x_CMain::MakeNewWindow();
   gCWidgets->OpenWind (where);
}
```

CMain Activate Override Function Code

To show the floating palette when the Main view is active, I added an override for the Activate function (inherited from the CDirector class) to our CMain class. The code is as follows:

```
void CMain::Activate()
{
   x_CMain::Activate();
   gCWidgets->ShowWindow();
}
```

In the foregoing code, I call the Activate function in the base class and then use the global `gCWidgets` variable to call the ShowWindow function for the palette.

CMain Deactivate Override Function Code

Just to complete the functionality of the floating palette, I also added an override for the Deactivate function of the CDirector class in our CMain source code. The code hides the floating window when the Main view becomes inactive, as follows:

```
void CMain::Deactivate()
{
   x_CMain::Deactivate();
   gCWidgets->HideWindow();
}
```

As with the Activate function, the foregoing Deactivate function calls the base class version of the function and then calls the window director's HideWindow function by using the `gCWidgets` global variable to reference the object.

CWidgets DoCommand Function Code

When the user clicks on any of the tools in the palette, the TCL calls the DoCommand function of the current gopher with a 32-bit (long integer) value representing the command associated with the selected tool. Unique command numbers are formed by storing the `commandBase` value in the high order 16 bits of the word, with a tool-number in the low order 16 bits, and then the entire word is negated.

You can determine whether the command pertains to your palette by first determining whether the command number is negative, negating the word, and then comparing the high order bits of the command value with what you assigned as your `commandBase` (in our case it is 3000). If the value matches, then you can determine which tool was clicked by negating the command and examining its low order 16-bit word.

I have added code to the VA-generated code for the CWidgets class to show how selection commands can be recognized. The code is as follows:

```
void CWidgets::DoCommand(long theCommand)
{
   short base, tool;

   if (theCommand < 0)
   {
      base = HiShort(-theCommand);
      tool = LoShort(-theCommand);

      if (base == kBaseWidgetCmd)
      {
         // handle the selection of the tool
      }
      else
      {
         x_CWidgets::DoCommand(theCommand);
      }
   }
   else
   {
      x_CWidgets::DoCommand(theCommand);
   }
}
```

The foregoing code is only interested in handling commands if the command identifier (`theCommand`) is negative. If it is a positive value, then the DoCommand function for the base class is called. If the command identifier is negative, then we check to see whether its upper 16-bit word (after negating the value) matches the value for our `kBaseWidgetCmd` constant. If so, then we know that we can use the value in the `tool` variable to determine which tool was selected.

Tools are specified in the low order 16-bit word of the command identifier by commencing with the value 1 for the first row and column, and then advancing the tool number by 1 as you proceed left to right and top to bottom in the palette. The top-left tool number is 1; the top-right tool number is 3; the bottom-left tool number is 22; and the bottom-right tool number is 24.

Creating a Tear-off Menu View

A tear-off menu is very similar to a floating window, but offers the additional functionality of a normal menu. When the menu is torn off, it becomes a floating window, just like what was described in the previous section. If you create a tear-off menu view, you have the option of using just the floating window or including the functionality of a tear-off menu.

Creating the Tear-off View

Before creating the tear-off view, you should prepare a `PICT` resource that contains the view that you want to be shown. Store the resource into the **Visual Architect.rsrc** file, where the VA can access it.

The tear-off menu view is created in the VA in just the same way as any other view. Choose **New View** from the **View** menu, and then name and select the view type as shown in Figure 4-34.

After the dialog is dismissed, the VA will display a window containing a sample `PICT` image, as shown in Figure 4-35. Double-click on the `PICT` image, or select the image and choose the **Pane Info** command from the **Pane** menu. This will cause a window to be displayed, showing the various classes in the hierarchy of the new view. Note that the VA has decided to create the new view as a CPICTGrid object.

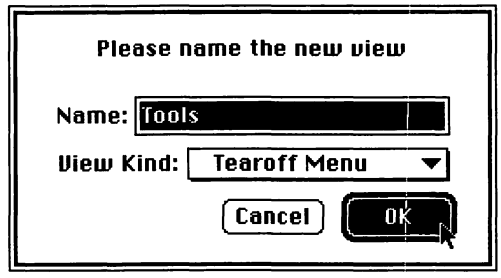

Figure 4-34
Creating the Tools tear-off menu view

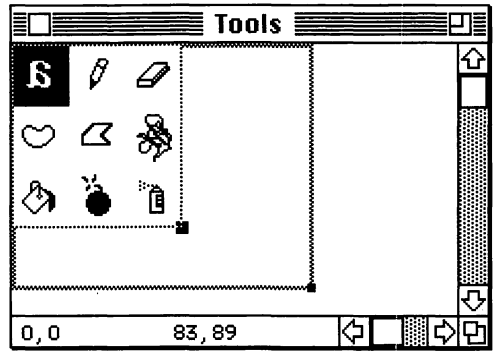

Figure 4-35
Default window contents for tear-off menu view

Twist the indicator next to the CPICTGrid class, and then click on the image shown in the pane. This will display a window that allows you to choose the image you want to be associated with the view. An example of this window is shown in Figure 4-36. I have chosen to use a copy of the "Widgets" `PICT` image that was used in the previous section for this example.

After you click OK in the `PICT` selection window, you will need to specify a command for the menu to execute when the user chooses one of the tools in the menu. Click on the Command pop-up menu, and choose **Other**. Then choose **New Command** from the **Edit** menu, and enter `cmdSelectTool` as the name of the command. You also need to select the class in which the command is to be handled and specify what the command should accomplish. In our case, we specified that the CTools class should

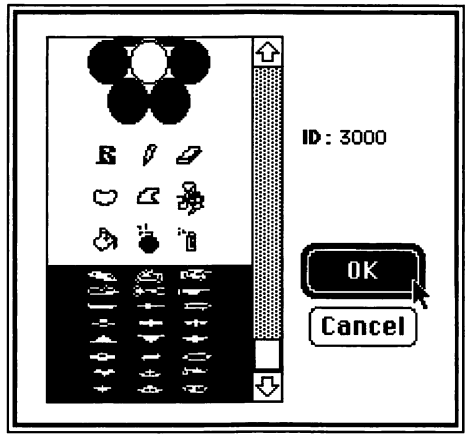

Figure 4-36
Selector window for tear-off view's PICT image

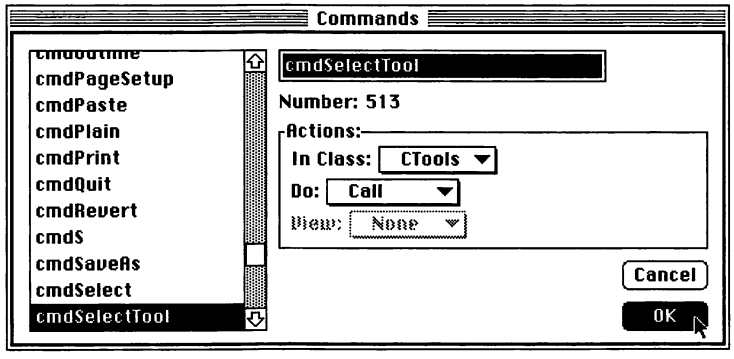

Figure 4-37
Specifying the command for a selection from the Tools menu

handle the command and that it should call a function to do so. The result is shown in Figure 4-37.

After the command has been chosen, you will need to specify some of the settings for the CGridSelector base class in the Tools window. Twist the indicator next to the CGridSelector class name, and change the settings so that they correspond to what is shown in Figure 4-38. Notice that the "Grid On" option for the CGrid-Selector class has also been selected. After these settings have been made, the view is complete. After you dismiss the Tools settings window, the Tools window is displayed in the VA, as shown in Figure 4-39.

What remains to be done is to specify a menu in the menu bar to contain the tear-off view. This is an optional step and if you choose not to create a menu, then the resulting generated code will be much the same as what we generated for the floating win-

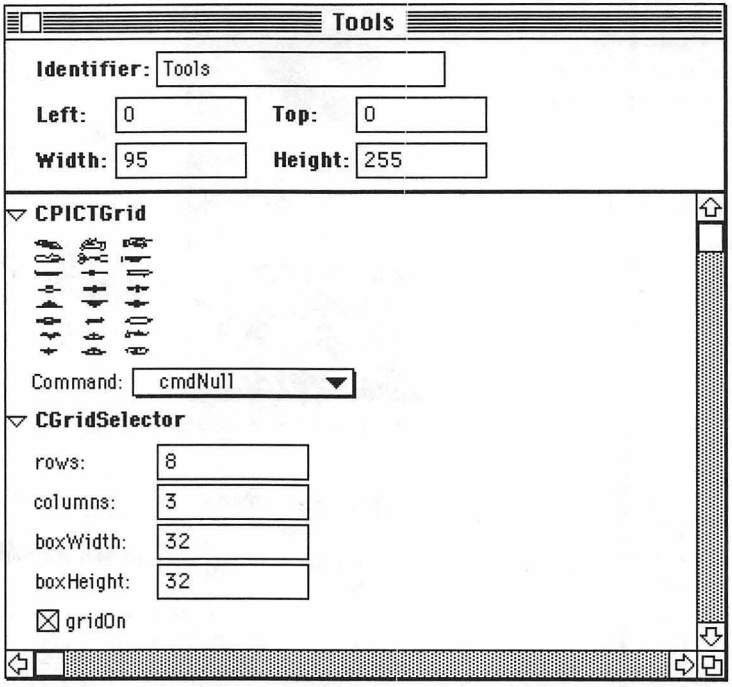

Figure 4-38
The complete settings for the Tools tear-off menu view

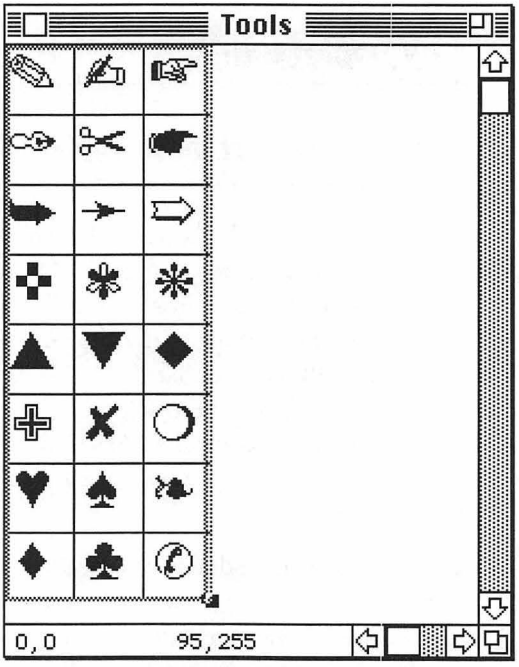

Figure 4-39
Completed Tools view inside the VA

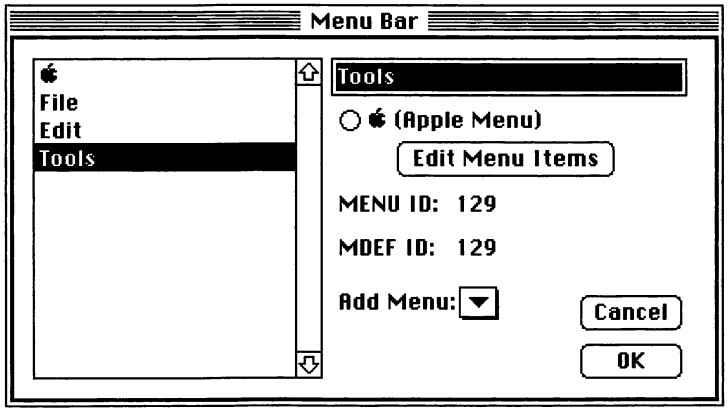

Figure 4-40
Tools menu added to the menu bar

dow view. If you do decide to create a menu and allow it to be torn off, then the steps for doing so are quite simple. Choose **Menu Bar** from the **Edit** menu, and then click the arrow next to the "Add Menu" label in the window. Choose the Tools palette, which probably has a name something like "Tearoff 129," or something similar. After you have chosen that menu, you can rename it. I have chosen to call our version "Tools," as shown in Figure 4-40.

Finally, after the menu has been added, you can generate code and test the functionality of the skeleton application, as generated by the VA. As you can see in Figure 4-41, the menu is fully implemented. Although I can't show it being torn off, you'll have to take my word that it works just fine.

Examining the Tear-off View's Code

The VA-generated code for the Tools view is very much like what was generated for the floating window. In fact, after a menu is torn off, it *becomes* a floating window. So what is different in this case is the functionality of the menu and the ability for the user to tear it off. All of this is handled automatically by the TCL when you add the tear-off view to the menu bar.

Once again, it is useful to point out that if you prefer to create your Tool palettes as tear-off menus and then simply avoid adding them to the menu bar, they will function in the same manner as the floating window view shown earlier.

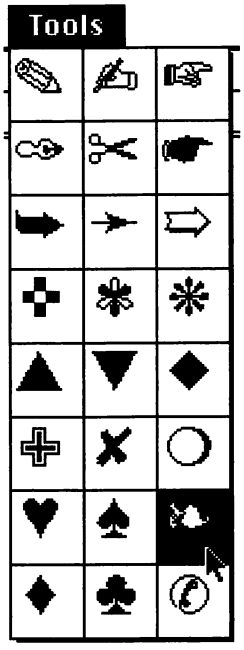

Figure 4-41
Tools menu inside the running application with one of the tools selected

SetUpMenus Function Code

As with the "floating window" view, the VA generates code in the SetUpMenus function of the x_CApp class to create the tear-off menu's floating window view. The code is as follows:

```
void x_CApp::SetUpMenus()
{
   /* Create floating windows*/
   gCTools = TCL_NEW(CTools,());

   CApplication::SetUpMenus();
}
```

In the foregoing code, the CTools object is created first, its pointer is stored in a global variable called `gCTools`, and then the SetUpMenus function in the CApplication base class is called.

CTools Constructor Function Code

The constructor for the CTools function takes on the task of creating the floating window view, which is hidden initially. The code for that function is as follows:

```
CTools::CTools()
{
   // Call the IViewPICTGridDirector member function
   // to read in the window and initialize it.
   // Here instead of in a lower-level class so
   // can be modified.

   x_CTools::IViewPICTGridDirector("\pTools", 129);
}
```

The foregoing code calls one of the VA's library functions to create the floating window. This is different from the approach you saw in the generated code for the floating-window example (CWidgets), where the MakeNewWindow code was called. The foregoing code takes advantage of the TCL's Object I/O facilities to load the values of the various parameters specified for the CGridSelector class in the Pane Info window (see Figure 4-38) for the Tools pane; for the CWidgets code, I had to create a separate 'PcGd' resource.

No other code is needed to make the new Tools menu operate as a menu or be torn off to create a floating palette. The TCL contains all of the requisite functionality to handle these tasks.

A View Summary

The examples in this chapter have shown that it is possible to create almost any type of view by using the VA to create the basic elements of the view and then generate skeleton code. The generated code is often quite easy to customize to provide the complete functionality required by a given application.

The "Business View" was the most complex, mainly because it required us to create a custom draw function and make provisions for handling multiple rows and unaligned columns in a single column view. The remaining examples were quite straightforward and were implemented almost solely in the VA, with a minimum of additional custom code.

We will be looking at other examples of views in the chapters that follow. The very next chapter deals with dialog views—both modal and modeless.

Chapter 5

Creating and Managing Dialogs

Dialogs, both modal and modeless, are views in the strict sense of the word. We treat them differently mainly because in the past the Macintosh Dialog Manager handled them. This is no longer the case because the TCL treats dialogs just like other windows, except for which menus it allows to be displayed and how events are processed. A modeless dialog is not very different from a normal window view. The VA handles it differently, mainly because dialogs are intended to solicit input from the user, while windows normally display output to the user. A modal dialog is especially unique in the respect that it *requires* the user to input a response and dismiss the dialog before any other work can be performed.

There are many similarities to modal and modeless dialogs, however, and I will attempt to make these apparent as we look at examples of each.

Creating a Text Style Modal Dialog

Let's begin our examination of dialogs by describing a typical application. Our application contains a simple Edit Text window, whose text we wish to format in different fonts, sizes, styles, and justifications. One approach to this problem is to create a series of menus, each of which specifies a single characteristic and requires the user to choose font, size, style, and justification parameters one at a time. Another approach—and the one we'll use for this example—is to allow the user to open a single dialog, make all of the changes at once, and then dismiss the dialog.

The example that illustrates the text style dialog is called "Notebook," because it portrays a notebook in which one might enter various types of text. The notebook does not implement "styled text," but, instead, assigns the selected font,

size, style, and justification to the entire text. In this respect, the notebook cannot be used as a substitute for a full-fledged word processor program.

Creating the Main and Notebook Views

Create a new VA project in the Project Manager, and then double-click the **Visual Architect.rsrc** file to launch the VA. We don't want to keep the contents of the Main view, so double-click the Main view name in the VA's list of views, choose **Select All** from the **Edit** menu, and press the Delete key to delete the contents of the view. In place of the picture and static text, choose the **Panorama** tool from the **Tools** menu, position the crosshairs of the mouse pointer at the top-left corner of the window, and draw a panorama that has the appearance shown in Figure 5-1. You'll have to adjust the gray rectangle (the window port's dimension) so that there is room at the bottom of the window for its size box.

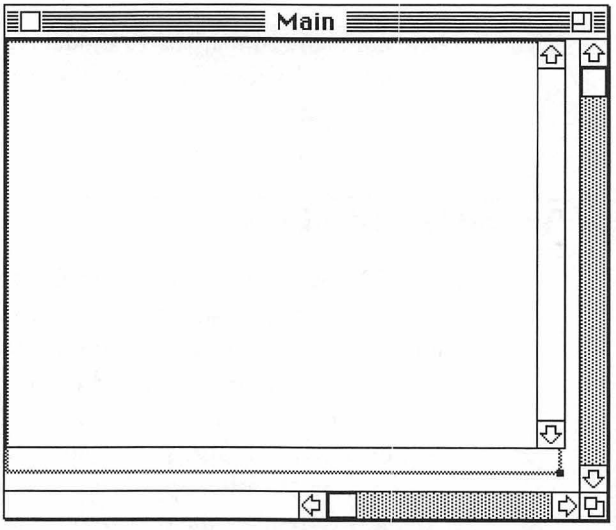

Figure 5-1
New Main view with Edit Text panorama

When you create the new panorama in the Main view, the VA creates it as a CEditText object. This is what we want in this case.

Now that the Main view has been created to hold editable text, we can create a dialog—that is our main purpose for this section—to give us the ability to change the text's font, size, style, and justification settings. To create the dialog, choose **New View** from the VA's **View** menu, enter "Notebook" as its name, make sure that

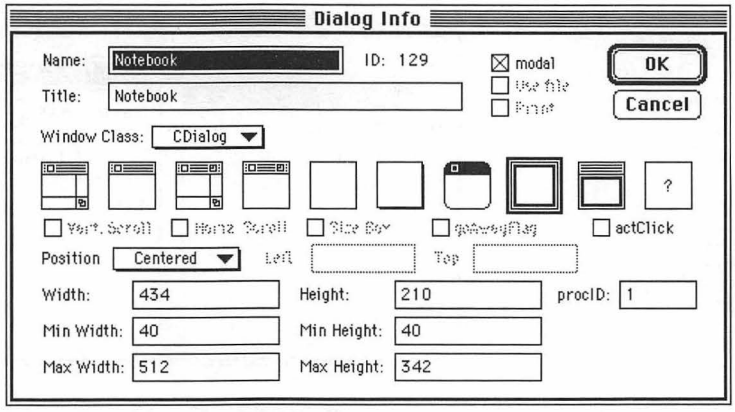

Figure 5-2
View Info for Notebook view

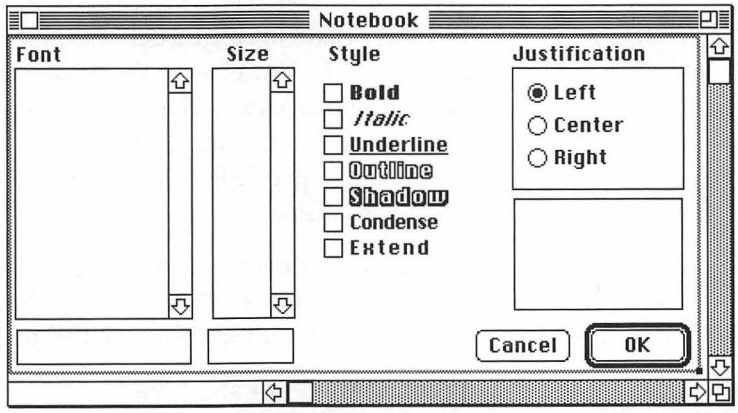

Figure 5-3
Appearance of completed Notebook dialog

the view type is Dialog, and then click OK to dismiss the view type dialog. The VA will display a new view window in which we can enter the fields and controls that will make up our Modal dialog. Choose the **View Info** command from the **View** menu, and make sure that the settings for the dialog correspond to what is shown in Figure 5-2. Note that the Modal checkbox is checked in the figure. Click OK to dismiss the View Info window.

The appearance of the completed Notebook dialog is shown in Figure 5-3. You may wish to refer to this figure as you create the various elements of the dialog.

Creating The CFontList and CSizeList Classes

Before creating the list element under the **Font** label in the Notebook view, you will need to create a new class that is derived from

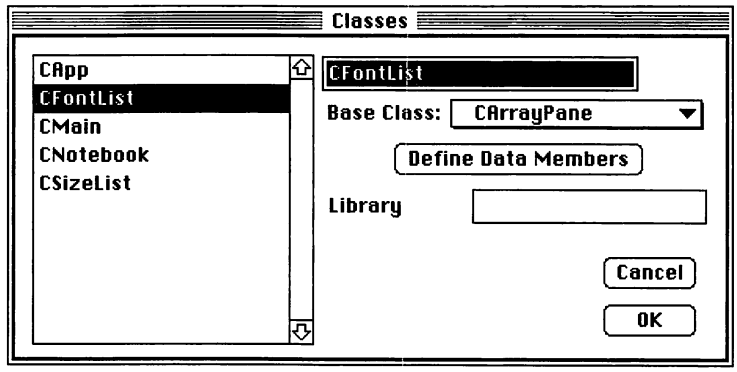

Figure 5-4
CFontList class created

the CArrayPane class so that you can supply the contents of the list to the CTable class when the DrawCell function is called. You will always have to create a derived class when you use the List/Table tool in the VA. Choose the **Classes** command from the **Edit** menu, and then choose **New Class** (or the Command-K keyboard shortcut). Fill in the dialog as shown in Figure 5-4. Note that the Base Class is specified as CArrayPane in the dialog. Click OK to dismiss the dialog.

Create another new class called CSizeList, as shown in the list of classes in Figure 5-4. Make sure that the new class is also derived from CArrayPane, and then click OK to dismiss the dialog.

Creating the Font and Size Lists

After the CFontList and CSizeList classes have been created, you can create the lists themselves. Choose the **Table/List** tool from VA's **Tool** menu, and draw the font list so that it appears like what is shown in Figure 5-3. Double-click the list pane, change the name of the list to FontList, and also change the `selectionFlags` settings in the CTable class to `selOnlyOne`. This makes sure that the user can select only one entry at a time. None of the other settings need be changed. You can dismiss the Pane Info window by clicking in its close box.

To associate the FontList element with the proper class, make sure it is still selected in the Notebook view, pull down the **Pane** menu, and choose CFontList from the list of classes in the **Class** hierarchical menu.

Create the SizeList by using the **Table/List** tool, creating a list that has the appearance shown in Figure 5-3. Double-click the list

pane and change the element's name to SizeList, change its `selectionFlags` settings to `selOnlyOne`, and then click the close box to dismiss the window. Change the class of the SizeList by selecting it (if it isn't selected already) and choosing the CSizeList class from the **Class** hierarchical menu in the **Pane** menu.

Creating the Style Checkboxes

The checkboxes under the Style label are created by choosing the Check Box tool from the VA's **Tool** menu and then clicking at the position where you wish for the checkbox to be placed. At that time you will be able to type in a label for the checkbox. After typing the label, make sure the checkbox and its label are selected and then pull down the **Pane** menu and choose the style that corresponds to the checkbox's name from the **Style** hierarchical menu (for example, Underline is underlined and *Italic* is in italics).

Creating the Justification Radio Buttons

The radio buttons that select the justification of the text (Left, Center, or Right) are created by first using the Rectangle tool to draw a frame within which the buttons will be placed and then choosing the Button tool and clicking and naming each of the buttons. When you have created and positioned the three buttons, double-click on the button named Left, and change its `ctrlValue` setting in the CControl class of the Pane Info window to 1, making that selection the default for the radio group.

Creating the CDialogText Fields

Each of the FontList and SizeList lists has a text field underneath the list. And there is a third text field underneath the set of radio buttons. Choose the Edit Text tool from the VA's **Tool** menu, and draw each of these fields. When editable text fields are placed into dialog windows, the VA automatically creates a framed field of the CDialogText class (rather than the CEditText field that is created for normal windows). Make sure that the fields under the lists are tall enough for a single 12-point line of text and that the field under the radio buttons is about 64 pixels tall (we show a sample of the text, the way it will appear in the Main view, in this field).

Creating the OK and Cancel Buttons

Choose the Button tool from the VA's **Tool** menu, and click to create the OK button. The first button you create will be named

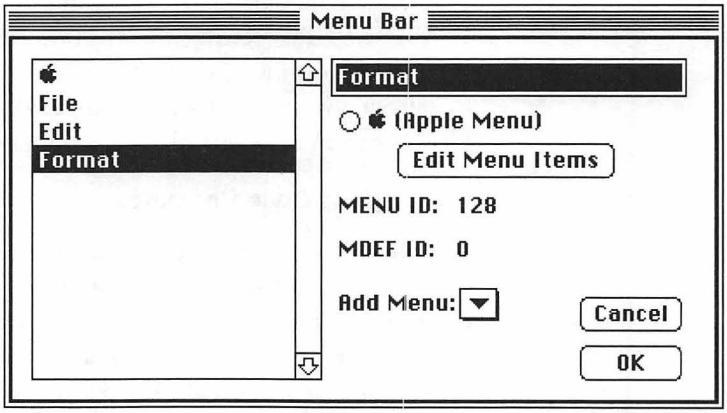

Figure 5-5
Creating the Format menu

OK, automatically; the second button you create will be named Cancel, automatically. The OK button will be outlined as the default button for the dialog, automatically.

Creating the Labels

Each of the labels (Font, Size, Style, and Justification) is created with the VA's Static Text tool. Just use the default font, size, and style for these labels to duplicate their appearance in Figure 5-3.

Creating the Format Menu

In order to display the Notebook dialog, we need to provide the means for the user to choose to do so. Our approach will be to add a **Format** menu to the Notebook project's menu bar.

Choose **Menu Bar** from the VA's **Edit** menu to display the Menu Bar window. Choose **New Menu** from the **Edit** menu (or use the Command-K keyboard shortcut), and enter the name **Format** for the new menu. The result is shown in Figure 5-5.

Click the Edit Menu Items button in the Menu Bar window to display the Menu Items dialog. Choose **New Menu Item** from the VA's **Edit** menu, and enter **Notebook...** as its name (the ellipsis is created by using the Option-; [semicolon] keyboard character).

While the Menu Items dialog is still being shown, click the pop-up menu next to the word "Command" and choose "Other." When the Commands window is displayed, choose **New Command** from the **Edit** menu, and create the `cmdNotebook` com-

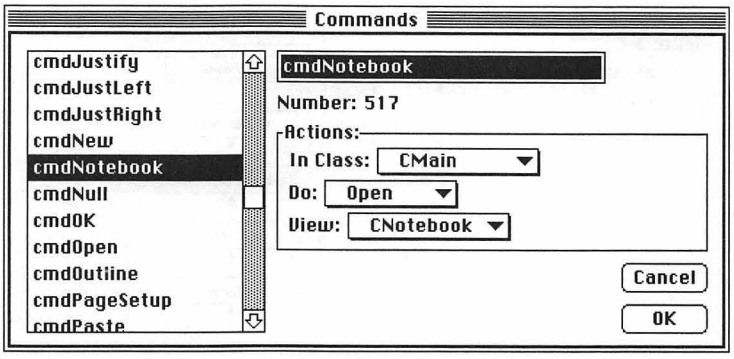

Figure 5-6
Creating the Notebook command

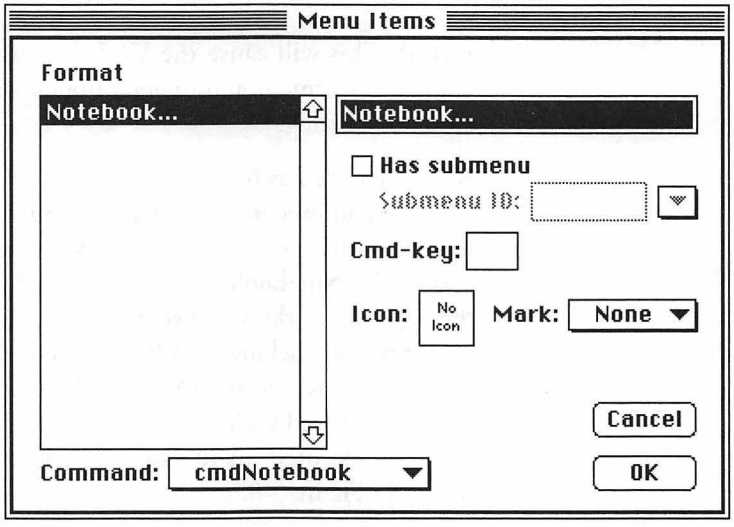

Figure 5-7
Completed Notebook menu command for the Format menu

mand for the Notebook menu item, making sure that it corresponds to what is shown in Figure 5-6.

After you have created the `cmdNotebook` command, dismiss the Commands dialog by clicking the OK button. The Menu Items window should still be on your screen, and its appearance should correspond to what is shown in Figure 5-7.

Generating and Running the Skeleton Code

After the Notebook view has been created, you can save the VA file and generate code by choosing the **Generate All** command from the Project menu (the menu to the right of the **Windows**

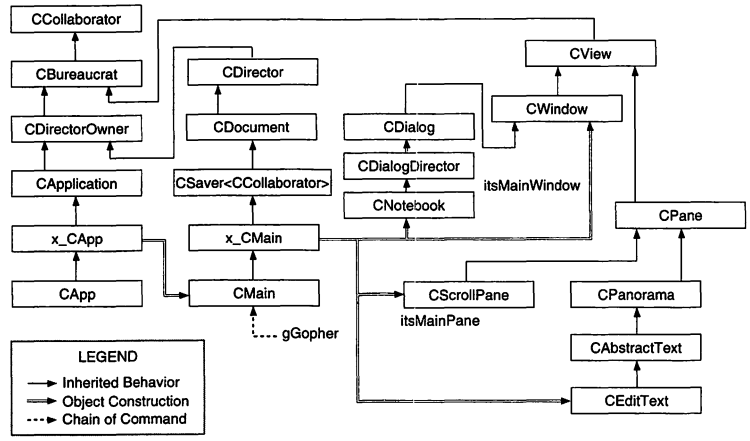

Figure 5-8
Dynamic structure of application with Notebook dialog

menu). This will cause the VA to generate the default skeleton code to implement and operate the user interface, including the Notebook dialog.

After the code has been generated, you can quit the VA and then compile and execute the newly created code. Note when you do that you can type text into the Main window, and when you choose the **Notebook** command from the **Format** menu, each of the controls works as expected. Of course, the lists are empty of contents, and clicking the OK button to dismiss the dialog has no effect on the text in the Main window, but that functionality will have to be added with our custom code. When the skeleton application is executed, its dynamic structure corresponds to what is shown in Figure 5-8.

The figure is not quite accurate, but the major objects are shown. The base class for the Notebook dialog (x_CNotebook) is not shown, nor are any of the objects (controls, static text, edit text, and so on) that populate the dialog. All of the controls and other window contents are created by the MakeNewWindow function of the x_CNotebook class.

Instead of examining the skeleton code and then looking at how it needs to be customized to create a fully functional dialog, we'll jump right to the bottom line and examine the final form of the code related to the dialog, as it has been customized in both the CMain and CNotebook classes. In so doing, you will gain a better understanding of how the various components of the document, its window, and the dialog work together.

Examining the Custom CApp Code

The only change that I have made to the CApp (application) code is to access the Font menu that VA provides and populate it with all of the user's fonts in the SetUpMenus function.

SetUpMenus Function Code

The code for the SetUpMenus function is as follows:

```
void CApp::SetUpMenus()
{
   MenuHandle macMenu;

   // Override SetUpMenus to install a FONT menu into the
   // bartender's list, containing all of the current font names

   x_CApp::SetUpMenus();
   macMenu = GetMenu (MENUfont);
   FailNILRes (macMenu);
   AddResMenu (macMenu, 'FONT');
   gBartender->AddMenu (MENUfont, TRUE, hierMenu);
}
```

The foregoing code calls the SetUpMenus function for its base class and then accesses the Font menu from the VA-generated resources and loads all of the user's font names into it using the AddResMenu toolbox call. The menu is added to the bartender's list, but as though it is a hierarchical menu (`hierMenu`, which tells the bartender not to redraw the menu bar).

Examining the Custom CMain Code

If you recall, the Main view contains a CEditText object to hold the text that the user types. The object has an associated scroll bar that will activate to allow the user to enter up to 32,000 bytes of text. The view is not intended to be a text editor, but more like a simple notebook for recording text in a particular font, size, style, and justification. The main point of the code is to show how a fairly complex dialog can be constructed to operate in conjunction with a text window.

Defining the CTextSettings Structure

In order to associate the text with its characteristics, I have defined a structure to hold the font name and its size, style, and justification values. The structure is defined in a separate header file that

was created manually with the Symantec editor. The file is named **CTextSettings.h** and its contents are as follows:

```
/************************************************************
 CTextSettings.h

   Font settings for text in dialog
************************************************************/

#pragma once

struct CTextSettings
{
   Str255  spFontName;
   Str255  spFontSize;
   short   nFontStyle;
   short   nFontJust;

};
```

Although I could have defined the CTextSettings structure inside the **CMain.h** header file, I chose for it to be separate because of the Object I/O support that we will be adding to this application in a later chapter and also because the structure must be known to both the CMain class and its CNotebook dialog.

The CMain Header File Contents

The **CMain.h** header file has been modified by adding one new public function, two protected functions that override the corresponding base class functions, and also a newly added function to update the window text's appearance. In addition, the header file contains a private member variable that holds the CTextSettings data. The contents of the **CMain.h** header file are as follows:

```
/************************************************************
 CMain.h

        Header File For CMain Document Class

   Copyright © 1994 Richard O. Parker. All rights reserved.
************************************************************/

#pragma once

#include "x_CMain.h"
#include "CTextSettings.h"

class CMain : public x_CMain
{
public:

   TCL_DECLARE_CLASS
```

```
   void    ICMain(void);

   virtual void    MakeNewContents(void);
   virtual void    ContentsToWindow(void);
   virtual void    WindowToContents(void);
   virtual void ExchangeSettings(          // newly added
      CTextSettings& itssettings,          //    settings
      Boolean bFromDialog);                //    from dialog?

protected:
   virtual void    DoCmdNotebook(void);     // override
   virtual void    UpdateWindowText(void); // newly added
   virtual void    MakeNewWindow(void);     // override

private:
   CTextSettings   settings;                // newly added
};

// If you have multiple document classes, you must change
// the file type below to the appropriate type for this class.
// If not, this #define is not used.

#define CMainFType 'TEXT'
```

The added functions and `settings` member variable are accompanied with comments that indicate whether they override existing code in the base class or are newly added. The `settings` variable is accessible only within the CMain class.

ICMain Initialization Function Code

I have modified the ICMain function to initialize the contents of the `settings` structure when the document object is created. This is so we will have some initial settings to use for a new document. The code is as follows:

```
void CMain::ICMain()
{
   Ix_CMain();

   // Initialize Font Settings

   TCLpstrcpy(settings.spFontName, "\pChicago");
   TCLpstrcpy(settings.spFontSize, "\p12");
   settings.nFontStyle = normal;
   settings.nFontJust  = teFlushLeft;
}
```

The first action of the foregoing code is to call the Ix_CMain base class initialization function. Then the code initializes the contents of each of the fields in the `settings` structure.

DoCommand Function Code

The code to intercept the command to open the Notebook dialog is generated into the DoCommand function for the x_CMain class. That code remains unchanged, as it was generated by the VA, and is as follows:

```
void x_CMain::DoCommand(long theCommand)
{
   switch (theCommand)
   {
      case cmdNotebook:
         DoCmdNotebook();
         break;
      default:
         CDocument::DoCommand(theCommand);
   }
}
```

The foregoing code tests only whether the function has been called to handle the `cmdNotebook` command and if so, it calls a function named DoCmdNotebook to handle the command. All other commands are passed on to the CDocument base class to handle.

DoCmdNotebook Override Function Code

Though the VA has generated skeleton code for the DoCmdNotebook function in the x_CMain class, I have chosen to override that function in the CMain class (as indicated in the **CMain.h** header file). The VA-generated code merely creates the Notebook dialog and "runs" it, not taking into consideration what function it might perform. The newly added override function in the CMain class is as follows:

```
void CMain::DoCmdNotebook()
{
   CNotebook  *dialog;

   // Respond to command by creating a dialog object
   dialog = TCL_NEW(CNotebook,());

   // create the dialog window, with its controls
   dialog->ICNotebook(this);

   // run the dialog
   if (dialog->DoModalDialog(cmdNull) == cmdOK)
   {
      // update the window's text to correspond with
      // the newly specified settings.

      UpdateWindowText();
```

```
        SetChanged (TRUE);
    }
    ForgetObject(dialog);
}
```

As the foregoing code shows, after creating the CNotebook object, the ICNotebook initialization function is called, causing the dialog's window to be created with initial settings for the values of the controls, lists, and EditText fields.

Initializing the Dialog

The ICNotebook function causes a series of other functions to be called. The code for the function is as follows:

```
void CNotebook::ICNotebook(CDirectorOwner *aSupervisor)
{
    // access the document to retrieve the current settings to use
    // for initializing the controls in the dialog.

    ((CMain *)aSupervisor)->ExchangeSettings (itsSettings, FALSE);

    // create the window and initialize the controls with the
    // settings transferred from the document.

    x_CNotebook::Ix_CNotebook(aSupervisor);
}
```

The foregoing initialization code commences its execution by calling a function named ExchangeSettings, located in the supervisor of the dialog (which, in this case, is the document—the CMain class). The purpose of the ExchangeSettings function is to transfer the font information settings either from the document to the dialog or to the document from the dialog. In either case, the first argument to the function is a reference to the dialog's CSettings structure (itsSettings).The second argument specifies whether the settings are to be transferred from the document to the dialog (FALSE) or from the dialog to the document (TRUE). The ICNotebook function calls the function with a value of FALSE, to cause the document's initial settings to be transferred to the dialog before the dialog's window (and its associated controls, lists, and fields) is created.

ExchangeSettings Function Code

The ExchangeSettings function is located in the CMain class and is newly added (custom) code. The function is quite simple and the code is as follows:

```
void CMain::ExchangeSettings (CTextSettings& itsSettings,
   Boolean bFromDialog)
{
   if (bFromDialog)
   {
      //
      // move settings from the dialog to the document
      //
      settings = itsSettings;
   }
   else
   {
      //
      // move settings from the document to the dialog
      //
      itsSettings = settings;
   }
}
```

The foregoing code tests the value of the `bFromDialog` variable and then either transfers the contents of the `itsSettings` argument to the document's `settings` structure or transfers the contents of the document's `settings` structure to the `itsSettings` argument.

Examining the Notebook Dialog Code

Ix_CNotebook Function Code

The ICNotebook function calls the Ix_CNotebook function to finalize the initialization of the dialog. The VA-generated code for that function ((in the x_CNotebook class) is as follows:

```
void x_CNotebook::Ix_CNotebook(CDirectorOwner *aSupervisor,
   Boolean push)
{
   IDialogDirector(aSupervisor);

   // There are several circumstances where we don't want
   // ProviderChanged to be called. During initialization,
   // during calls to UpdateData, etc. The ignore flag
   // heads these off.

   ignore = TRUE;      /* Don't call UpdateData now    */
   MakeNewWindow();    /* Create the dialog's window   */
   DoBeginData(push); /* Gather initial values        */
   ignore = FALSE;
}
```

The first act of the foregoing code is to call the IDialogDirector function for the x_CNotebook's base class (CDialogDirector). This function call results in a chain of calls to initialize each of the classes in the CNotebook class's hierarchy (that is, IDialogDirector

calls IDirector, which calls IDirectorOwner, which calls IBureaucrat, which finally stores the `aSupervisor` value, contained in the original call to ICNotebook, into the `itsSupervisor` variable). The functionality of most of the foregoing initialization functions is described in Chapter 2, beginning on page 31, in step 6 (with regard to the application object as the supervisor in that case). The IDialogDirector function merely sets the `dismissCmd` (dismiss command) value to `cmdNULL` and then calls the IDirector function.

The Ix_CNotebook function continues by setting the `ignore` member variable of the x_CNotebook class to `TRUE`. The explanation given in the comments for this action is that it is to prevent the ProviderChanged function from being called when a change is made to any of the window's "collaborators" (controls, lists, text fields) when it is initialized. I will discuss how the collaboration mechanism works in a later chapter. For now, just bear in mind that setting the `ignore` variable to `TRUE` prevents inappropriate ProviderChanged actions from being taken until it is set to `FALSE` once again.

MakeNewWindow Function Code

The Ix_CNotebook function continues by calling the MakeNewWindow function, whose VA-generated code is as follows:

```
void x_CNotebook::MakeNewWindow(void)
{
   itsWindow = TCLGetNamedWindow("\pNotebook", this);

   // Initialize pointers to the subpanes in the window
   fNotebook_OkButton =
      (CButton*) FindPane(kNotebook_OkButtonID);
  ASSERT(member(fNotebook_OkButton, CButton));

  fNotebook_CancelButton =
      (CButton*) FindPane(kNotebook_CancelButtonID);
  ASSERT(member(fNotebook_CancelButton, CButton));

  fNotebook_FontLabel =
      (CStaticText*) FindPane(kNotebook_FontLabelID);
  ASSERT(member(fNotebook_FontLabel, CStaticText));

  fNotebook_SizeLabel =
      (CStaticText*) FindPane(kNotebook_SizeLabelID);
  ASSERT(member(fNotebook_SizeLabel, CStaticText));

  fNotebook_StyleLabel =
      (CStaticText*) FindPane(kNotebook_StyleLabelID);
  ASSERT(member(fNotebook_StyleLabel, CStaticText));
   fNotebook_JustLabel =
      (CStaticText*) FindPane(kNotebook_JustLabelID);
```

```
    ASSERT(member(fNotebook_JustLabel, CStaticText));

    fNotebook_FontList =
        (CFontList*) FindPane(kNotebook_FontListID);
    ASSERT(member(fNotebook_FontList, CFontList));

    fNotebook_SizeList =
        (CSizeList*) FindPane(kNotebook_SizeListID);
    ASSERT(member(fNotebook_SizeList, CSizeList));

    fNotebook_BoldCheck =
        (CCheckBox*) FindPane(kNotebook_BoldCheckID);
    ASSERT(member(fNotebook_BoldCheck, CCheckBox));

    fNotebook_ItalicCheck =
        (CCheckBox*) FindPane(kNotebook_ItalicCheckID);
    ASSERT(member(fNotebook_ItalicCheck, CCheckBox));

    fNotebook_UnderlineCheck =
        (CCheckBox*) FindPane(kNotebook_UnderlineCheckID);
    ASSERT(member(fNotebook_UnderlineCheck, CCheckBox));

    fNotebook_OutlineCheck =
        (CCheckBox*) FindPane(kNotebook_OutlineCheckID);
    ASSERT(member(fNotebook_OutlineCheck, CCheckBox));

    fNotebook_ShadowCheck =
        (CCheckBox*) FindPane(kNotebook_ShadowCheckID);
    ASSERT(member(fNotebook_ShadowCheck, CCheckBox));

    fNotebook_CondenseCheck =
        (CCheckBox*) FindPane(kNotebook_CondenseCheckID);
    ASSERT(member(fNotebook_CondenseCheck, CCheckBox));

    fNotebook_ExtendCheck =
        (CCheckBox*) FindPane(kNotebook_ExtendCheckID);
    ASSERT(member(fNotebook_ExtendCheck, CCheckBox));

    fNotebook_Rect16 =
        (CRectOvalButton*) FindPane(kNotebook_Rect16ID);
    ASSERT(member(fNotebook_Rect16, CRectOvalButton));

    fNotebook_LeftRadio =
        (CRadioControl*) FindPane(kNotebook_LeftRadioID);
    ASSERT(member(fNotebook_LeftRadio, CRadioControl));

    fNotebook_JustCenterRadio =
        (CRadioControl*) FindPane(kNotebook_JustCenterRadioID);
    ASSERT(member(fNotebook_JustCenterRadio, CRadioControl));

    fNotebook_JustRightRadio =
        (CRadioControl*) FindPane(kNotebook_JustRightRadioID);
    ASSERT(member(fNotebook_JustRightRadio, CRadioControl));

    fNotebook_FontName =
        (CDialogText*) FindPane(kNotebook_FontNameID);
    ASSERT(member(fNotebook_FontName, CDialogText));

    fNotebook_FontSize =
        (CDialogText*) FindPane(kNotebook_FontSizeID);
    ASSERT(member(fNotebook_FontSize, CDialogText));

    fNotebook_FontSample =
        (CDialogText*) FindPane(kNotebook_FontSampleID);
    ASSERT(member(fNotebook_FontSample, CDialogText));
}
```

Although the foregoing code is quite lengthy, it is quite simple. The statements use the VA's Object I/O features to read in the contents of the 'CVue' resource that describes the dialog and its contents, create each element in the process, and store its pointer into a member variable of the x_CNotebook class. After each of the window's subpanes is created, an ASSERT statement validates the class of the subpane's pointer, just to ensure that the initialization process is robust. When the MakeNewWindow function's execution is complete, the window and all of its subpanes have been created, but the window has not yet been made visible to the user.

DoBeginData Function Code

The initialization process continues in the Ix_CNotebook function, after MakeNewWindow returns, by calling the DoBeginData function with the value of an argument named push. If you look back at the Ix_CNotebook function definition (see page 184), you will see that the variable push is the second argument to that function; however, if you look at the ICNotebook function code that calls the Ix_CNotebook function, you will see that it doesn't supply a value for this second argument. This situation illustrates an interesting feature of the C++ language (the facility for including *default values* in function declarations). If you look into the **x_CNotebook.h** header file, you will see that the Ix_CNotebook function has been declared as follows:

```
void Ix_CNotebook(CDirectorOwner *aSupervisor,
   Boolean push = FALSE);
```

Note in the foregoing that the push argument to the function has been given a default value of FALSE. It will take on this value if another value is not supplied for the argument. This is exactly the case when our ICNotebook function calls the function. Also note that when the Ix_CNotebook function calls the DoBeginData function, it passes the value of the push variable to that function. The VA-generated code for DoBeginData is as follows:

```
void x_CNotebook::DoBeginData(Boolean push)
{
   CNotebookData   data = {0};   // The initial value record
   BeginData(&data);             // Ask subclass for initial values

   if (!push)
   {
```

```
      DispensePaneValues(data);  // Set panes
   }
   // Save the initial values in case user cancels
   saveData = data;
}
```

Note that the foregoing code begins by setting the value of a temporary data structure named `data` to 0. The CNotebookData structure is declared in the **x_CNotebook.h** header file to contain the values for each of the controls and fields in the dialog. The structure declaration is as follows:

```
typedef struct
{
        /* Array pane (table)*/
    Point  fNotebook_FontList;
        /* Array pane (table)*/
    Point  fNotebook_SizeList;
        /* Control (radio or checkbox)*/
   short   fNotebook_BoldCheck;
        /* Control (radio or checkbox)*/
   short   fNotebook_ItalicCheck;
        /* Control (radio or checkbox)*/
   short   fNotebook_UnderlineCheck;
        /* Control (radio or checkbox)*/
   short   fNotebook_OutlineCheck;
        /* Control (radio or checkbox)*/
   short   fNotebook_ShadowCheck;
        /* Control (radio or checkbox)*/
   short   fNotebook_CondenseCheck;
        /* Control (radio or checkbox)*/
   short   fNotebook_ExtendCheck;
   short   fNotebook_Rect16;
        /* Control (radio or checkbox)*/
   short   fNotebook_LeftRadio;
        /* Control (radio or checkbox)*/
   short   fNotebook_JustCenterRadio;
        /* Control (radio or checkbox)*/
   short   fNotebook_JustRightRadio;
        /* Dialog text  */
   Str255  fNotebook_FontName;
        /* Dialog text  */
   Str255  fNotebook_FontSize;
        /* Dialog text  */
   Str255  fNotebook_FontSample;
} CNotebookData;
```

As you can see in the foregoing definition, the CNotebookData structure contains a field for each of the dialog's elements. I didn't want to save all of this data, and that is why I defined the CTextSettings structure (see page 179) to hold only the values that I was interested in saving from one invocation of the dialog to the next. The VA-generated code keeps track of *all* of the dialog's settings in the foregoing structure.

BeginData Function Code

The DoBeginData function continues by calling BeginData with a pointer to the temporary `data` structure as its argument. The CNotebook class contains our highly modified version of this function. The purpose of the code is to initialize the fields in the CNotebookData structure with the settings passed to the dialog from the document object. The code is quite lengthy and I will describe it in sections, beginning as follows:

```
void CNotebook::BeginData(CNotebookData *initial)
{
   StringPtr pFontName, pFontSize;
   short index;

   // Define values for CNotebookData fields, starting
   // with the font style value.

   initial->fNotebook_BoldCheck = 0;
   initial->fNotebook_ItalicCheck = 0;
   initial->fNotebook_UnderlineCheck = 0;
   initial->fNotebook_OutlineCheck = 0;
   initial->fNotebook_ShadowCheck = 0;
   initial->fNotebook_CondenseCheck = 0;
   initial->fNotebook_ExtendCheck = 0;

   if (itsSettings.nFontStyle & bold)
      initial->fNotebook_BoldCheck = 1;
   if (itsSettings.nFontStyle & italic)
      initial->fNotebook_ItalicCheck = 1;
   if (itsSettings.nFontStyle & underline)
      initial->fNotebook_UnderlineCheck = 1;
   if (itsSettings.nFontStyle & outline)
      initial->fNotebook_OutlineCheck = 1;
   if (itsSettings.nFontStyle & shadow)
      initial->fNotebook_ShadowCheck = 1;
   if (itsSettings.nFontStyle & condense)
      initial->fNotebook_CondenseCheck = 1;
   if (itsSettings.nFontStyle & extend)
      initial->fNotebook_ExtendCheck = 1;
```

The foregoing section of the code sets the values in the `data` structure for each of the checkbox controls to 0, thereby turning them off. The code continues by testing whether a particular checkbox should be checked, by testing the value of the font style that was stored into the `itsSettings` variable when the ExchangeSettings function was called to acquire this value from the CMain object. The BeginData code continues as follows:

```
   // now, initialize the text justification setting

   initial->fNotebook_LeftRadio = 0;
   initial->fNotebook_JustCenterRadio = 0;
   initial->fNotebook_JustRightRadio = 0;
```

```
switch (itsSettings.nFontJust)
{
   case teFlushLeft:
   {
      initial->fNotebook_LeftRadio = 1;
      break;
   }
   case teCenter:
   {
      initial->fNotebook_JustCenterRadio = 1;
      break;
   }
   case teFlushRight:
   {
      initial->fNotebook_JustRightRadio = 1;
      break;
   }
   default:
   {
      initial->fNotebook_LeftRadio = 1;
      break;
   }
}
```

The foregoing code sets the state of each of the dialog's radio button values to 0 (off) and then tests the acquired initial settings to determine the current text justification setting and select the appropriate radio button in the dialog. The BeginData code continues as follows:

```
// next, create the array to receive the font names, set the
// array pointer into the fNotebook_FontList list, and then
// load the font names into the array to cause them to be
// displayed.

itsFontArray = TCL_NEW (CArray, (sizeof (StringPtr)));
fNotebook_FontList->SetArray (itsFontArray, TRUE);
MenuHandle fontMenu = GetMHandle (MENUfont);
short numFonts = CountMItems (fontMenu);

for (index=1; index <= numFonts; index++)
{
   Str255 spMenuText;
   GetItem (fontMenu, index, spMenuText);
   pFontName = (unsigned char *)malloc ((long) spMenuText[0] +1);
   FailNIL (pFontName);
   TCLpstrcpy(pFontName, spMenuText);
   itsFontArray->Add (&pFontName);
}
```

As the comments that precede the foregoing code indicate, that section of the code is responsible for loading the dialog's font list array with the names of all of the fonts that are available on the user's system. It does this by accessing the handle to a Font menu that was created within the VA and then populated with the systems's fonts in the SetUpMenus function of the CApp class (see

page 179). It operates by accessing the Font menu for each font name and then adding each entry into the `itsFontArray` array. The BeginData code continues as follows:

```
// create an array to receive the font sizes, set the array
// pointer into the fNotebook_SizeList list, and then load
// the sizes into the array to cause them to be displayed.

unsigned char *p;
itsSizeArray = TCL_NEW (CArray, (sizeof (StringPtr)));
fNotebook_SizeList->SetArray (itsSizeArray, TRUE);
for (index=0; fontSizes[index] != NULL; index++)
{
   itsSizeArray->Add (&fontSizes[index]);
}
```

The foregoing code adds a list of predefined font sizes to the size list. The font sizes were created as an array of Pascal strings in the CSizeList class (the CArrayPane-derived class for the size list control in the dialog). We will look at the code for the CSizeList class shortly. The BeginData function code continues as follows:

```
// set up the initial selections in the font and size lists
// and then put the list item values into the text fields.

SetPt (&initial->fNotebook_FontList, 0, 0);
CArrayIterator fIter (itsFontArray, kStartAtBeginning);
for (index=0; fIter.Next (&pFontName); index++)
{
   if (IUCompString(itsSettings.spFontName, pFontName) == 0)
   {
      SetPt (&initial->fNotebook_FontList, 0, index);
      break;
   }
}
TCLpstrcpy (initial->fNotebook_FontName, itsSettings.spFontName);

SetPt (&initial->fNotebook_SizeList, 0, 0);
CArrayIterator sIter (itsSizeArray, kStartAtBeginning);
for (index=0; sIter.Next (&pFontSize); index++)
{
   if (IUCompString(itsSettings.spFontSize, pFontSize) == 0)
   {
      SetPt (&initial->fNotebook_SizeList, 0, index);
      break;
   }
}
TCLpstrcpy (initial->fNotebook_FontSize, itsSettings.spFontSize);
```

The foregoing code iterates through the array of font names, looking for the font that was specified as the "current" font in the settings. When it is found, the index into the array is stored as the cell to be selected in the Font list control's data field. If the font

isn't found, the first font name will be selected. In a similar fashion, the size list array is searched for the size that was specified in the initial settings. If it is found, the corresponding entry in the size list control's data field is selected. If the entry is not found, then the first size list entry will be selected. The code for the BeginData function concludes as follows:

```
   // set the font sample field to "Sample", set its font name,
   // size, style, and justification, and then draw the sample.

   TCLpstrcpy (initial->fNotebook_FontSample, "\pSample");
   fNotebook_FontName->SetTextString (itsSettings.spFontName);
   fNotebook_FontSize->SetTextString (itsSettings.spFontSize);
   fNotebook_FontSample->SetFontStyle (itsSettings.nFontStyle);
   fNotebook_FontSample->SetAlignment (itsSettings.nFontJust);
   DrawSample();
}
```

The last act of the BeginData function is to store the final values into the controls for the FontSample, FontName, FontSize, Font Style, and FontJust controls and then call a function named DrawSample to render the word "Sample" in the corresponding font, size, style, and justification in the EditText field underneath the radio button controls.

DrawSample Function Code

The DrawSample function is newly added and is as follows:

```
void CNotebook::DrawSample ()
{
   Str255 theFontText;
   short  theFontNum;
   short  theFontSize;

   if (fNotebook_FontName->GetLength() > 0)
   {
      fNotebook_FontName->GetTextString (theFontText);
      GetFNum (theFontText, &theFontNum);
   }
   else
   {
      theFontNum = systemFont;
   }
   if (fNotebook_FontSize->GetLength() > 0)
   {
      fNotebook_FontSize->GetTextString (theFontText);
      theFontSize = atoi ((const char *)(&theFontText[1]));
      if (theFontSize <= 0 || theFontSize > 72)
      {
         theFontSize = 12;
      }
   }
   else
```

```
    {
        theFontSize = 12;
    }
    fNotebook_FontSample->SetFontNumber (theFontNum);
    fNotebook_FontSample->SetFontSize (theFontSize);
    fNotebook_FontSample->SetTextString ("\pSample");
}
```

The DrawSample function doesn't actually draw anything. It updates the font and size specifications for the FontSample field, which causes the TCL to redraw the contents of the field with the new settings. The style and justification settings have already been made in the BeginData function and will be changed dynamically, causing the FontSample field to be redrawn, as the user interacts with the dialog.

After the DrawSample function returns to BeginData, it, in turn, returns control to the DoBeginData function described beginning on page 187. The DoBeginData function continues by testing whether the value of the push variable is TRUE or FALSE. If it is FALSE, then the DispensePaneValues function is called to set the dialog's subpanes to the values contained in the fields of the CNotebookData structure. This is the case, by default, for the DoBeginData function.

DispensePaneValues Function Code

The DispensePaneValues function code is generated by the VA to take each field in the NotebookData structure and store its value into the corresponding control in the dialog, as follows:

```
void x_CNotebook::DispensePaneValues(const CNotebookData& data)

{
    // Initialize the panes based on the values supplied.
    // The ASSERT statements ensure that the generated
    // code is in synch with the view resource.

    fNotebook_FontList->SelectCell(
        *(Cell*) &data.fNotebook_FontList, FALSE, TRUE);

    fNotebook_SizeList->SelectCell(
        *(Cell*) &data.fNotebook_SizeList, FALSE, TRUE);

    fNotebook_BoldCheck->SetValue(data.fNotebook_BoldCheck);

    fNotebook_ItalicCheck->SetValue(data.fNotebook_ItalicCheck);

    fNotebook_UnderlineCheck->SetValue(data.fNotebook_UnderlineCheck);

    fNotebook_OutlineCheck->SetValue(data.fNotebook_OutlineCheck);

    fNotebook_ShadowCheck->SetValue(data.fNotebook_ShadowCheck);
```

```
    fNotebook_CondenseCheck->SetValue(data.fNotebook_CondenseCheck);
    fNotebook_ExtendCheck->SetValue(data.fNotebook_ExtendCheck);
    fNotebook_LeftRadio->SetValue(data.fNotebook_LeftRadio);
    fNotebook_JustCenterRadio->SetValue(data.fNotebook_JustCenterRadio);
    fNotebook_JustRightRadio->SetValue(data.fNotebook_JustRightRadio);
    fNotebook_FontName->SetTextString(data.fNotebook_FontName);
    fNotebook_FontSize->SetTextString(data.fNotebook_FontSize);
    fNotebook_FontSample->SetTextString(data.fNotebook_FontSample);
}
```

After the DispensePaneValues function completes execution and returns to the DoBeginData function, all of the dialog's controls will contain the current settings. The DoBeginData function concludes execution by storing the values in its temporary `data` structure into a corresponding instance of the CNotebookData structure called `saveData`. This is so the initial values can be restored if the user decides to cancel the dialog after the settings have been changed.

When the DoBeginData function returns, the Ix_CNotebook function regains control and concludes its own execution by setting the value of the `ignore` variable to `FALSE`. This will allow any further changes to the controls or fields in the dialog to be acted upon in the ProviderChanged member function.

When the Ix_CNotebook function returns, the DoCmdNotebook function of the CMain class regains control. The dialog is still not visible to the user, so the DoCmdNotebook function continues its own execution (you can refer back to pages 182–183 to see this code) by calling the DoModalDialog function for the dialog object. This causes the dialog to be made visible and for a modal event loop to be executed. The user can interact with the dialog, change the values of any of its controls, and then dismiss the dialog either with the OK or Cancel buttons. Until the dialog is dismissed, no other actions can be performed.

Before examining the functions that are involved with the user's dynamic changes to the dialog, it might be instructive to recap the sequence of initialization actions. Here's what happened:

1. The user chooses the **Notebook** command from the **Format** menu, causing a `cmdNotebook` command to be sent to the current gopher's DoCommand function. At this point in time, the current gopher is the document object (CMain).

2. The DoCommand function recognizes the `cmdNotebook` command and calls the DoCmdNotebook function to handle the command.

3. DoCmdNotebook creates the CNotebook (dialog director) object and calls its ICNotebook function.

4. The CNotebook object's ICNotebook function calls the document's ExchangeSettings function to acquire the current (previous) settings from the document, storing these into the `itsSettings` structure that we have defined. The structure definition is contained in the **CTextSettings.h** file.

5. Next the ICNotebook function calls the Ix_CNotebook function in its base class to continue the initialization process.

6. The Ix_CNotebook function calls MakeNewWindow to create the document's window and its subpanes and then calls the DoBeginData function.

7. DoBeginData calls the BeginData function in the CNotebook class that we modified heavily to convert the document's settings into values for each of the dialog's fields in the `data` structure that is local to the DoBeginData function. The values stored into the fields of the `data` structure are used to initialize the corresponding controls (subpanes) in the dialog with the values we have supplied.

8. DoBeginData returns to the Ix_CNotebook function, which returns to the ICNotebook function, which returns to the DoCmdNotebook function in the CMain class. That function calls DoModalDialog to "run" the dialog.

Examining the Code for Running the Notebook Dialog

Running of the dialog commences when the DoModalDialog function is called by the DoCmdNotebook function of the CMain class. When control returns to the DoCmdNotebook function, the dialog will have been dismissed by the user by clicking either its OK or Cancel button.

DoModalDialog Function Code

The x_CNotebook class contains an override of the DoModalDialog function inherited from its CDialogDirector base class. The override code is as follows:

```
long x_CNotebook::DoModalDialog(long defaultCmd)
{
   long result = CDialogDirector::DoModalDialog(defaultCmd);

   DoEndData(result);
   return result;
}
```

The purpose of overriding the DoModal dialog function is so that the foregoing function can regain control after the dialog has been dismissed, but before control returns to the DoCmdNotebook function in the CMain class. The foregoing function calls the DoModalDialog function in the CDialogDirector class of the TCL to perform the major task of running the dialog. That function calls DoChangeableDialog (in that same class) to perform the actions of managing the dialog. The code in the TCL for the DoChangeableDialog function is very instructive and is as follows:

```
long CDialogDirector::DoChangeableModalDialog(long defaultCmd,
   Boolean changeDoc)
{
   CDialog       *itsDialog = (CDialog*) itsWindow;
   tGopherInfo   gopherInfo;
   Boolean       wasChanged = GetChanged();

      // Setup the desired default command, and make sure
      // the window is modal and visible.

   itsDialog->SetDefaultCmd(defaultCmd);
   itsDialog->SetModal(kModal);
   DisableTheMenus();    // (2.0.4)
   itsDialog->Select();

      // Set the initial gopher

   itsDialog->FindGophers(&gopherInfo);
   itsGopher = gopherInfo.firstGopher? gopherInfo.firstGopher
      : itsDialog;

   if (member(itsGopher, CAbstractText))
      ((CAbstractText*)itsGopher)->SelectAll(TRUE);

   if (active)
      itsGopher->BecomeGopher(TRUE);

   dismissCmd = cmdNull;    // So DoModalDialog can be repeated

   while (dismissCmd == cmdNull)
   {
```

```
      try_  // post an exception handler to prevent exceptions from
      {     // prematurely aborting the dialog
         do
         {
            gApplication->Process1Event();

         } while(dismissCmd == cmdNull);

      }
      catch_(CException, thrown)
      {
         HiliteMenu(0);
         if (thrown->GetErr() != kSilentErr)
            ErrorAlert(thrown->GetErr(), thrown->GetMsg());
      }
      end_try_

   }

   EnableTheMenus();

      // Restore the changed state if the dialog was canceled
      // or if changes are not allowed

   if (dismissCmd == cmdCancel || !changeDoc)
      SetChanged(wasChanged);

   return dismissCmd;
}
```

The DoChangeableModalDialog function permits you to have changes in the dialog reflect the "changed" status of the document. The value of the `changeDoc` argument to the function (that is set to `FALSE`, by default, in the function's prototype) determines whether the document's status is affected by the user's actions in the dialog.

After setting the value of the `defaultCmd` (to `cmdNULL` in our case), calling the SetModal function to ensure that the window is treated as a modal dialog, and calling DisableTheMenus to disable all of the menus in the menu bar, the function calls the Select function to (finally) make the window visible and allow user interaction with the dialog.

Before commencing to execute the private event loop, the DoChangeableModalDialog function examines the possible gophers in the dialog, then tests whether the first gopher is a member of, or derived from, the CAbstractText class (meaning that it is a text field). If so, then the contents of the field are selected, causing it to become hilited.

Finally the DoChangeableModalDialog function commences executing an event loop (within a "try-catch" block to protect against exception errors) by repeatedly calling the application object's Process1Event function, until the `dismissCmd` variable is found to contain something other than the (initial) `cmdNull` value.

While the foregoing loop is running, the user is allowed to perform any allowable actions with the dialog's controls or fields. Such actions as clicking a checkbox or a radio button, or selecting a font name or size from their respective lists results in an event being posted to the application's event queue. The Process1Event function removes one event from the queue and calls the Switchboard object's ProcessEvent function to handle the event. We will cover events in a later chapter, but the end result of the user's interaction with the dialog's controls, in this case, is that the ProviderChanged function in the x_CNotebook class is called. This is because each of the controls and fields is descended from the CCollaborator class in the TCL, and when its state is changed, it calls its BroadcastChange function. If the control doesn't override the BroadcastChange function, then the BroadcastChange function of the CBureaucrat class will handle the call. In that case, the ProviderChanged function in the control's supervisor (in this case it's the CNotebook object, in the CDialogDirector-derived class) is called to perform any functions necessary to handle the change to the control's state.

x_CNotebook ProviderChanged Function Code

The VA-generated code for the ProviderChanged function of the x_CNotebook class handles changes to all of the fields and controls in the dialog. The code is quite lengthy and is as follows:

```
void x_CNotebook::ProviderChanged(CCollaborator *aProvider,
   long reason, void* info)
{
   CNotebookUpdate data;    /* The update value record */
   Str255          str;
   Boolean         saveIgnore = ignore;

   if (ignore)              /* Don't be a chatterbox */
      return;
   ignore = TRUE;

   TRY
   {
      if (FALSE) {}
      else if (reason == tableSelectionChanged
           && aProvider == fNotebook_FontList)
      {
```

```
    RgnHandle rgn = ((CTable*) aProvider)->GetSelection();

    data.selection = topLeft((**rgn).rgnBBox);
    UpdateData(&data, kNotebook_FontListID);
}
else if (reason == tableSelectionChanged
      && aProvider == fNotebook_SizeList)
{
    RgnHandlergn = ((CTable*) aProvider)->GetSelection();

    data.selection = topLeft((**rgn).rgnBBox);
    UpdateData(&data, kNotebook_SizeListID);
}
else if (reason == controlValueChanged
      && aProvider == fNotebook_BoldCheck)
{
    data.value = *(short*) info;
    UpdateData(&data, kNotebook_BoldCheckID);
}
else if (reason == controlValueChanged
      && aProvider == fNotebook_ItalicCheck)
{
    data.value = *(short*) info;
    UpdateData(&data, kNotebook_ItalicCheckID);
}
else if (reason == controlValueChanged
      && aProvider == fNotebook_UnderlineCheck)
{
    data.value = *(short*) info;
    UpdateData(&data, kNotebook_UnderlineCheckID);
}
else if (reason == controlValueChanged
      && aProvider == fNotebook_OutlineCheck)
{
    data.value = *(short*) info;
    UpdateData(&data, kNotebook_OutlineCheckID);
}
else if (reason == controlValueChanged
      && aProvider == fNotebook_ShadowCheck)
{
    data.value = *(short*) info;
    UpdateData(&data, kNotebook_ShadowCheckID);
}
else if (reason == controlValueChanged
      && aProvider == fNotebook_CondenseCheck)
{
    data.value = *(short*) info;
    UpdateData(&data, kNotebook_CondenseCheckID);
}
else if (reason == controlValueChanged
      && aProvider == fNotebook_ExtendCheck)
{
    data.value = *(short*) info;
    UpdateData(&data, kNotebook_ExtendCheckID);
}
else if (reason == controlValueChanged
      && aProvider == fNotebook_LeftRadio)
{
    data.value = *(short*) info;
    UpdateData(&data, kNotebook_LeftRadioID);
}
else if (reason == controlValueChanged
      && aProvider == fNotebook_JustCenterRadio)
{
    data.value = *(short*) info;
    UpdateData(&data, kNotebook_JustCenterRadioID);
}
```

```
      else if (reason == controlValueChanged
           && aProvider == fNotebook_JustRightRadio)
      {
         data.value = *(short*) info;
         UpdateData(&data, kNotebook_JustRightRadioID);
      }
      else if (reason == dialogTextChanged
           && aProvider == fNotebook_FontName)
      {
         ((CDialogText*) aProvider)->GetTextString(data.stringvalue);
         UpdateData(&data, kNotebook_FontNameID);
      }
      else if (reason == dialogTextChanged
           && aProvider == fNotebook_FontSize)
      {
         ((CDialogText*) aProvider)->GetTextString(data.stringvalue);
         UpdateData(&data, kNotebook_FontSizeID);
      }
      else if (reason == dialogTextChanged
           && aProvider == fNotebook_FontSample)
      {
         ((CDialogText*) aProvider)->GetTextString(data.stringvalue);
         UpdateData(&data, kNotebook_FontSampleID);
      }
      else
         CDialogDirector::ProviderChanged(aProvider, reason, info);
   }
   CATCH
      ignore = saveIgnore;
   ENDTRY

   ignore = saveIgnore;  /* ProviderChanged() can't Close()! */
}
```

The foregoing code consists of many tests of the values for the `reason` and the `aProvider` arguments to the call. These identify which object was changed (`aProvider`) and what kind of change was made (`reason`). When a match is found for the `reason` and `aProvider` values, then the appropriate field in the CNotebookUpdate structure (`data`) is modified to reflect the change and the UpdateData function is called with a pointer to the `data` and the VA-assigned identifier of the control whose state was changed.

CNotebookUpdate Structure Definition

The CNotebookUpdate structure contains one of each type of field associated with the various subpanes in the dialog. In the case of the Notebook dialog, the VA-generated CNotebookUpdate structure is defined as follows:

```
typedef struct
{
   Str255     stringvalue;  // CDialogText
   long       longvalue;    // CIntegerText
   short      value;        // All other controls and buttons
```

```
   Point       selection;    // CArrayPane
} CNotebookUpdate;
```

The comments associated with the fields in the foregoing definition identify the their contents. For example, if the user clicks one of the style checkboxes, the `value` field will be changed to contain the state of the checkbox. When the UpdateData function is called with a pointer to the (`data`) instance of this structure and the identifier of the control to which it pertains (`aProvider`), then it can use this datum to make changes to the current text settings for the dialog.

UpdateData Function Code

The UpdateData function in the VA-generated code for the x_CNotebook class is empty, as is the code for that same function in the CNotebook class; however I have added custom code to react to the calls to UpdateData in the CNotebook class version of this function and the new custom code is as follows:

```
void CNotebook::UpdateData(CNotebookUpdate *update, long itemNo)

{
   StringPtr pString;

   // UpdateData is called every time the user or the program
   // changes the value of a dialog pane. (Changes you make to
   // panes during a call to UpdateData do not result in
   // recursive calls to UpdateData.)

   // Override to dynamically update other program objects.
   // Note that *only* the value corresponding to itemNo is
   // present in the update record. If you need the values of
   // other panes, you must ask the panes for them.

   switch (itemNo)
   {
      case kNotebook_FontListID:
      {
         //
         // the user selected a font from the font list,
         // so we change the font name in the text box
         // and redisplay the sample.
         //
         long index = update->selection.v + 1;
         itsFontArray->GetArrayItem (&pString, index);
         fNotebook_FontName->SetTextString (pString);
         DrawSample();
         break;
      }
      case kNotebook_SizeListID:
      {
         //
         // the user selected a size from the size list,
         // so we change the size string in the text box
```

```
            // and redisplay the sample.
            //
            long index = update->selection.v + 1;
            itsSizeArray->GetArrayItem (&pString, index);
            fNotebook_FontSize->SetTextString (pString);
            DrawSample();
            break;
        }

        //
        // the following cases handle changes to the font style
        // checkboxes
        //
        case kNotebook_BoldCheckID:
        {
            fNotebook_FontSample->SetFontStyle (bold);
            break;
        }
        case kNotebook_ItalicCheckID:
        {
            fNotebook_FontSample->SetFontStyle (italic);
            break;
        }
        case kNotebook_UnderlineCheckID:
        {
            fNotebook_FontSample->SetFontStyle (underline);
            break;
        }
        case kNotebook_OutlineCheckID:
        {
            fNotebook_FontSample->SetFontStyle (outline);
            break;
        }
        case kNotebook_ShadowCheckID:
        {
            fNotebook_FontSample->SetFontStyle (shadow);
            break;
        }
        case kNotebook_CondenseCheckID:
        {
            fNotebook_FontSample->SetFontStyle (condense);
            break;
        }
        case kNotebook_ExtendCheckID:
        {
            fNotebook_FontSample->SetFontStyle (extend);
            break;
        }

        //
        // the following cases handle font justification changes
        //
        case kNotebook_LeftRadioID:
        {
            if (update->value)
            {
                fNotebook_FontSample->SetAlignment (teFlushLeft);
            }
            break;
        }
        case kNotebook_JustCenterRadioID:
        {
            if (update->value)
            {
                fNotebook_FontSample->SetAlignment (teCenter);
            }
            break;
```

```
        }
        case kNotebook_JustRightRadioID:
        {
           if (update->value)
           {
              fNotebook_FontSample->SetAlignment (teFlushRight);
           }
           break;
        }
     }
}
```

The foregoing code tests each of the checkbox, radio button, and list selections to determine whether the associated control's state was changed. If so, then the appropriate change is made to the `fNotebook_FontSample` field. This results in the Sample text field being redrawn with the new font, size, style, or justification setting. In the case where a new font name or size is chosen from the respective list, the foregoing code changes the contents of the text field below the list to correspond with the string value of the newly selected item. In addition, the DrawSample function is called to redraw the contents of the Sample field.

By placing code in the UpdateData function, you can react immediately to any changes the user might make. Doing so is not always necessary, but in this case, we want the Sample field to always reflect the current settings. If the settings in the dialog are not needed until it is dismissed, then you don't have to put any code into the UpdateData function.

Examining the Code to Dismiss the Notebook Dialog

When the user clicks either the OK or Cancel buttons in the dialog, then the DoCommand function of the CDialogDirector class winds up handling the `cmdOK` or `cmdCancel` commands. No matter whether the OK or Cancel button was clicked, the DoCommand function of the CDialogDirector calls the EndDialog function with the command code and a Boolean value that indicates whether the Validate function is to be called before the dialog is dismissed. The `fValidate` argument to the EndDialog function is `TRUE` in the case where the OK button was clicked and is `FALSE` in the case where the Cancel button was clicked.

EndDialog Function Code

The EndDialog function is quite straightforward. Its purpose is to cause the final contents of the dialog to be validated if the OK

button was clicked and make a decision whether to close the dialog or leave it open. The code is as follows:

```
Boolean CDialogDirector::EndDialog(long withCmd,
   Boolean fValidate)
{
   Boolean closeIt = TRUE;

   if (itsWindow)
   {
      if (fValidate)
         closeIt = Validate();

      if (closeIt)
         dismissCmd = withCmd;
   }
   return closeIt;
}
```

The foregoing code shows that the `closeIt` variable is set to `TRUE` at the beginning of the function and then if the window still exists (which it should), then the `fValidate` argument is tested to determine whether to call the Validate function or not. If Validate is called, its return value will change the state of the `closeIt` variable. If the `closeIt` variable is `TRUE`, then the value of the `dismissCmd` is changed to the value of the `withCmd` argument, which will cause the event loop in the DoChangeableDialog function to be exited (see pages 196–197). If the result of the Validate function is `FALSE`, then that function will display a dialog indicating the reason for the failure in validating the contents of the dialog. When the user dismisses that dialog, the event loop in the DoChangeableDialog function will continue running.

The default action of the Validate function is to determine whether any of the text fields in the dialog were specified as "required" fields. This ensures that such text fields are never left empty when the dialog is dismissed. In order to take advantage of this automatic behavior, you need to check the `isRequired` checkbox in the CDialogText settings for the field when the field's characteristics are described in the VA. Of course, you can override the Validate function in your dialog director (which, in our case, would be the CNotebook class) and perform any validation procedures you like. Validate returns a `TRUE` or `FALSE` value, and that result determines whether the dialog is closed or not.

DoEndData Function Code

If the dialog was dismissed with the Cancel button, or if the OK button was clicked and the Validate function returned a `TRUE` result, then the dialog window will be closed and the DoModalDialog function in the x_CNotebook class will eventually receive control. When it does, it calls the DoEndData function with the command code that was used to dismiss the dialog. The code in the VA-generated DoEndData function is as follows:

```
void x_CNotebook::DoEndData(long theCommand)
{
   CNotebookData    data;       /* The initial value record */
   RgnHandle        rgn;        /* Selection region */

                                /* If user canceled the dialog,*/
   if (theCommand == cmdCancel)/* return the initial values */
   {
      data = saveData;
   }
   else
      CollectPaneValues(data);// Get current pane values

   EndData(&data);             // Tell the derived class

   saveData = data;            // Now has current values
}
```

The foregoing code is the first step in the process of retrieving the final settings in the dialog and returning these to the document (or other caller) for its use. If the value of the `theCommand` argument is `cmdCancel`, then the temporary `data` structure is loaded with the saved data, from when the dialog was first entered, from the `saveData` variable. This ensures that the settings returned to the document (or other caller) reflect the original settings if the user cancels the dialog. If the dialog is dismissed with the OK button, then the foregoing code calls the CollectPaneValues function to access each of the controls and fields in the dialog and retrieve its current (final) value.

EndData Function Code

The value stored into the `data` structure is passed to the EndData function. As stated previously, the `data` structure (an instance of the CNotebookData structure) may contain either the original settings (in case the dialog was dismissed by clicking the Cancel button) or the final values (if the dialog was dismissed with the OK button). The code for the EndData function is as follows:

```
void CNotebook::EndData(CNotebookData *final)
{
   // The values of all panes are returned by this function,
   // which is called just before Close for a modeless dialog,
   // or just before returning from DoModalDialog.

   // If DoModalDialog returns cmdCancel, EndData is called
   // with the values initially supplied to BeginData, allowing
   // you to back out any intermediate changes made in response
   // to UpdateData. If you do not use UpdateData, you can
   // test the value of dismissCmd to see whether to respond
   // to EndData.

   //
   // transfer the pane values into the itsSettings structure, in
   // preparation for the transferral of that structure by the
   // ExchangeSettings function.
   //
   itsSettings.nFontStyle = normal;
   if (final->fNotebook_BoldCheck)
      itsSettings.nFontStyle |= bold;
   if (final->fNotebook_ItalicCheck)
      itsSettings.nFontStyle |= italic;
   if (final->fNotebook_UnderlineCheck)
      itsSettings.nFontStyle |= underline;
   if (final->fNotebook_OutlineCheck)
      itsSettings.nFontStyle |= outline;
   if (final->fNotebook_ShadowCheck)
      itsSettings.nFontStyle |= shadow;
   if (final->fNotebook_CondenseCheck)
      itsSettings.nFontStyle |= condense;
   if (final->fNotebook_ExtendCheck)
      itsSettings.nFontStyle |= extend;

   if (final->fNotebook_LeftRadio)
      itsSettings.nFontJust = teFlushLeft;
   if (final->fNotebook_JustCenterRadio)
      itsSettings.nFontJust = teCenter;
   if (final->fNotebook_JustRightRadio)
      itsSettings.nFontJust = teFlushRight;


   if (final->fNotebook_FontName[0] > 0)
      TCLpstrcpy (itsSettings.spFontName, final->fNotebook_FontName);
   else
      TCLpstrcpy (itsSettings.spFontName, "\pChicago");
   if (final->fNotebook_FontSize[0] > 0)
      TCLpstrcpy (itsSettings.spFontSize, final->fNotebook_FontSize);
   else
      TCLpstrcpy (itsSettings.spFontSize, "\p12");

   //
   // call the document's ExchangeSettings function to store the
   // new settings into the document's copy only if the dialog
   // was dismissed with the OK button.
   //

   if (dismissCmd == cmdOK)
   {
      ((CMain *)itsSupervisor)->ExchangeSettings(itsSettings, TRUE);
   }

   //
   // free all of the font name strings allocated by the
   // BeginData function. The itsFontArray and itsSizeArray will
```

```
    // be disposed when the corresponding lists are disposed.
    //
    StringPtr pFontName;
    CArrayIterator iter (itsFontArray, kStartAtBeginning);
    while (iter.Next (&pFontName))
    {
       free (pFontName);
    }
}
```

Our version of the EndData function in the CNotebook class (shown in the foregoing code) takes the information from the data structure (called `final`) and condenses it into the abbreviated structure that we defined in the `itsSettings` variable. This variable (also a structure) is an instance of the CTextSettings structure shown on page 180. It is the value of the `itsSettings` variable that we pass back to the document as the result of the dialog's execution.

Toward the end of the foregoing code, the ExchangeSettings function is called with a reference to the `itsSettings` structure and also a `TRUE` second argument, which tells the ExchangeSettings function that the data are to be transferred from the dialog to the document.

After the exchange of data is complete, the EndData function cleans up by disposing of the font name strings that were allocated when the dialog was first invoked. The contents of the two array objects will be disposed automatically when their corresponding lists are disposed (this is because the `fOwnership` argument of the SetArray function is specified to be `TRUE`, indicating that the list owned the array once it was made known to the list). If you want to manage the storage for the array independently of the list, then you must specify `FALSE` for the `fOwnership` argument in the SetArray function call (see page 190 for an example of the SetArray call for the font name list in the BeginData function).

Update Function Code

After the foregoing steps are complete, control returns to the DoCmdNotebook function, whose code is shown beginning on page 182. When DoModalDialog returns and it is determined that the return value is `cmdOK` (indicating that the OK button was clicked to dismiss the dialog), then the DoCmdNotebook function calls the UpdateWindowText function to modify the ap-

pearance of the text within the Main view to reflect the new dialog settings. The UpdateWindowText code is as follows:

```
void CMain::UpdateWindowText ()
{
   short nFontNum, nFontSize;

   GetFNum (settings.spFontName, &nFontNum);
   nFontSize = atoi ((const char
*)&settings.spFontSize[1]);
   fMain_TextPane->SetFontNumber (nFontNum);
   fMain_TextPane->SetFontSize (nFontSize);
   fMain_TextPane->SetFontStyle (normal);
   fMain_TextPane->SetFontStyle (settings.nFontStyle);
   fMain_TextPane->SetAlignment (settings.nFontJust);
}
```

The foregoing code uses the values in the `settings` structure to set the font name, size, style, and alignment for the text in the Main view.

Creating a Category Editor Dialog

Let's assume that you have continued to develop the Business View that was described in the previous chapter. If so, you may have come to the point where you wish to construct a list of categories that you can use to verify the user's transaction entries.

This task cries out for a modeless dialog in which the current list of categories can be displayed alongside the normal account window. The dialog will also provide the user with the means to create new categories, edit existing categories, or delete categories.

In addition to showing how to create and make use of modeless dialogs, this example illustrates the similarities in the VA-generated code for modeless and modal dialogs. The example also constructs a modal dialog for creating or editing category names.

Creating the Dialog Views

I have created a new VA project to illustrate the features of the Categories view and its associated NewCat view. I have not altered the Main view that the VA creates by default, but have added new views, as shown in Figure 5-9.

The first new view constructed was the NewCat view. I knew that when the user wanted to enter a new category that the means of

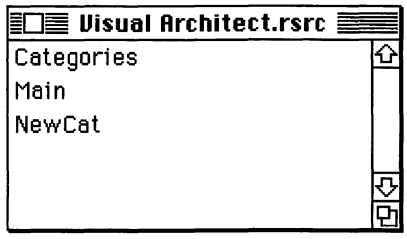

Figure 5-9
Views constructed for Categories project

doing so should be simple and accessible from the main Categories view, but after the new category was entered, the screen space taken up by that process shouldn't add to the size of the Categories view. With these requirements in mind, I designed a separate, modal dialog, that could be opened from the Categories view. By creating the modal dialog first, we can refer to it when constructing the Categories view. Choose **New View** from the VA's **View** menu to create the view. Enter the name "NewCat," and choose Dialog as the view type in the dialog that the VA displays.

The NewCat view is shown in its completed form in Figure 5-10. It is a simple view, with two Edit Text fields (the category name is a required entry), a "Tax-Related" checkbox, and standard OK and Cancel buttons. The "View Info" for the NewCat view is shown in Figure 5-11. Note that the "modal" checkbox is checked for the view and that the standard bordered dialog (third from the right) is selected as the window type.

The Pane Info specifications for the category name (NewCatName) field are shown in Figure 5-12. Note that the `isRequired` checkbox is checked to ensure that the user enters a nonempty string in that field. The Pane Info specifications for the description (NewCatDescrip) field are shown in Figure 5-13.

The "Expense" and "Income" radio buttons are unremarkable and need no additional explanation, except that I changed the setting for the `cntrlValue` attribute of the CControl class for the Expense radio button to 1 so that it would be active initially.

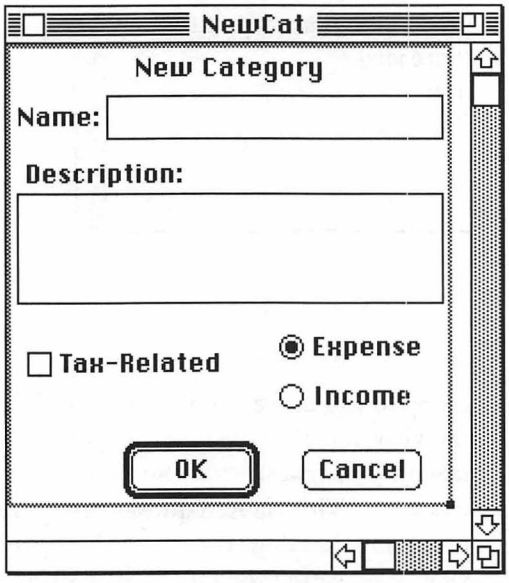

Figure 5-10
NewCat view fully constructed

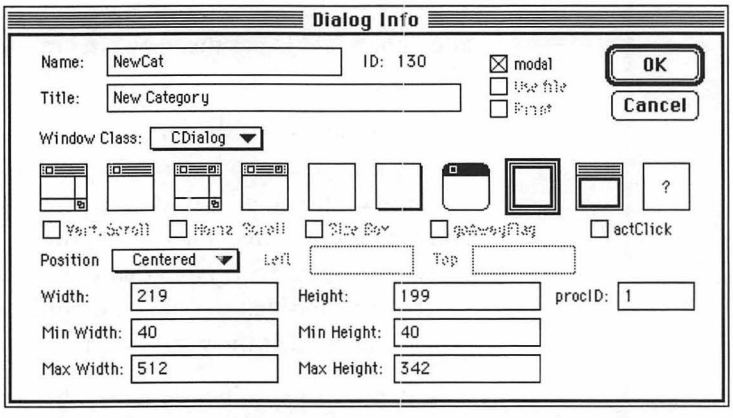

Figure 5-11
View Info for NewCat modal dialog

The Pane Info specifications for the Tax-Related (NewTaxable) checkbox are shown in Figure 5-14.

After the NewCat view has been constructed, we can turn our attention to the Categories view. Close the NewCat view, and then choose **New View** from the **View** menu. Enter "Categories" as the name, and select Dialog as the view type in the dialog that the VA displays. The completed appearance of the Categories view is shown in Figure 5-15. I have tried to minimize the screen real estate taken up by this view, so the buttons below the list of catego-

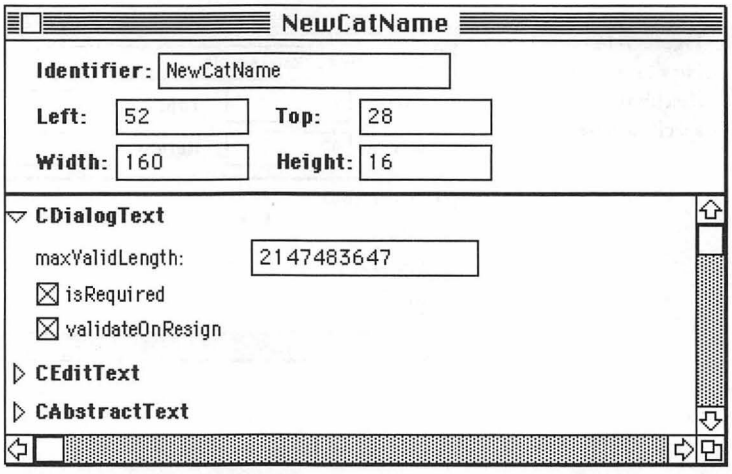

Figure 5-12
NewCatName field specifications

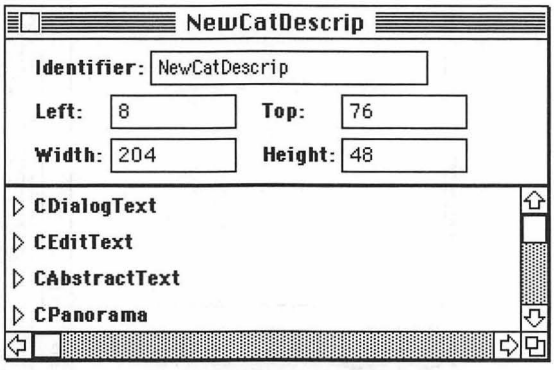

Figure 5-13
NewCatDescrip field specifications

ries are smaller than the normal size. This is a reasonable deviation from the user interface guidelines. After constructing the "Use" button and assigning it a command, I chose **Set Default Command** from the **View** menu and specified the same command that was assigned to the "Use" button. This allows the user to select a category in the list and then press the return key to cause that category to be "used" for the current transaction. The intention is for the Categories view to communicate the user's selections to the CMain document class, as transactions are being entered and categories are chosen. Also, the Use, Edit, and Delete buttons will be

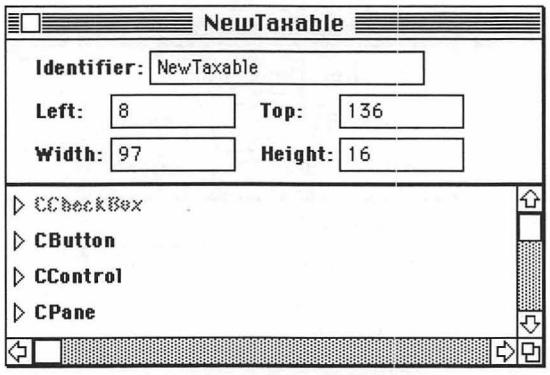

Figure 5-14
NewTaxable checkbox specifications

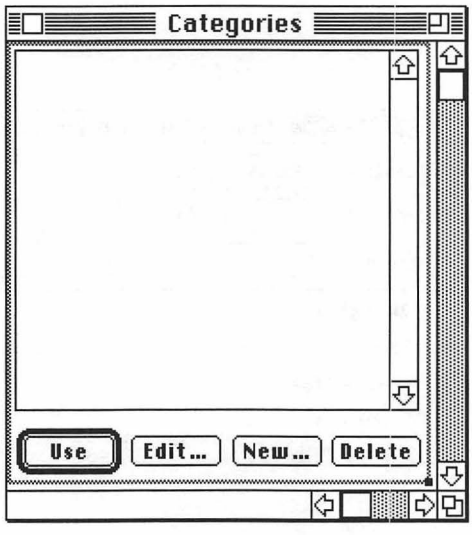

Figure 5-15
Categories view fully constructed

disabled unless a category is selected in the view's list. We can disable the buttons in the initialization function for the view and then enable them when the ProviderChanged function receives notification that a list entry has been selected. The New button should always be enabled, because we want the user to be able to enter new categories at any time.

The Pane Info specifications for the category list (CatTable) are shown in Figure 5-16. Note that the `selOnlyOne` checkbox is

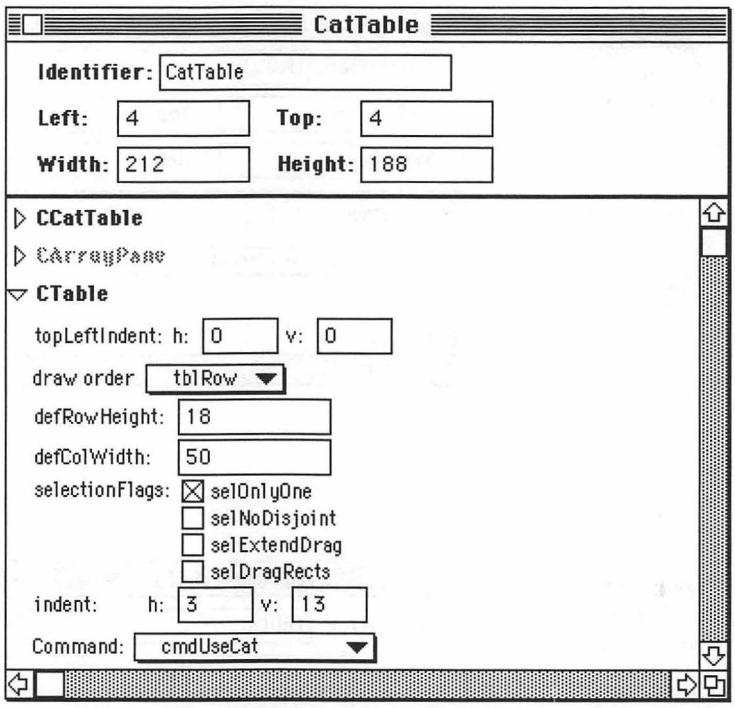

Figure 5-16
CatTitle list specifications

checked and that the `cmdUseCat` command was assigned to the list. The command is issued when the user double-clicks on an entry in the list. You will notice that our CatTable list is created from the CCatTable object, which is a new class that was created in the VA. It is derived from the CArrayPane class so that we can supply the contents for the table, when called to do so, in the GetCellText override function (see page 237).

The Pane Info specifications for the Use button are shown in Figure 5-17. Note that the `cmdUseCat` command has been assigned to this button as well, allowing the user to select a category and click the button or press the Return key to cause that category to be used in the current transaction.

The Pane Info specifications for the Edit button are shown in Figure 5-18. The `cmdEditCat` command was assigned to this button.

The Pane Info specifications for the New button are shown in Figure 5-19. The `cmdNewCat` command was assigned to this button.

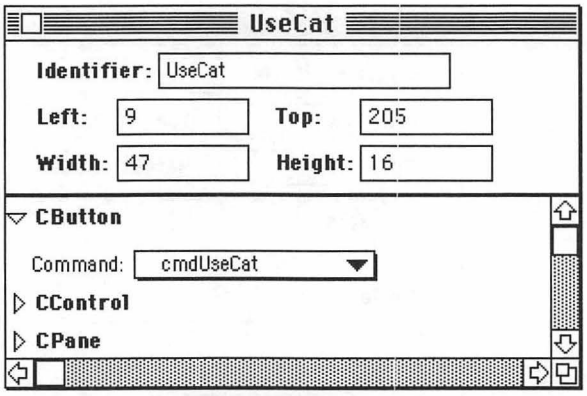

Figure 5-17
Use button specifications

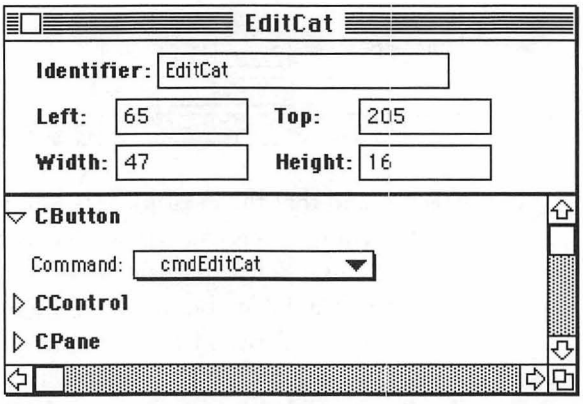

Figure 5-18
Edit button specifications

The Pane Info specifications for the Delete button are shown in Figure 5-20. The `cmdDeleteCat` command was assigned to this button.

The command behavior for the various buttons is different. The Use button is specified to "Call" a function in the CCategories class when the command is issued. Both the Edit and New buttons are specified to "Open" the CNewCat view. The Delete button is specified to "Call" a function in the CCategories class when the command is issued.

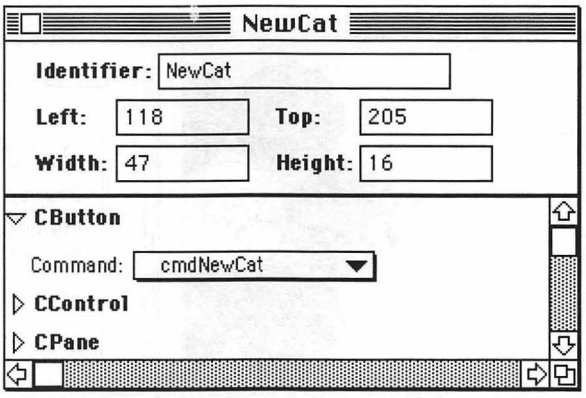

Figure 5-19
New button specifications

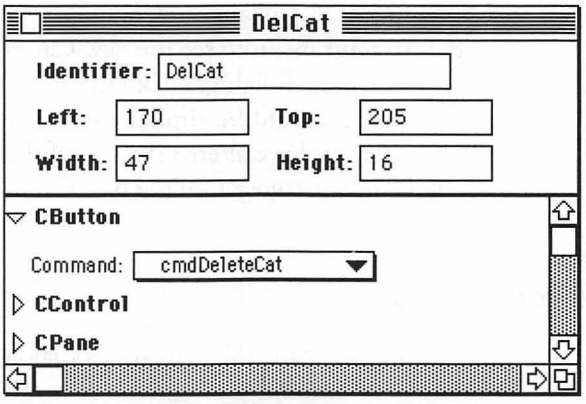

Figure 5-20
Delete button specifications

Finally, in addition to the views, an **Account** menu was constructed and a single menu item (command) was added to it. The command entry is named **Categories** and the command assigned to that menu item is `cmdEditCategories`, for which the behavior to "Open" the CCategories view was assigned.

The foregoing creates the user interface framework for what we want to accomplish. At this point, choose the **Generate All** command from the VA's project menu, and then compile and execute

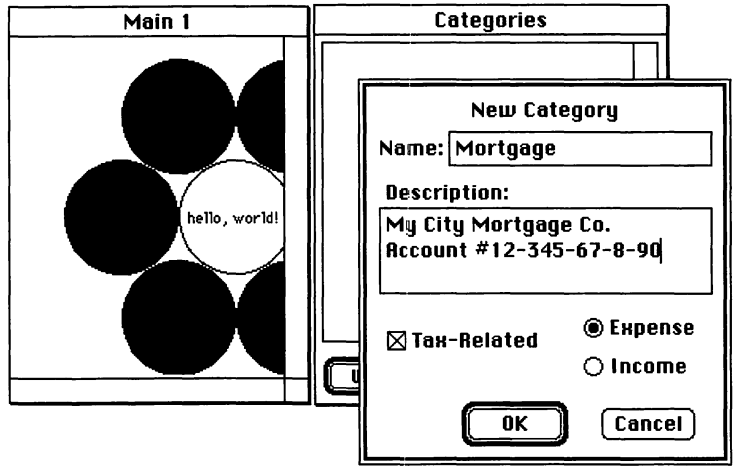

Figure 5-21
Categories application in execution, with all views shown

the application. After the application commences execution, the Main window will appear. Choose the **Categories** command from the **Account** menu to see the new Categories view. To also see the NewCat modal dialog, click either of the New or Edit buttons. The result should be similar to what is shown in Figure 5-21. Note that we have altered the size of the Main view and have entered a new category into the NewCat modal dialog.

Examining the CMain Code

In order to "run" the Categories dialog, a number of functions were built into the CMain class to handle interaction with the dialog. Because the Categories dialog is modeless, it will remain on the screen until the user dismisses it explicitly. Therefore, when the user adds a new category, modifies an existing category, deletes a category, or even "uses" an existing category, the document needs to be notified of this fact. The list of categories is kept in the document, rather than in a member variable of the CCategories class, mainly because the document survives the closure of the dialog. That is, the user may elect to define some categories in the dialog and then close the dialog window to make its screen space available for something else. In this scenario, you certainly wouldn't want the newly defined categories to be discarded.

A new CArray member variable was added to the CMain class to hold all of the category information. Because there is more than just the category name to consider, a new class called CCat was created, which defines member variables and access functions for each of a category's related data items (name, information, type [that is, expense or income], and tax-related status). The array in **CMain.h** holds a CCat object for each category that is defined.

CMain.h Header File Contents

The contents of the **CMain.h** header file includes prototypes for new member functions as well as the CArray and is as follows:

```
/************************************************************
 CMain.h

        Header File For CMain Document Class

   Copyright © 1995 Richard O. Parker. All rights reserved.

   Generated by Visual Architect™ 12:02 PM Wed, Dec 14, 1994

************************************************************/

#pragma once
#include "x_CMain.h"
class CArray;
class CCat;

class CMain : public x_CMain
{
public:

   TCL_DECLARE_CLASS

   CArray  *categories;     // list of category names
   void    ICMain(void);

   virtual void MakeNewContents(void);
   virtual void ContentsToWindow(void);
   virtual void WindowToContents(void);

   virtual void AddCategory (CCat *aCat);  // add a category
   virtual void DelCategory (long index);  // delete a category
   virtual CCat* GetCategory (long index);  // find a category
   virtual void SetCategory (CCat *aCat,   // store category
                  long index);
   virtual void SetSelectedCategory (long index);// handle "use"
   virtual void DoCmdEditCategories (void);   // override

private:

   virtual void SortCat ();              // sort category  list
};


#define CMainFType 'TEXT'
```

The newly added portions of the foregoing header file are annotated with comments. The `categories` variable is a pointer to a CArray object that we will create when the document is initialized. The newly added functions that interface with the CCategories class include AddCategory, DelCategory, GetCategory, SetCategory, SetSelectedCategory, and the "private" SortCat function. The DoCmdEditCategories function is an override of the function of that same name that the VA generated into our x_CMain class to handle the **Categories** command from the **Account** menu. The SortCat function was defined to be private because it is used only by the category-oriented member functions that were added to the class.

ICMain Function Code

A single statement was added to the ICMain function in order to create the `categories` array when the document is initialized. The code is as follows:

```
void CMain::ICMain()
{
   Ix_CMain();

   // create the categories list
   categories = TCL_NEW (CArray, (sizeof (CCat *)));
}
```

The foregoing code illustrates that the base class function is called first, and then the `categories` array is created. It also indicates that the size of each element of the array is what is required to hold a pointer to a CCat object.

The constructor of the CArray object also takes an optional second argument that specifies the number of slots that are allocated each time the array is expanded. By default, the value of this argument is 3. You can provide your own value for this argument if you wish, but the default value will minimize the amount of storage taken up by unused members of the array.

AddCategory Function Code

The AddCategory function is called by the command handler for the "Add" button in the Categories dialog to add a new category to the document's `categories` array. The code for this function is as follows:

```
void CMain::AddCategory (CCat *aCat)
{
   //
   // the easiest way to add a category is to add it
   // to the end of the array and then sort the array.
   //
   categories->Add (&aCat);
   SortCat();
}
```

The comments in the foregoing code indicate the methodology used for adding new categories; the list of categories is kept alphabetized at all times. You'll also note that no attempt is made to ensure that duplicate category names aren't added. Code to perform that function could be added later.

DelCategory Function Code

The DelCategory function is called to delete a category from the `categories` array. The code is as follows:

```
void CMain::DelCategory (long index)
{
   long num = categories->GetNumItems();

   if (index <= num && index >= 1)
   {
      categories->DeleteItem (index);
   }
}
```

The foregoing code takes an array index as its argument, validates the index, and then uses it to call the DeleteItem function of the CArray class.

GetCategory Function Code

The GetCategory function provides the means for a category to be accessed from the `categories` array. The code is as follows:

```
CCat* CMain::GetCategory (long index)
{
   CCat *aCat;
   long num;

   num = categories->GetNumItems();
   if (index > num || index < 1)
   {
      return NULL;
   }
   categories->GetArrayItem (&aCat, index);
```

```
    return aCat;
}
```

The foregoing code takes an array index as its argument, validates the index, and then uses it in the call to the GetArrayItem function of the CArray class. The return value from the function is a pointer to a CCat object (which contains all of the information pertinent to a category entry).

SetCategory Function Code

The SetCategory function provides the means to store a category back into the `categories` array. The code is as follows:

```
void CMain::SetCategory (CCat *aCat, long index)
{
   CCat *theCat = GetCategory (index);
   if (theCat == NULL)
   {
      ASSERT (!"Invalid category index");
   }
   DelCategory (index);
   AddCategory (aCat);
}
```

The foregoing code takes a pointer to a CCat object and the original index of that object in the `categories` array. The purpose of the function is to replace an existing entry with a new version of that same entry (when the user "edits" the category). The function performs its task by calling the GetCategory function to retrieve the original category entry using the supplied index (simply to verify that the category entry exists), then deletes the existing category, and, finally, adds the new CCat object to the array by calling the AddCategory function.

SortCat Function Code

The SortCat function performs a simple N^2 algorithm to sort the elements of the `categories` array into alphabetical sequence according to the category name. The code is as follows:

```
void CMain::SortCat ()
{
   CCat *pCat, *nCat;
   Str255 pName, nName;

   CArrayIterator pIter (categories, kStartAtBeginning);
   CArrayIterator nIter (categories, 0);
```

```
   //
   // perform a simple N^2 sort of the categories array
   //
   while (pIter.Next (&pCat))
   {
      pCat->GetCatName(pName);
      nIter.MoveTo (pIter.GetCursor());
      while (nIter.Next (&nCat))
      {
         nCat->GetCatName(nName);
         if (IUCompString(nName, pName) < 0)
         {
            categories->Swap (pIter.GetCursor(), nIter.GetCursor());
         }
      }
   }
}
```

The foregoing code is very straightforward, but it uses two CArrayIterator objects to iterate through the array. The outer loop commences with the `pIter` iterator initialized to the beginning of the array. Following its initialization, the `pIter` iterator is used to access the next (first) CCat object and then call its GetCatName access function to get the object's category name. The inner loop commences with the `nIter` iterator initialized to the position immediately following the `pIter` iterator. The inner loop accesses the next CCat object in the list and uses its pointer to call the GetCatName function for the object to get the object's category name. The IUCompString function (a toolbox routine) is used to compare the two category names. If the object addressed by the `nIter` iterator has a name that should precede the object addressed by the `pIter` iterator, then the two entries are swapped by calling the Swap function of the CArray class. Notice that, in this case, we are merely swapping object pointers, so the sort algorithm is fairly efficient—certainly enough for our purpose. We will cover more about collections and iterators in a later chapter.

SetSelected Category Function Code

The code for the SetSelected category function is called when the user selects a category from the list in the Categories dialog and then clicks the "Use" button. The skeleton code for this function is as follows:

```
void CMain::SetSelectedCategory (long index)
{
   // this function is empty, but it should be revised to store
   // the relevant information from the specified category index
```

```
   // into the currently active account entry.
}
```

As is apparent in the foregoing, I have not implemented any specific behavior when the function is called. I presume that if users are editing account transactions they will store the selected category into the current transaction. Because the main purpose of this example application is to show the functionality of the modeless Categories dialog, I didn't feel justified in obscuring that with a lot of other code.

DoCmdEditCategories Function Code

When the **Account** menu was created in the VA, a single menu command called **Categories** was also added. The code specified that the behavior when choosing the **Categories** command was to respond to the `cmdEditCategories` command by calling a function that the VA has named DoCmdEditCategories. The original code for this function is generated into the x_CMain class, automatically, and you saw the result of choosing the function in Figure 5-21. The Categories dialog window is opened when the function is executed. However, because we also wish to disable several of the buttons when the dialog is first opened, the base class code is overridden, as follows:

```
void CMain::DoCmdEditCategories()
{
   CCategories   *dialog;

   dialog = TCL_NEW(CCategories,());
   dialog->ICCategories(this);
   dialog->BeginDialog();
   dialog->DisableButtons();
}
```

The foregoing code creates a CCategories object, whose pointer is stored into the `dialog` variable. The ICCategories initialization function is called to create the dialog's window and controls, the BeginDialog function is called to display the dialog, and then the DisableButtons function is called to ensure that the Use, Edit, and Delete buttons are inactive when the dialog is first opened.

Notice that, unlike a modal dialog where control doesn't return to the caller until the user dismisses the dialog, the BeginDialog function returns control after activating the dialog, so that we can

continue to execute and process normal events. Of course, the pointer to the CCategories object is lost when the DoCmdEdit-Categories function returns, but we don't need to communicate with the dialog. It will communicate with the CMain object by storing the CMain object pointer (`this`) as the value of the `itsSupervisor` variable. If you need to retain a pointer to the dialog for any reason, then you could set aside a member variable in the CMain class to hold it.

Examining the CCategories Code

The CCategories class holds all of the code that is relevant to the operation of the Categories dialog. The dialog is constructed from the code generated into the x_CCategories base class source file and the custom code to support the behavior of the dialog is vested in the additions we have made to the CCategories derived class. The following sections discuss the relevant portions of both the base class and its derived class.

One thing you'll notice in the case of a modeless dialog is that the code is structured very much like that of the modal dialog. For example, when the dialog is first invoked, the BeginData function is called to initialize the contents of the various subpanes of the dialog (that is, its controls). Also, when the dialog is closed (in this instance, the user must "close" the dialog, rather than dismiss it) the EndData function is called to allow the current (final) settings to be saved. Also, as for the modal dialog example, the various subpanes of the dialog window, as well as the window itself, are constructed in the MakeNewWindow function, which is called from the object's initialization code.

When a modeless dialog is running, its events are posted in the normal event queue and are handled just the same as other events. So, in the respect that a modeless dialog allows the user to choose menu commands and interact with windows other than the dialog when it is open, a modeless dialog is not much different than a normal window and is handled in much the same way by the TCL.

The x_CCategories Class Header File

The intrinsic features of the Categories dialog are provided by member variables in the **x_CCategories.h** header file, and the data exchanged while the dialog is running are held in structures defined in that file as well. The contents of the file are as follows:

```
/************************************************************
 x_CCategories.h

        Header File For CCategories Lower-Layer Dialog Class

   Copyright © 1995 Richard O. Parker. All rights reserved.

   Generated by Visual Architect™

   This file is rewritten each time you generate code. You should
   not make changes to this file; changes should go in the My.h
   file, instead.

   If you want to change how Visual Architect generates this
   file, you can change the template for this file. It is
   "_Dialog.h" in the Visual Architect Templates folder.

************************************************************/

#pragma once
#include "CDialogDirector.h"

class CCatTable;
class CButton;

   // Data struct for initializing dialog items
   // and receiving changed values

typedef struct
{
                              /* Array pane (table)*/
    Point fCategories_CatTable;
} CCategoriesData;

// We define a separate struct for UpdateData() which
// eliminates duplicate data types

typedef struct
{
   Str255  stringvalue;  // CDialogText
   long    longvalue;    // CIntegerText
   short   value;        // All other controls and buttons
   Point   selection;    // CArrayPane
} CCategoriesUpdate;

class CDirectorOwner;
class CPanorama;
class CPane;

class x_CCategories : public CDialogDirector
{
public:

   TCL_DECLARE_CLASS

   // Pointers to panes in window
   CCatTable  *fCategories_CatTable;
   CButton    *fCategories_UseCat;
   CButton    *fCategories_EditCat;
   CButton    *fCategories_NewCat;
   CButton    *fCategories_DelCat;

   void Ix_CCategories(CDirectorOwner *aSupervisor,
      Boolean push = FALSE);

   virtual long    DoModalDialog(long defaultCmd);
```

```
   virtual Boolean Close(Boolean quitting);
   virtual void    ProviderChanged(CCollaborator *aProvider,
      long reason, void* info);
   virtual void    DoCommand(long theCommand);
   virtual void    UpdateMenus(void);

protected:
   Boolean ignore;

   virtual void    MakeNewWindow(void);

   // Pull-style functions
   virtual void    BeginData(CCategoriesData *initial);
   virtual void    UpdateData(CCategoriesUpdate *update,
      long itemNo);
   virtual void    EndData(CCategoriesData *final);

   // Push-style functions
   virtual void    SetData(const CCategoriesData& initial);
   virtual void    GetData(CCategoriesData& final);

   virtual  void    DispensePaneValues(const CCategoriesData& data);
   virtual void    CollectPaneValues(CCategoriesData& data);

   virtual void    DoCmdUseCat(void);
   virtual void    DoCmdEditCat(void);
   virtual void    DoCmdDeleteCat(void);
   virtual void    DoCmdNewCat(void);

   CPane   *FindPane(long ID);

private:
   virtual void    DoBeginData(Boolean push);
   virtual void    DoEndData(long theCommand);
};

#define CVueCCategories  129
```

The main features of the foregoing declarations are the member variable declarations for the pointers to panes in the window and the declarations for functions to implement the behaviors for the controls. In addition, Ix_CCategories (initialization), MakeNewWindow, BeginData, and EndData are standard functions in any dialog, either modal or modeless.

Ix_CCategories Function Code

The Ix_CCategories function is called by the ICCategories function right after the CCategories object is first constructed in the DoCmdEditCategories function (see page 222). The code for the Ix_CCategories function is as follows:

```
void x_CCategories::Ix_CCategories(CDirectorOwner *aSupervisor,
   Boolean push)
{
   IDialogDirector(aSupervisor);

   // There are several circumstances where we don't want
```

```
   // ProviderChanged to be called. During initialization,
   // during calls to UpdateData, etc. The ignore flag
   // heads these off.

   ignore = TRUE;          /* Don't call UpdateData now  */
   MakeNewWindow();        /* Create the dialog's window */
   DoBeginData(push);      /* Gather initial values      */
   ignore = FALSE;
}
```

It is interesting to compare the foregoing code with what was generated by the VA for the Notebook modal dialog in its Ix_CNotebook function on page 184. You'll notice that the code is identical. This is because, at least in the beginning, both modal and modeless dialogs are initialized in the same fashion.

The rationale for setting the `ignore` variable to `TRUE`, creating the window, initializing the dialog's controls, and then setting the `ignore` variable to `FALSE` was discussed on page 185.

MakeNewWindow Function Code

The first main task of the Ix_CCategories function is to create the objects that define the controls and other panes in the dialog. The MakeNewWindow function accomplishes this task with the code that follows:

```
void x_CCategories::MakeNewWindow(void)
{
   itsWindow = TCLGetNamedWindow("\pCategories", this);

   // Initialize pointers to the subpanes in the window

  fCategories_CatTable = (CCatTable*) FindPane(kCategories_CatTableID);
  ASSERT(member(fCategories_CatTable, CCatTable));

  fCategories_UseCat = (CButton*) FindPane(kCategories_UseCatID);
  ASSERT(member(fCategories_UseCat, CButton));

  fCategories_EditCat = (CButton*) FindPane(kCategories_EditCatID);
  ASSERT(member(fCategories_EditCat, CButton));

  fCategories_NewCat = (CButton*) FindPane(kCategories_NewCatID);
  ASSERT(member(fCategories_NewCat, CButton));

  fCategories_DelCat = (CButton*) FindPane(kCategories_DelCatID);
  ASSERT(member(fCategories_DelCat, CButton));
}
```

The foregoing code calls the TCLGetNamedWindow function to access the 'CVue' resource for the specified window name and then makes calls to the FindPane function to locate each subpane with its associated identifier in the VA-constructed 'CVue' re-

source. This constructs the window and its subpanes, but they are not yet made visible to the user. The window and its contents are made visible only after the BeginDialog function is called in the DoCmdEditCategories function (see page 222).

Although the next action in the Ix_CCategories function is to call the DoBeginData function (which calls BeginData), we will defer an examination of the BeginData function for a moment, while we catch up with the code in the CCategories derived class.

The CCategories Class Header File

The header file for the CCategories class contains declarations that were generated by the VA as well as declarations that were added to customize the code. Our declarations contain comments that indicate that they have been newly added. The header file is as follows:

```
/************************************************************
 CCategories.h

        CCategories Dialog Director Class

   Copyright © 1995 Richard O. Parker. All rights reserved.

   Generated by Visual Architect™ 12:02 PM Wed, Dec 14, 1994

   This file is only generated once. You can modify it by filling
   in the placeholder functions and adding any new functions you
   wish.

   If you change the name of the dialog class, a fresh version
   of this file will be generated. If you have made any changes
   to the file with the old name, you will have to copy those
   changes to the new file by hand.

************************************************************/

#pragma once
#include "x_CCategories.h"


class CDirectorOwner;
class CCat;             // newly added

class CCategories : public x_CCategories
{
public:

   TCL_DECLARE_CLASS

   // Insert your own public data members here
   CCat    *settings;      // newly added

   void    ICCategories(CDirectorOwner *aSupervisor);
   virtual void    ProviderChanged(CCollaborator *aProvider,
```

```
      long reason, void* info);
   virtual void    DoCommand(long theCommand);

   virtual void    ExchangeSettings (CCat *&aCat,   // new
      Boolean from);
   virtual void    DisableButtons (void);           // new

protected:

   virtual void    BeginData(CCategoriesData *initial);
   virtual void    UpdateData(CCategoriesUpdate *update,
      long itemNo);
   virtual void    EndData(CCategoriesData *final);

   virtual void    DoCmdUseCat(void);    // override
   virtual void    DoCmdEditCat(void);   // override
   virtual void    DoCmdDeleteCat(void); // override
   virtual void    DoCmdNewCat(void);    // override
};
```

In the foregoing, I added the declaration of the `settings` variable, which is a pointer to a CCat object; I also added the ExchangeSettings and DisableButtons public member function declarations, as well as the DoCmdUseCat, DoCmdEditCat, DoCmdDeleteCat, and DoCmdNewCat protected function declarations to override what the VA generated for those functions in the x_CCategories class. All of the remaining declarations and content were generated by the VA.

ICCategories Function Code

The ICCategories function is called by the DoCmdEditCategories function of the CMain class when the user chooses the **Categories** command from the **Account** menu. The code for that function is shown beginning on page 222. The code for the ICCategories function is as follows:

```
void CCategories::ICCategories(CDirectorOwner *aSupervisor)
{
   // Initialize data members that must be set up before
   // BeginData is called here.

   x_CCategories::Ix_CCategories(aSupervisor, FALSE);
}
```

The foregoing code merely calls the Ix_CCategories function to perform the initialization for the CCategories object. That code was shown beginning on page 225.

BeginData Function Code

The BeginData function is called during the initialization process to specify the initial values for each of the dialog's panes. The newly modified version of this code is as follows:

```
void CCategories::BeginData(CCategoriesData *initial)
{
   // Base class calls BeginData once after the window is created
   // to gather the initial values for the dialog panes. Note
   // that BeginData is called *before* Ix_CCategories returns.
   // The initial struct is cleared to zeros.

   // Calling CollectPaneValues copies the initial values you set
   // in Visual Architect from the panes. This lets you
   // use these values as the starting point every time the
   // dialog is run. If you want to use values determined by your
   // program instead, omit this call.

   CollectPaneValues(*initial);

   // Specify the array associated with the category list. By
   // doing so, the table will be redrawn by calling its
   // GetCellText function. Note: we do _not_ own this array.
   // It's owned by the document.
   fCategories_CatTable->SetArray (((CMain *)itsSupervisor)
      ->categories, FALSE);
}
```

The only change made to the foregoing function was to assign the `categories` array (defined in the CMain class) as the array associated with the CCatTable (CArrayPane-derived) object in the dialog. As the comments indicate, we do not want the CCatTable object to own the array, because we do not want the array to be disposed when the dialog is closed. Therefore, we provide `FALSE` as the second argument to the SetArray call. When we execute the SetArray call and the array is "attached" to the CCatTable object, the code in the SetArray function will cause the list to be redrawn with any existing category names, automatically. It does this by adding as many rows as there are entries in the array and then calls the Refresh function of the CTable class to cause the GetCellText function to be called to furnish the contents to be drawn for each row. Our CCatTable class defines the GetCellText function (see page 237) for just this purpose. I should emphasize that anytime you create a CArrayPane object in the VA, you should create a derived class of CArrayPane (also within the VA) to implement either or both of the DrawCell and GetCellText functions. For the Categories dialog, the list is defined to be implemented by the

CCatList class (inheriting most of its functionality from the CArrayPane and CTable base classes, of course).

DisableButtons Function Code

When we first created the Categories dialog and called its BeginDialog function to cause it to be shown, we then called the DisableButtons function to disable the Use, Edit, and Delete buttons in the dialog. The code for that function is as follows:

```
void CCategories::DisableButtons (void)
{
   fCategories_UseCat->Deactivate();
   fCategories_EditCat->Deactivate();
   fCategories_DelCat->Deactivate();
}
```

CmdUseCat Function Code

When any of the Use, Edit, New, or Delete buttons is clicked, a corresponding command is dispatched (see pages 212–214 for the button command assignments) to the current gopher's DoCommand function. The VA has generated code in the DoCommand function of the x_CCategories class to handle the button commands by calling an appropriate function. In the case of the Use button, the CmdUseCat function is called. We have overridden the VA-generated code in the CCategories class, as follows:

```
void CCategories::DoCmdUseCat()
{
   Cell aCell = {0, 0};

   if (fCategories_CatTable->GetSelect (TRUE, &aCell))
   {
      ((CMain *)itsSupervisor)->SetSelectedCategory (aCell.v+1);
   }
}
```

In the foregoing code, we handle the "click command" for the Use button by accessing the category list to determine the currently selected item, and then call the SetSelectedCategory function in the CMain (itsSupervisor) object. That function is shown beginning on page 221.

CmdDeleteCat Function Code

The "click command" for the Delete button is also handled by the DoCommand function in the x_CCategories class. It calls the

CmdDeleteCat function, which we have overridden to provide the custom behavior we require. The code is as follows:

```
void CCategories::DoCmdDeleteCat()
{
   Cell theCell = {0, 0};

   if (fCategories_CatTable->GetSelect (TRUE, &theCell))
   {
      long index = theCell.v + 1;
      ((CMain *)itsSupervisor)->DelCategory (index);
   }
}
```

As for the Use button, the foregoing code determines the index of the selected category that the user wishes to delete and then calls the DelCategory function in the CMain class (itsSupervisor) to perform the operation. Recall that the CMain (document) class "owns" the array in which the categories are stored, so it is sensible for it to perform the management functions for that array's contents. The DelCategory function is shown on page 219.

CmdNewCat Function Code

We purposely left the explanations of the New and Edit button "click commands" until last, because both of these invoke the NewCat modal dialog to allow the user to create or edit a category entry. We will cover the source code for the CNewCat class shortly, but it is very similar in structure to what was presented earlier for the Notebook dialog (see pages 184–208). As with the other buttons in the Categories dialog, when the New button is clicked, the DoCommand function in the x_CCategories class is called to handle that command. The DoCommand function, in the case of the `cmdNewCat` command, calls the CmdNewCat function to open the NewCat dialog. You can see that it performs this task, automatically, by looking at the skeleton VA-generated code and the result of compiling and executing that code, shown in Figure 5-21. However, because we wish to customize the behavior of the NewCat dialog, the CmdNewCat function is overridden. The newly added code is as follows:

```
void CCategories::DoCmdNewCat()
{
   CNewCat *dialog;

   // create a new CCat object as the "settings"
   settings = TCL_NEW (CCat, ());
```

```
    // then create and  run the CNewCat dialog
    dialog = TCL_NEW(CNewCat,());
    dialog->ICNewCat(this);

    if (dialog->DoModalDialog(cmdNull) == cmdOK)
    {
       ((CMain *)itsSupervisor)->AddCategory (settings);
    }
    else
    {
       // if the user dismissed the dialog with Cancel,
       // we want to delete the CCat object.

       TCLForgetObject (settings);
    }
    TCLForgetObject(dialog);
}
```

The foregoing code is fairly straightforward. If you recall, we have defined a member variable in the CCategories class that holds a pointer to a CCat object (see page 228 for the definition in the **CCategories.h** header file). The foregoing code creates a new CCat object and stores its pointer into the `settings` variable. The contents of this variable are transferred to the NewCat dialog to establish the initial settings. In this case, a new category is being defined, so the settings are empty (we will cover the code for the CCat constructor function shortly).

The DoCmdNewCat function continues by creating the CNewCat object (`dialog`) and then running the dialog by calling DoModalDialog. During the time that the NewCat dialog is running, no other tasks can be undertaken, because it is a modal dialog. If the user dismisses the dialog with its OK button, then the foregoing code calls the AddCategory function of the CMain class (itsSupervisor) to add the newly specified category to the document's array. This addition will cause the CTable class, by virtue of the intimate connection between the CArray object and the table, to cause the newly added category to be drawn into the CCatTable list. This occurs as a result of the CArray object (the provider) calling the BroadcastChange function with a message type of `arrayInsertElement`, that the CArrayPane class (the dependent) interprets as an instruction to add a new row to the table. The CTable class handles the AddRow function call by calling the GetCellText function (which we override in our CCatTable class) to furnish the contents for the newly added row. In this case, our function will provide the newly added category name to that function. The AddCategory function code is shown beginning on page 218.

The DoCmdNewCat function also handles the case where the user dismisses the NewCat dialog with the Cancel button. In that case, because we have created a CCat object, we need to delete that object, because it will not be added to the document's array. In any case, the function completes its work by deleting the CNewCat dialog object.

DoCmdEditCat Function Code

When the user has selected an existing category name in the list and clicks the Edit button, the DoCommand function in the x_CCategories class handles this command by calling the DoCmdEditCat function, that we have overridden, as follows:

```
void CCategories::DoCmdEditCat()
{
   CNewCat *dialog;
   Cell aCell = {0, 0};

   // Get a pointer to the category to be edited, but don't do
   // anything if a cell isn't selected in the list of category
   // names.

   if (fCategories_CatTable->GetSelect (TRUE, &aCell))
   {
      long index = aCell.v + 1;
      settings = ((CMain *)itsSupervisor)->GetCategory(index);

      // Respond to command by opening a dialog

      dialog = TCL_NEW(CNewCat,());
      dialog->ICNewCat(this);
      if (dialog->DoModalDialog(cmdNull) == cmdOK)
      {
         ((CMain *)itsSupervisor)->SetCategory(settings, index);
      }
      TCLForgetObject(dialog);
   }
}
```

Unlike the DoCmdNewCat function, we do not have to create a CCat object. The object already exists. In this case, we determine which entry in the list the user has selected and access the object by calling the GetCategory function from the CMain class. Then the foregoing code "runs" the dialog. If the user dismisses the dialog with the OK button, then we call the SetCategory function of the CMain class to store the revised category description back into the document's `categories` array. If the user dismisses the dialog with the Cancel button, we don't have to do anything more. In either case, however, we delete the CNewCat dialog object.

ExchangeSettings Function Code

When the CNewCat dialog runs, we must provide it with initial settings. When the dialog is dismissed with the OK button, and just before it is disposed, we must receive the final settings of the dialog. Our ExchangeSettings function serves both purposes and is as follows:

```
void CCategories::ExchangeSettings (CCat *&aCat, Boolean from)
{
   if (from)
   {
      aCat = settings;
   }
   else
   {
      settings = aCat;
   }
}
```

Note that the foregoing function uses a reference to a CCat object pointer as its first argument and that the second argument indicates whether the contents of the pointer are to be loaded *from* the CCategories class, or the contents of the pointer are to be stored into the `settings` variable of the CCategories class. We will see how the ExchangeSettings function operates when we examine the code for the CNewCat dialog class.

Examining the CNewCat Dialog Code

The CNewCat class and its x_CNewCat base class implement a modal dialog in much the same fashion as has already been described for the Notebook dialog, earlier in this chapter. The NewCat dialog is modal, which means that when it is "running," no other actions can be taken by the user until it is dismissed.

As with the Notebook dialog (and the Categories dialog as well), the initialization code calls the MakeNewWindow function to create the dialogs window and its panes and then calls the BeginData function to specify the initial values for the dialog's panes. It is at the point of the call to BeginData that we will examine this dialog's code.

BeginData Function Code

We have customized the code for the BeginData function so that we can initialize the dialog's panes with values supplied by a call to

the ExchangeSettings function that was shown on page 234. The newly added code to the BeginData function is as follows:

```
void CNewCat::BeginData(CNewCatData *initial)
{
   Str255 name;

   CollectPaneValues(*initial);

   // Exchange the settings and specify the pane values
   // if new data are supplied.
   ((CCategories *)itsSupervisor)
      ->ExchangeSettings (itsCategory, TRUE);
   itsCategory->GetCatName (name);
   if (name[0] > 0)
   {
      // an existing category object was supplied, so use its
      // member values.

      itsCategory->GetCatName(initial->fNewCat_NewCatName);
      itsCategory->GetCatDescrip(initial->fNewCat_NewCatDescrip);
      if (itsCategory->GetCatType() == kNewCat_ExpenseID)
      {
         initial->fNewCat_Expense = 1;
         initial->fNewCat_Income = 0;
      }
      else
      {
         initial->fNewCat_Expense = 0;
         initial->fNewCat_Income = 1;
      }
      initial->fNewCat_NewTaxable = itsCategory->GetCatTaxable();
   }
}
```

I have kept the VA-generated call to the CollectPaneValues function, but then call the ExchangeSettings function to access the CCat object that contains the settings we wish to substitute for the initial values. I determine whether the object retrieved from the CCategories class is newly created by testing whether its `name` field is empty. If so, then the settings specified by the execution of the CollectPaneValues function are used. If, however, the `name` field is not empty, then the foregoing function extracts the settings for the various panes by calling the access functions in the CCat class to ascertain their values. I have stored a single variable in the CCat object to indicate which of the Expense or Income radio buttons was previously set and then use the identifier stored in that variable to determine the states of the two buttons. I also access the category name, its description string, and its tax-related status (checkbox) value. These values are stored into the `initial` structure that the VA-generated code has provided as an argument to the BeginData function.

EndData Function Code

In contrast to our approach to the Notebook modal dialog, we aren't interested in handling changes to the various settings until the user has dismissed the dialog. Therefore, I have not included any code to customize the UpdateData function for the CNewCat dialog class. The EndData function, however, is called when the dialog has been dismissed by the user. We are very interested in preserving the values that have been entered, but only if the OK button was used as the dialog's dismisser. The code is as follows:

```
void CNewCat::EndData(CNewCatData *final)
{
   if (dismissCmd == cmdOK)
   {
      // use the final values to update the category entry

      itsCategory->SetCatName (final->fNewCat_NewCatName);
      itsCategory->SetCatDescrip (final->fNewCat_NewCatDescrip);
      itsCategory->SetCatTaxable (final->fNewCat_NewTaxable);
      if (final->fNewCat_Expense > 0)
      {
         itsCategory->SetCatType (kNewCat_ExpenseID);
      }
      else
      {
         itsCategory->SetCatType (kNewCat_IncomeID);
      }
      ((CCategories *)itsSupervisor)
         ->ExchangeSettings (itsCategory, FALSE);
   }
}
```

The first thing to check in the foregoing code is the value of the `dismissCmd` variable, which holds the command associated with the button used to dismiss the dialog. If the command is `cmdOK`, then we access the values for each of the dialog's panes from the `final` data structure passed to the function and store these into the CCat object by using that object's access functions. When the user's settings have been stored into the local CCat object, we pass a reference to that object to the CCategories class by calling its ExchangeSettings function with a `FALSE` second argument. This causes the `settings` member variable in the CCategories class to be updated with the new values.

CNewCat Header File Additions

The foregoing BeginData and EndData functions are the only code that have been modified from what was generated automati-

cally by the VA; however, I did add a single member variable to the **CNewCat.h** header file. That declaration is as follows:

```
// Insert your own public data members here
CCat *itsCategory;    // newly added
```

You'll note that the foregoing is simply a placeholder for the CCat object pointer that is passed into and out of the CNewCat object when the ExchangeSettings function is called.

Examining the CCatTable Class Code

I created the category name list element of the CCategories class as a CCatTable object that was derived from the CArrayPane class. Creating the derived class for the table was necessary for us to override the GetCellText function of the CArrayPane class. By overriding that function, we are able to supply the text for a category name when the CTable class calls the function. The GetCellText function is all that we have changed and it is as follows:

```
void CCatTable::GetCellText(Cell aCell, short availableWidth,
   StringPtr itsText)
{
   CCat     *aCat;
   long     item = aCell.v + 1;
   CArray   *itsArray = GetArray();
   itsArray->GetArrayItem (&aCat, item);
   aCat->GetCatName (itsText);
}
```

The foregoing code uses the vertical coordinate of the `aCell` argument to the function to access the specified item in the list's associated array and use the item pointer to access the category name for the item, storing it into the `itsText` argument.

Examining the CCat Class Code

In order to encapsulate all of the information relevant to a category entry, I defined a new class called CCat and created the source and header files for this class using the Symantec Project Manager's text editor. Because objects such as this should be encapsulated fully, I defined all of its member variables to be protected and then provided access functions to enable the values of those variables to be retrieved or changed.

CCat Class Header File

The header file for the CCat class is as follows:

```
/************************************************************
 CCat.h

        Header File For CCat class

   Copyright © 1995 Richard O. Parker. All rights reserved.

************************************************************/

#pragma once

class CString;

class CCat TCL_AUTO_DESTRUCT_OBJECT
{
   public:

      TCL_DECLARE_CLASS

      CCat();
      virtual ~CCat();

      void       PutTo(CStream&);
      void       GetFrom(CStream&);

      void       GetCatName(StringPtr name);
      void       SetCatName (StringPtr name);
      void       GetCatDescrip(StringPtr descrip);
      void       SetCatDescrip (StringPtr descrip);
      long       GetCatType ();
      void       SetCatType (long type);
      short      GetCatTaxable();
      void       SetCatTaxable (short taxable);

   protected:

      CString    catName;
      CString    catDescrip;
      long       catType;
      short      catTaxable;
};
```

All of the foregoing declarations are newly added. I have used CString objects to hold the `catName` and `catDescrip` strings because of the ease with which these objects can be manipulated.

CCat Constructor Function Code

The constructor for the CCat class initializes the object with default values. The code is as follows:

```
CCat::CCat()
{
   catName = "";
```

```
   catDescrip = "";
   catType = 0L;
   catTaxable = 0;

   TCL_END_CONSTRUCTOR
}
```

As is apparent, the fields of the CCat object are initialized to empty values so that a newly created object can be distinguished from one whose values have been specified fully.

PutTo and GetFrom Function Code

We have defined PutTo and GetFrom functions for the CCat object so that we may add Object I/O support for reading in or writing out these objects. At this point in the implementation of the code, the functions are empty and serve only to remind us that the code should be provided at some later date. The code for these functions is as follows:

```
/*************************************************************
 PutTo

   Writes a CCat object onto the stream.
*************************************************************/

void CCat::PutTo (CStream& stream)
{
}

/*************************************************************
 GetFrom

   Reads a CCat object from the stream.
*************************************************************/

void CCat::GetFrom (CStream& stream)
{
}
```

I show the foregoing code only because it is important to stress that you should make provision for adding these functions to each of your newly created objects, if you intend for them to be read from or written to an external file.

CCat Class Access Function Code

The next series of functions provide the ability for the values associated with the member variables in the object to be retrieved and stored. These functions are very simple, but it is important to create them because they allow the CCat object to be treated as a

"black box," rather than exposing all of its internal structure to the developer. Such "information hiding" principles are a major objective of object-oriented technology. If you get into the habit of creating access functions for all of your object's member variables, you will have fewer problems when the functionality of the object needs to change. As long as you preserve the calling conventions (that is, the object's interfaces), you can change every aspect of an object, without causing any interference to existing code.

The code for the CCat object's access functions is as follows:

```
/************************************************************
 GetCatName

   Retrieves the name of a CCat object.
************************************************************/

void CCat::GetCatName(StringPtr name)
{
   catName.GetPStr (name);
}

/************************************************************
 SetCatName

   Specifies the name for a CCat object.
************************************************************/

void CCat::SetCatName (StringPtr name)
{
   CString s (name);
   catName = s;
}

/
************************************************************
GetCatDescrip

   Retrieves the description for a CCat object.
************************************************************/

void CCat::GetCatDescrip (StringPtr descrip)
{
   catDescrip.GetPStr (descrip);
}

/************************************************************
 SetCatDescrip

   Specifies the description of a CCat object.
************************************************************/

void CCat::SetCatDescrip (StringPtr descrip)
{
   CString s (descrip);
   catDescrip = s;
}

/************************************************************
 GetCatType
```

```
   Retrieves the expense or income type of a CCat object.
************************************************************/

long CCat::GetCatType()
{
   return catType;
}

/************************************************************
 SetCatType

   Specifies the expense or income type of a CCat object.
************************************************************/

void CCat::SetCatType (long type)
{
   catType = type;
}

/************************************************************
 GetCatTaxable

   Retrieves the tax-related status of a CCat object.
************************************************************/

short CCat::GetCatTaxable()
{
   return catTaxable;
}

/************************************************************
 SetCatTaxable

   Specifies the tax-related status of a CCat object.
************************************************************/

void CCat::SetCatTaxable (short taxable)
{
   catTaxable = taxable;
}
```

The foregoing functions provide the means to retrieve and store the values of each of the object's member variables. Although each function consists of only one or two lines of code, by providing these functions to access the object's member variables, we are able to insulate the object fully from its clients.

Creating a Dynamic Modeless Dialog

One of the interesting features of the VA is its ability to create subviews that can be shared between multiple views and also swapped into and out of views, whether they are windows or dialogs. This section describes a modeless dialog that includes several (two in our example) subviews that are swapped into the basic dialog, when the user selects the options to which they pertain.

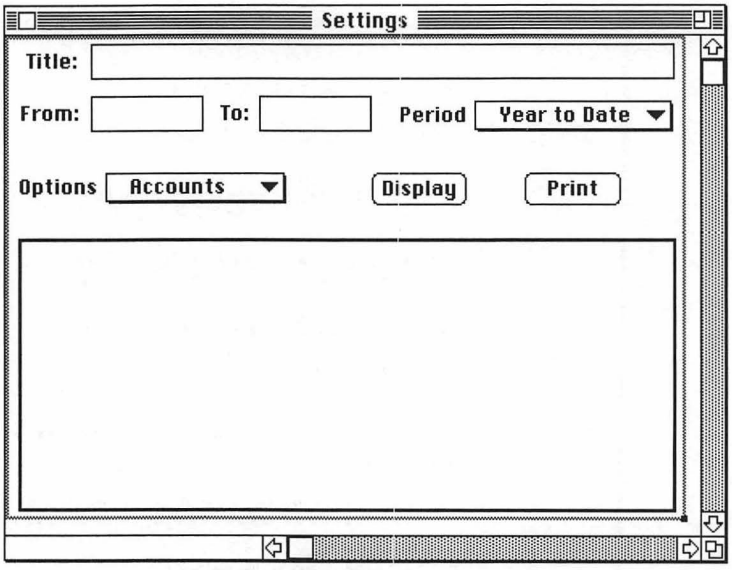

Figure 5-22
Main Settings dialog

For this example, I hypothesize that a business application might require a set of reports and that the reports might each have various options. Rather than create a separate options dialog for each set of options, I have elected to create a single dialog that contains a section at its bottom in which various subviews can be displayed. I start out by creating the base dialog, as shown in Figure 5-22.

Note in the foregoing figure that a bordered area (created with the rectangle tool) has been set aside at the bottom of the dialog. This is to hold the various subviews that are to be swapped into and out of the dialog. The dialog itself is not remarkable. The various elements are self-explanatory. The pop-up menu to the right of the Options label selects which subview is to be displayed. The View Info for the dialog is shown in Figure 5-23.

After creating the rectangular frame at the bottom of the dialog, I jotted down its location and dimensions in the dialog's window. The width and height are needed to create the subviews in the VA, and the location of the top-left corner is needed when placing the subview into the view in the application's code.

After the main view has been created, we can create a menu named **Report** whose single command is **Settings**. We might wish

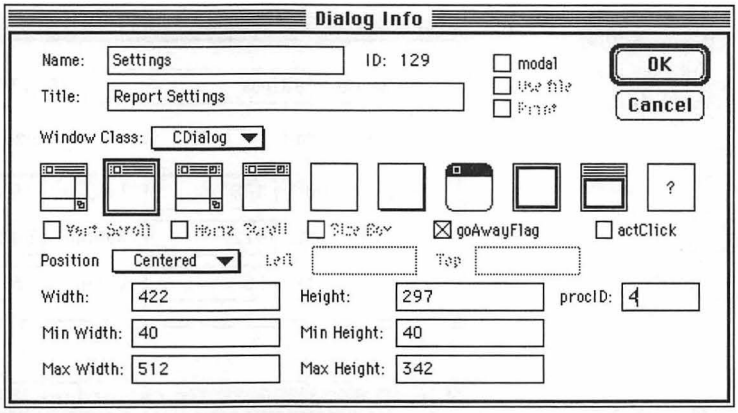

Figure 5-23
View Info for Settings dialog

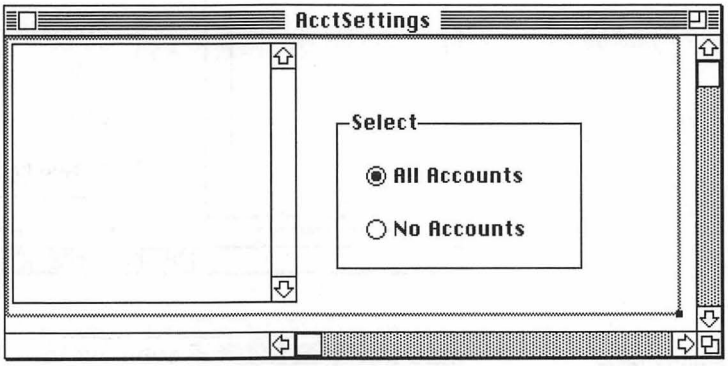

Figure 5-24
Completed AcctSettings subview

to add other commands at a later date, but, for the time being, the **Settings** command is the only entry. The behavior of that command is to "Open" the CSettings dialog when the command is chosen by the user.

The next step is to create the subviews. To do so, choose **New View** from the **View** menu, and select the Subview type from the pop-up menu. The VA will present you with a new window in which to construct the subview. The first subview is named AcctSettings. Its completed appearance is shown in Figure 5-24. The Subview Info settings for the subview (choose **View Info** from the **View** menu for this subview) are shown in Figure 5-25. Note in the figure that the subview width is the same as the width of the rectangular area we outlined in the Settings dialog, and that the height is also the same. The Bounds and Step settings aren't relevant to this usage.

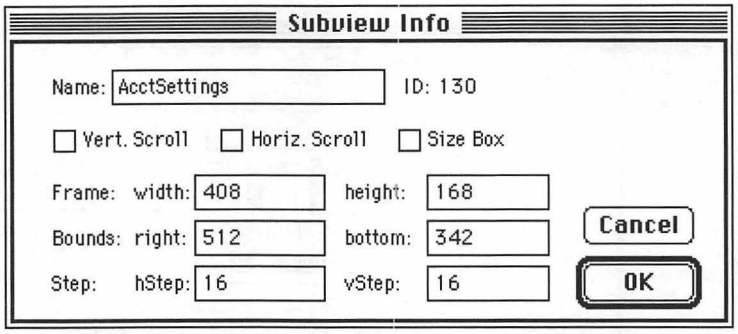

Figure 5-25
Subview Info for AcctSettings subview

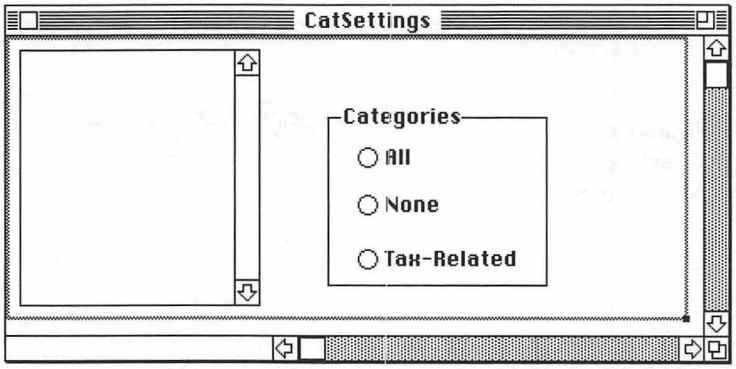

Figure 5-26
Completed CatSettings subview

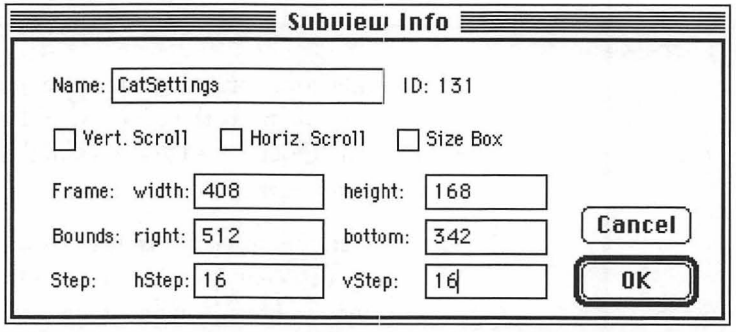

Figure 5-27
Subview Info for CatSettings subview

The next step is to construct the CatSettings subview in order to provide the user the ability to select the categories to be reported. The completed CatSettings subview is shown in Figure 5-26, and the Subview Info settings are shown in Figure 5-27.

After the foregoing elements have been created (I have purposely not spent any time discussing the details of doing so), you can generate code and begin the implementation of the custom code

that is needed to swap the subviews into and out of the Settings dialog. The main purpose of this section was to show how the subviews could be swapped easily, using the built-in features of the TCL to do so.

Examining the x_CMain Code

The generated code in the x_CMain class for this example is identical to most of the other code presented previously for this class. One difference, however, is the addition of a case in the DoCommand function to handle the user's choice of the Settings command from the Report menu. The DoCommand code, as generated by the VA, is as follows:

```
void x_CMain::DoCommand(long theCommand)
{
   switch (theCommand)
   {
      case cmdOpenSettings:
         DoCmdOpenSettings();
         break;
      default:
         CDocument::DoCommand(theCommand);
   }
}
```

Note that the foregoing code tests whether the `cmdOpenSettings` has been chosen, and if so, it calls the DoCmdOpenSettings function. The code for that function is as follows:

```
void x_CMain::DoCmdOpenSettings()
{
   CSettings *dialog;

      // Respond to command by opening a dialog

   dialog = TCL_NEW(CSettings,());
   dialog->ICSettings(this);
   dialog->BeginDialog();
}
```

As was the case with the Categories dialog, the Settings dialog is created by constructing a new CSettings object, initializing the object, and then calling its BeginDialog function. All of the logic to swap subviews into and out of the Settings dialog is contained in the CSettings class itself.

Examining the CSettings Code

The VA-generated code in the CSettings class was modified to provide the functionality required for this dialog. The first step in doing so was to specify new member variables and functions that would be needed. The contents of the **CSettings.h** header file that shows these features are as follows:

```
/************************************************************
 CSettings.h

          CSettings Dialog Director Class

   Copyright © 1995 Richard O. Parker All rights reserved.

   Generated by Visual Architect™ 2:00 PM Mon, Dec 12, 1994

   This file is only generated once. You can modify it by filling
   in the placeholder functions and adding any new functions you
   wish. If you change the name of the dialog class, a fresh
   version of this file will be generated. If you have made any
   changes to the file with the old name, you will have to copy
   those changes to the new file by hand.

************************************************************/

#pragma once

#include "x_CSettings.h"


class CDirectorOwner;
class CSubviewDisplayer;
class CView;
class CArrayPane;
class CRectOvalButton;
class CStaticText;
class CRadioButton;

enum subviewType
{
   kAcctSettings,
   kCatSettings
};

class CSettings : public x_CSettings
{
public:

   TCL_DECLARE_CLASS

   // Pointer to current subview in Settings window & subview ID

   CSubviewDisplayer  *fSettingsSubview;    // subview pointer
   subviewType        fSubviewID;           // subview type

   // Account subview-related controls
   CArrayPane         *fAcctSettings_AcctNames; // Account Names
   CRectOvalButton    *fAcctSettings_Rect7;     // Account Radio
                                                // Group Rect
```

```
   CStaticText        *fAcctSettings_Stat8;    // Static Text
   CRadioButton       *fAcctSettings_SelAllAccts;// All Radio
   CRadioButton       *fAcctSettings_SelNoAccts;// None Radio

   void ICSettings(CDirectorOwner *aSupervisor);

   virtual void    ProviderChanged(CCollaborator *aProvider,
      long reason, void* info);
   virtual void    DoCommand(long theCommand);

protected:

   virtual void BeginData(CSettingsData *initial);
   virtual void UpdateData(CSettingsUpdate *update, long itemNo);
   virtual void EndData(CSettingsData *final);

   virtual void MakeNewWindow (void);    // override
};
```

In the foregoing header file, I have defined an enumerator called subviewType that defines the two subviews that we created (`kAcctSettings` and `kCatSettings`). These values are used only to keep us aware of which subview currently resides in the Settings dialog. The value for that is stored in the `fSubviewID` variable. I have provided a variable to hold a pointer to a CSubviewDisplayer object, which was created for the purpose of displaying the various subviews.

In addition, I have created variables to hold pointers to the various controls in each of the subviews when they are active. The use of these variables will become clear when I show the code to create and display the subviews.

MakeNewWindow Override Function Code

The logic to display the initial subview (AcctSettings by default) is contained in the MakeNewWindow override function provided. The code for that function is as follows:

```
void CSettings::MakeNewWindow (void)
{
   x_CSettings::MakeNewWindow();// call the base class version

   // load the initial subview into the newly created dialog
   // and specify the type of subview being displayed.

   fSettingsSubview = TCL_NEW (CSubviewDisplayer,
      ("\pAcctSettings", itsWindow, this, 404, 164, 10, 126));
   fSubviewID = kAcctSettings;
}
```

The foregoing code calls the base class version of the MakeNewWindow function and then creates a new CSubviewDisplayer object whose constructor takes the name of a subview (as defined in the VA), the subview's enclosure, its supervisor (`this`), its width, height, and its horizontal and vertical offset values. The subview is located, constructed by the Object I/O code in the TCL (this process is described in a later chapter), and the subview is displayed. After constructing the subview, I indicate which one is being displayed currently by storing its enumeration value into the `fSubviewID` variable.

Once the initial subview has been created, we can rely on the ProviderChanged function being called whenever the user chooses a different subview name from the pop-up menu in the main portion of the dialog.

ProviderChanged Function Code

The ProviderChanged function implements swapping the subviews into and out of the CSubviewDisplayer object. The code to do this is as follows:

```
void CSettings::ProviderChanged(CCollaborator *aProvider,
   long reason, void* info)
{
   Boolean saveIgnore = ignore;
   short selectedItem;

   if (ignore)
   {
      return;
   }
   ignore = TRUE;

   TRY
   {
      if (reason == popupMenuNewSelection
         && aProvider == fSettings_Options)
      {
         selectedItem = *(short *) info;
         switch (selectedItem)
         {
            case 1:// Account Options
            {
               if (fSubviewID == kAcctSettings)
               {
                  return;
               }
               fSettingsSubview->Empty();
               fSettingsSubview->SetViewName ("\pAcctSettings");
               fSettingsSubview->Fill();
               fSubviewID = kAcctSettings;
               break;
            }
            case 2:// Category Options
```

```
            {
               if (fSubviewID == kCatSettings)
               {
                  return;
               }
               fSettingsSubview->Empty();
               fSettingsSubview->SetViewName ("\pCatSettings");
               fSettingsSubview->Fill();
               fSubviewID = kCatSettings;
               break;
            }
            case 3:// Payment Options
            {
               break;
            }
            case 4:// Deposit Options
            {
               break;
            }
         }
      }
      else
      {
         x_CSettings::ProviderChanged(aProvider, reason, info);
      }
   }
   CATCH
      ignore = saveIgnore;
   ENDTRY

   ignore = saveIgnore;
}
```

I have written the ProviderChanged function according to the pattern of that function in the base class. That is, I defined the `ignore` variable, tested to see whether it is `TRUE` and returned if so, and then set it to `TRUE` before continuing the process of testing what type of semantic event has occurred.

The main body of the function is concerned with only one type of semantic event. This is the `popupMenuNewSelection` event that is broadcast by the `fSettings_Options` pop-up menu control. If the reason or provider don't match these values, then the ProviderChanged function in the base class is called to handle the event.

When the user chooses one of the entries in the Options pop-up menu, a pointer to the item number is stored into the `info` argument that is passed to the ProviderChanged function. A switch statement is executed to test the value of the `info` argument to determine which item has been chosen. Each item corresponds to a subview. I have made provisions for Payment and Deposit subviews to be defined at a later date. If the chosen item corresponds to the AcctSettings subview (item 1), then we test whether that

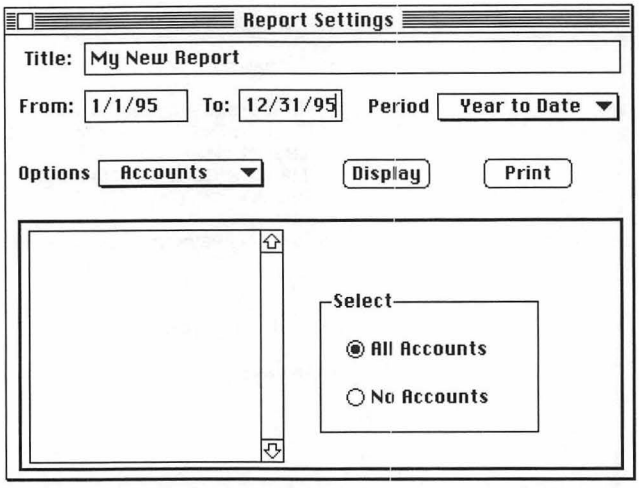

Figure 5-28
Report Settings dialog with AcctSettings subview shown

subview is already being displayed. This can occur if the user clicks the pop-up menu and chooses the item that is already the current selection. In that case, we simply return. If the AcctSettings subview is not currently being displayed, then we call the Empty function of the CSubviewDisplayer object (`fSettingsSubview`) to remove the current subview from the dialog and dispose of it. Then we call the SetViewName function to specify the name of the subview that we intend to load, and then call the Fill function to create and display the new subview. After doing so, we set the value of the `fSubviewID` variable to reflect the fact that the AcctSettings subview is currently being displayed.

The process for handling the CatSettings subview is the same as for the AcctSettings subview. All of the machinations needed to create and dispose of the various elements in the subviews are handled automatically by the TCL. The appearance of the Settings dialog with the AcctSettings and CatSettings subviews installed is shown in Figures 5-28 and 5-29.

Modal and Modeless Dialog Summary

The three examples in this chapter illustrated modal, modeless, and dynamic modeless dialogs. It was probably evident to you that their structures were quite similar, and, in fact, they are. However, some of their individual properties are as follows:

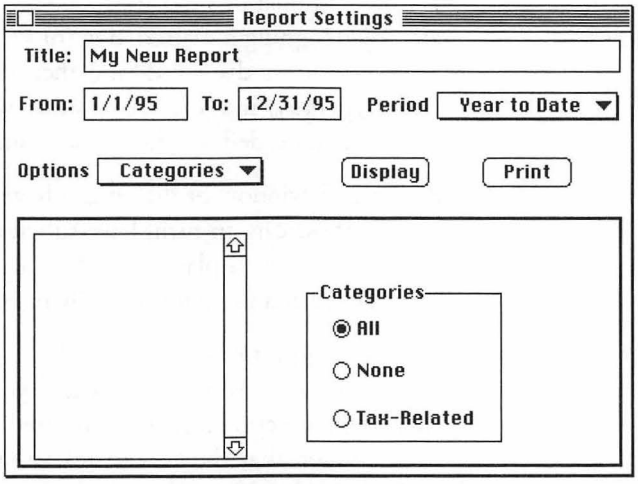

Figure 5-29
Report Settings dialog with CatSettings subview shown

- A modal dialog sets up its own event loop and requires the user to confine his/her activities to the controls and fields in the dialog until it is dismissed.
- A modeless dialog, once opened, allows the user to freely interact with the menu bar and other windows. It is, in effect, quite similar to an ordinary window; however, its director is a derived class of CDialogDirector.
- The exchange of settings between the object that invokes a modal dialog and the dialog itself usually takes place only when the dialog is first created (in the BeginData function) and when it is dismissed (in the EndData function).
- Modeless dialogs, because they are fully accessible while they are open, require that you make provision for storing changes to their settings as the changes occur. A good way to provide this exchange facility is to create access functions in the object that creates the dialog, so that when the dialog is finally closed, the settings will be preserved.
- Modal dialogs are closed by using a "dismisser," which is usually an OK or Cancel button (but can be any object that is capable of issuing a command).
- Modeless dialogs are closed when the user clicks the dialog's close box (or by any other means that causes the Close function to be called).

- Dynamic modal or modeless dialogs (or windows) can be created by designing a main dialog (or window), leaving space for a subview to be displayed, and then creating each of the subviews separately in the VA. The subviews can be loaded dynamically, when needed, by the application.
- A dynamic window or dialog can have multiple subviews, and each of these can, in turn, have subviews, nested to any depth. Each subview is displayed in a CSubviewDisplayer object that must be created programmatically in your custom code.

The foregoing points are "typical" for modal and modeless dialogs. They are not cast in concrete and you are free to experiment to change the functionality demonstrated in the examples. Bear in mind, however, that the TCL assumes that you will be following the guidelines that I have described.

The next chapter describes various types of controls, how they are created in the VA, how they operate, and the various types of semantic events that they generate.

Chapter 6

Creating and Managing Controls

This chapter describes various controls that you can create with the VA and also programmatically, with the help of the TCL. The main thing I'm going to try to do is explain how to customize controls and manage the "semantic events" they produce when operated by the user.

In earlier days, controls were limited to buttons, checkboxes, and scroll bars; however, the word "control" has now become a generic definition of any subpane of a window upon which the user can exercise some type of control. This includes both static and editable text fields, tables and visual lists of various types, "progress" gauges, pop-up menus, scroll bars, tear-off menus, and a large number of buttons of all varieties. In fact, it is almost impossible to enumerate all of the various types of controls that are contained in the full spectrum of today's Macintosh applications. Because of this, we will limit our focus to the controls that are supported directly by the VA and the TCL.

So What *Is* a Semantic Event?

Controls, in the large sense described in the foregoing explanation, are manipulated by the user to inform the application about actions that they mean for it to execute. When we select an element from a list or type some text into an Edit Text field, we are informing the application that we wish for it to accept these inputs and do the proper thing. In addition, the actions of the application itself may require that another section of the same application respond to that action. In support of these ideals, the TCL includes a base class called CCollaborator, from which most of the control classes derive their functionality. In addition to controls, all of the classes derived from CView descend from CCollaborator, as do the CApplication, CDocument, and a grand total of nearly 80 classes—the bulk of the TCL.

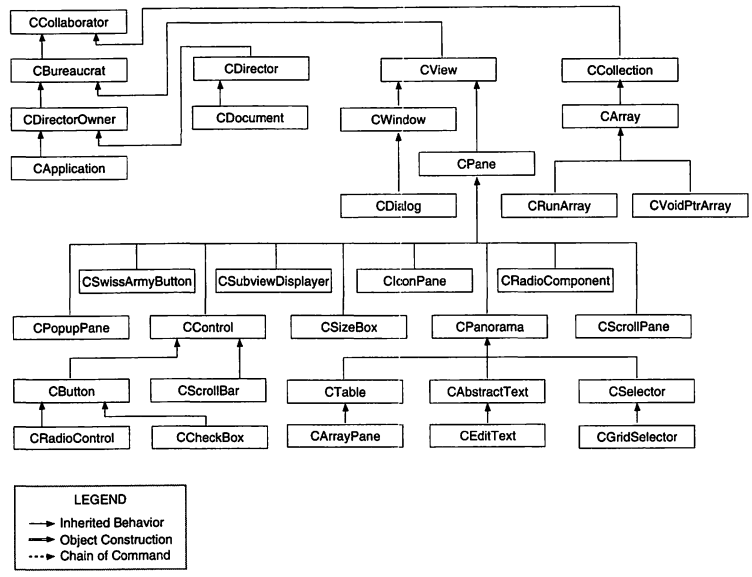

Figure 6-1
The CCollaborator class and its descendents

Because of the intimate relationship of these classes to one another—by virtue of their common ancestry—they are in a perfect position to communicate one with the other. You might wonder what type of inter-object communications would be useful in this regard, and rightly so. I will describe the specifics when discussing each type of control, later in this text. But for now, I will describe the "collaboration mechanism" and its support for handling semantic events.

Many of the common descendents of the CCollaborator class are shown in Figure 6-1. However, bear in mind that the diagram is not exhaustive, though it does show most of the commonly used classes. Note that "collaborators" include the application object itself, the document object, any windows, dialogs, and their directors. Collaborators also include all of the descendents of CPane and all of the CCollection class derivatives (although only two of these are shown).

So what does it mean to be a collaborator, anyway? First of all, it means that any of the classes in the hierarchy shown in Figure 6-1 can communicate with one another if they arrange to do so. This is accomplished by sending a BroadcastChange message (call the function by that name) with a reason for the call and a pointer to any important information that needs to be passed on to the re-

ceiver. The BroadcastChange message travels up the hierarchy, pausing briefly in the CBureaucrat's BroadcastChange function (more about that later), and then proceeds to the BroadcastChange function in the CCollaborator class.

Each collaborator object has a pair of lists (CPtrArray objects) that contain dependents and providers associated with the object. If the object intends to be a "receiver" of semantic events, then it calls the DependUpon function with a pointer to the provider object whose events it intends to handle. The DependUpon function adds the pointer to the specified provider to the object's list of providers (`itsProviders`) and then calls the provider's AddDependent function with its own pointer. Therefore, when the BroadcastChange function for a collaborator is called, the function iterates through its list of dependents (`itsDependents`), calling the ProviderChanged function for each one in the list, with the identity of (pointer to) itself (`this`), and the reason and additional information data passed in the BroadcastChange function call.

The foregoing description is demonstrated quite succinctly by the code example for the Categories view, described in BeginData function code shown on page 235. Specifically, I am referring to the SetArray function call, which establishes a "connection" between the CArray object holding the category names and the CCatTable object (derived from CArrayPane) that displays the contents of the array in its cells. The connection in this case is through the collaboration mechanism. The CArray object is the provider, and the CCatTable object is the dependent in this case. The code for the SetArray function of the CArrayPane class is quite illuminating and is as follows:

```
void CArrayPane::SetArray(CArray *anArray, Boolean fOwnership)
{
   short    deltaRows;
   Long     Pt toPt = {0,0};

   TCL_ASSERT(anArray != NULL);

   if (itsArray)
   {
      if (itsArray->GetNumItems() > 0)
         ScrollTo(&toPt, FALSE);
      if (ownsArray)
         TCLForgetObject(itsArray);
      else
         CancelDependency(itsArray);
   }
```

```
    itsArray = anArray;
    ownsArray = fOwnership;

    DependUpon(itsArray);

       // make sure table has same number of rows as the new array
       // has elements.

    deltaRows = itsArray->GetNumItems() - tableBounds.bottom;

    if (deltaRows > 0)
       AddRow(deltaRows,  tableBounds.bottom-1);
    else if (deltaRows < 0)
    {
       deltaRows = -deltaRows;
       DeleteRow(deltaRows, tableBounds.bottom - deltaRows);
    }
    Refresh();
}
```

The foregoing code first determines whether there is an existing array attached to the dependent object (CCatTable in this case). If so, then the following actions are taken:

- If the array contains any items, the table is scrolled to the first item in the array. This prevents any rendering problems when the array ceases to exist.
- If the table "owns" the array, then that array is disposed by calling the `TCLForgetObject` macro.
- If the array is *not* owned by the table, then the CancelDependency function is called, with the existing array as its argument.

When the foregoing actions are complete, the table's array (`itsArray`) variable is set to point to the new array, the ownership of the array is recorded, and the table's DependUpon function is called, with the array as its provider.

For example, if a new entry is added to the table's CArray object, its InsertAtIndex function is called. The code for the InsertAtIndex function in the CArray class of the TCL is as follows:

```
void CArray::InsertAtIndex(void *itemPtr, long index)
{
   InsertItem(index);
   Store(itemPtr, index);
   BroadcastChange(arrayInsertElement, &index);
   if (itsIterators)
      itsIterators->ArrayChanged(arrayInsertElement,
         (void*) &index);
}
```

As you can see in the foregoing code, the item is inserted into the array (InsertItem followed by Store function calls), and then the BroadcastChange function is called with a reason of `arrayInsertElement` and a pointer to the index of the element that was inserted.

Following the foregoing BroadcastChange call into the Array's BroadcastChange function reveals the following code:

```
void CCollaborator::BroadcastChange(long reason, void* info)
{
   long nullInfo;

   // We allow the senders of BroadcastChange to pass NULL for
   // info. However, assuming dereferencing NULL is bad, it is
   // easier for receivers of ProviderChanged if dereferencing
   // info is always safe. Therefore, when info is NULL, we
   // substitute of pointer to 0L

   if (itsDependents)
   {
      if (!info)
      {
         nullInfo = 0L;
         info = &nullInfo;
      }

      // We use a CVoidPtrArrayIterator to avoid template
      // overhead. itsDependents is guaranteed to contain
      // collaborators,

      CVoidPtrArrayIterator iter(itsDependents, kStartAtBeginning);
      CCollaborator *aDependent;
      while (iter.Next((void*) aDependent))
         aDependent->ProviderChanged(this, reason, info);
   }
}
```

The foregoing code in the TCL illustrates how the BroadcastChange function call is handled. If the object's `itsDependents` list is empty, then nothing needs to be done; otherwise, after making sure that a valid pointer is created for the `info` argument, the function iterates through its list of dependents, calling the ProviderChanged function for each one, with a pointer to the current object, the reason for the call, and the additional `info` pointer.

Because we know that the CCatTable is the one and only dependent for the `categories` CArray object in our example, we know that the ProviderChanged function of the CArrayPane class (from which the CCatTable object is derived) will be called. Its code is as follows:

```
void CArrayPane::ProviderChanged(CCollaborator *aProvider,
   long reason, void* info)
{
   tMovedElementInfo  *moveInfo;
   Rect               cellRect;
   long               infoIndex = *(long*) info;

   if (aProvider == itsArray)
   {
      switch(reason)
      {
         case arrayInsertElement:
            // new element inserted, add a new row
            AddRow(1, infoIndex - 2);
            break;

         case arrayDeleteElement:
            // element was deleted, delete its row
            DeleteRow(1, infoIndex-1);
            break;

         case arrayMoveElement:
            // an element was moved, redraw all
            // elements encompassing its old and
            // new positions
            moveInfo = (tMovedElementInfo *) info;
            cellRect.left = 0;
            cellRect.right = tableBounds.right;
            cellRect.top = Min(moveInfo->originalIndex,
               moveInfo->newIndex) -1;
            cellRect.bottom = Max(moveInfo->originalIndex,
               moveInfo->newIndex);
            RefreshCellRect(&cellRect);
            break;

         case arrayElementChanged:
            // an element was changed, redraw its row

            cellRect.left = 0;
            cellRect.right = tableBounds.right;
            cellRect.top = infoIndex-1;
            cellRect.bottom = infoIndex;
            RefreshCellRect(&cellRect);
            break;

         default:
            CTable::ProviderChanged(aProvider, reason, info);
            break;
      }
   }
   else
      CTable::ProviderChanged(aProvider, reason, info);
}
```

The foregoing code is fairly straightforward. It includes cases to handle the `arrayInsertElement`, `arrayDeleteElement`, and `arrayElementChanged` reason codes for its array. Any other reasons, or if the provider isn't the current table's array, cause the code to call the ProviderChanged function of its base class (CTable).

So in answer to the original question, "So What *Is* a Semantic Event?" we can now answer that it is an event that communicates the *meaning* of a particular action during the course of an application's execution.

The TCL supports many such semantic events and we will cover most of these in the sections that follow. Before going on to do so, however, it is also important to discuss one last ramification of the calling of the BroadcastChange function. If you recall, I mentioned earlier that when that function was called, there was a brief pause in the implementation of that function in the CBureaucrat class. I will now describe the essence and meaning of that "pause" by showing the code, which is as follows:

```
void CBureaucrat::BroadcastChange(long reason, void* info)
{
   if (itsSupervisor)
      itsSupervisor->ProviderChanged(this, reason, info);

   CCollaborator::BroadcastChange(reason, info);
}
```

The BroadcastChange function in the CBureaucrat class pauses to determine whether the provider calling the function has a defined supervisor. If so, then the supervisor (`itsSupervisor`) object's ProviderChanged function is called immediately, with a pointer to the provider, and the reason and added information data. Following that call, the BroadcastChange function of the CCollaborator class is called. If you refer to that code, you will see that in the case where the `itsDependents` list is empty, no further action will be taken.

By calling the ProviderChanged function of the supervisor of the provider object, the CBureaucrat recognizes that in many cases there will be no explicit, registered, dependents to recognize and handle a given provider's semantic events. This allows us to put a ProviderChanged function directly into the CDirector-derived objects of our application (for example, our CApplication-derived, CDocument-derived, and CDialog-derived classes) and handle semantic events at that level. This is convenient because we would otherwise have to use the DependUpon mechanism for each provider whose actions we wish to monitor at the document or application level.

It is very convenient to handle semantic events at the document, dialog, or application level. Although this causes a lot of traffic through the ProviderChanged function, we can make it operate as efficiently as possible by checking whether the provider object in the call is of interest and whether any of its reasons are of interest.

As a final note with regard to semantic events, it is useful to know that you can create your own classes, derived from the CCollaborator class, and use the collaboration mechanism to communicate between your own objects, using semantic messages of your own design. This might be important in an application where you have a lot of cooperating objects that need to know what their peers are doing at any point in time.

Learning About Buttons

A CButton object is one of the most "pure" types of controls. In fact, if you look at Figure 6-1, you will see that CButton is one of only two "true" derived classes of the CControl base class (CScrollBar being the other). The TCL supports other types of button controls as well, such as those derived from the CSwissArmyButton class and the CIconPane class.

Pure CButton objects do not generate semantic events; instead, they generate commands, which, like menu commands, are handled by the DoCommand function of the button's supervisor (`itsSupervisor`). You will hear the term "click command" being applied to this type of event. CRadioControl objects are different in this respect and I will cover that difference shortly.

The buttons covered by this section include standard push buttons, icon buttons, picture buttons, radio buttons, checkboxes, round rectangle buttons, oval buttons, polygon buttons, and even lines! All of these are descendents of the CCollaborator class, and all are descendents of the CPane class. Just remember that a window is populated by panes of various types, and some of these can be control panes as well. There is no longer anything sacred about placing controls only into dialog windows. Because the original Mac toolbox supported rendering and operation of controls with its Dialog Manager, this concept has survived far too long. Mod-

ern user interfaces contain toolbars, buttons, palettes, and all variety of controls in both modal and modeless dialogs and in windows. The TCL supports all of these without the assistance of the Dialog Manager. The code to render and manage modern interface controls is part and parcel of the framework's code.

Examining the CButton Class

As indicated previously, the CButton class implements "pure" push buttons, according to the original Macintosh Interface Guidelines. Although the VA allows you to create CButton objects in any size, it uses a "standard" size of 59x20 pixels when you choose the Button tool and "click" anywhere in your window or dialog. The standard button is placed in that position (the crosshairs specify the top-left corner of where the button should be placed) and its size is set to the standard width and height. On the other hand, if you click and drag the mouse after choosing the Button tool, you can create a button with any dimensions you desire. It will always be a standard rounded rectangle button, though.

Button Properties

If you create a simple button called "Push Me" using the VA's Button tool, you will see that the member properties of the button's class hierarchy are as shown in Figure 6-2.

Let's examine these properties to gain an understanding of the flexibility of CButton objects and any other object derived from the CView base class. The important properties are as follows:

- The button's name (Push Me) has been entered into the "Identifier" field at the top of the properties. Immediately below that, the Left and Top positions, as well as the Width and Height, are specified. Any of these can be changed by entering new values into the corresponding fields, and the position and size of the button will change to match them.

- The member properties of the various classes follow, in order, from the most derived class (CButton) to the earliest base class (CView). A single "Command" property is provided for the CButton class. It is set to `cmdNULL` by default, but I have defined a "click command" for this button of `cmdPushMe`. This is accomplished as follows:

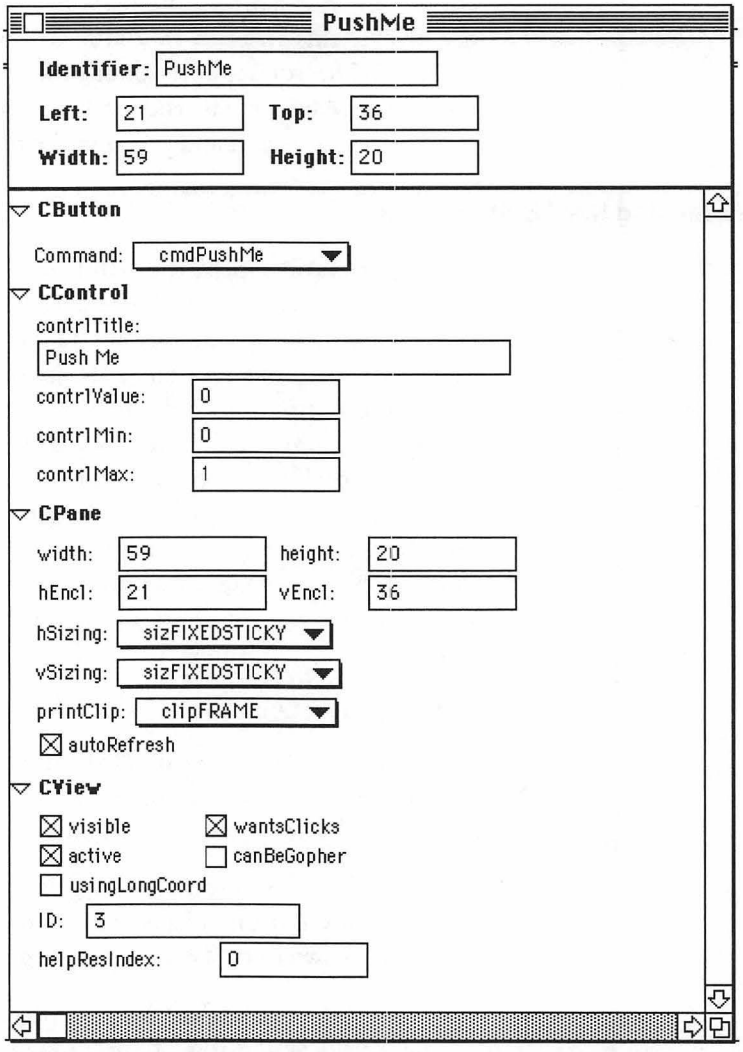

Figure 6-2
Member properties of a CButton object inside the VA

- Click on the pop-up menu that specifies the initial `cmd-NULL` value, and select the **Other** entry at the top of the menu. This will cause the VA to display the Commands dialog.
- Choose **New Command** from the **Edit** menu (or use the Command-K keyboard shortcut), enter the values shown in Figure 6-3, and then click the OK button in the Commands dialog to complete the entry and dismiss the dialog.

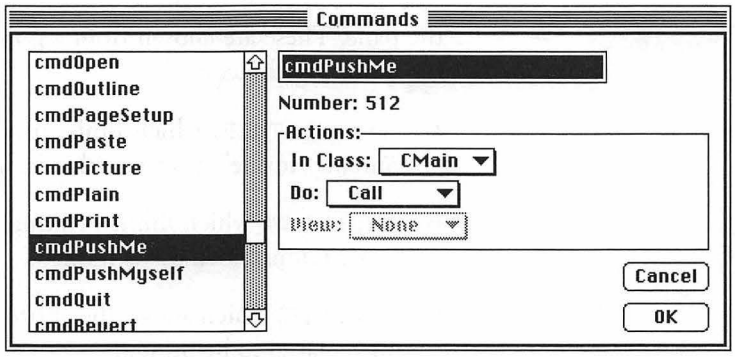

Figure 6-3
New cmdPushMe command created

- The properties of the CControl class include a title (`cntrlTitle`), current control value (`cntrlValue`), minimum control value (`cntrlMin`), and maximum control value (`cntrlMax`) for the control. Push buttons can have only two possible values: 0 and 1, for any of these values. The default current value is always 0.

- The properties for the CPane class include the `width` and `height` as well as the left (`hEncl`) and top (`vEncl`) positions of the pane in its enclosure. In addition, the horizontal and vertical sizing characteristics of the pane when its enclosure is resized are specified by pop-up menus that offer the following choices:

  - Horizontal properties of `sizFIXEDLEFT`, `sizFIXEDRIGHT`, `sizFIXEDSTICKY`, and `sizELASTIC`, which specify that when the enclosure is resized, the pane's left edge should remain fixed in place, its right edge should remain fixed, it should be firmly fixed both left and right, or that it can expand or shrink as needed.

  - Vertical properties of `sizFIXEDTOP`, `sizFIXEDBOTTOM`, `sizFIXEDSTICKY`, and `sizELASTIC`. Similar to the horizontal properties, but for the pane's vertical positioning when its enclosure is resized, these specify that the pane's top edge should remain fixed, its bottom edge should remain fixed, it should be firmly fixed both top and bottom, or that it can expand or shrink, as needed.

- The properties of CPane also include properties for printing the pane. These are chosen from a pop-up menu, which offers the following choices:
  - `clipAPERTURE`, which limits printing of the pane to the contents "inside" its frame (that is, its aperture)
  - `clipFRAME`, which limits printing of the pane to the contents circumscribed by its frame
  - `clipPAGE`, which allows the entire page on which the control is placed to be printed
- The CPane properties conclude with a checkbox titled `autoRefresh`. If this is checked, then the control will be redrawn whenever the TCL deems it appropriate to do so; otherwise, the application will be responsible for redrawing the control, at its discretion. The `autoRefresh` checkbox is usually checked for buttons.
- The CView class properties begin with a number of checkboxes that define the appearance of the view and its response to "click" events. These are as follows:
  - The `visible` checkbox indicates whether the view (button) is to be visible after it is created.
  - The `active` checkbox indicates whether the view (button) is enabled or disabled when it is visible. When checked, the button is enabled. A disabled button shows up drawn with a gray frame and gray text, rather than black. In this condition, the button will not accept any mouse clicks.
  - The `wantsClicks` checkbox indicates whether the view (button) is intended to accept mouse clicks and act upon them. Checking the checkbox allows a visible, active, button to receive clicks.
  - The `canBeGopher` checkbox allows the view (button) to become the current gopher if its BecomeGopher function is called. Generally speaking, this checkbox is unchecked for buttons.
  - The `usingLongCoord` checkbox indicates whether long coordinates (that is, 32-bit quantities) are to be used for

locations and sizes of the view. Generally speaking, this checkbox will be unchecked for buttons.

- The CView class also has a field with the view's `ID` and also its `helpResIndex`, with the latter offering the ability to attach balloon help to the view. If you create balloon help (by choosing **Balloon Help** from the VA's **Edit** menu, the VA will assign the `helpResIndex` value.

The foregoing property descriptions show the scope of the information that is associated with each button control that you create while inside the VA.

Button Actions

When the application is in execution and the user clicks a button, the actions taken depend upon various of the button's properties. If the button wasn't visible, then clicks in the location of where the button should be have no effect.

If a button is not visible, is not active, or its `wantsClicks` property is `FALSE`, then mouse clicks on the control will be ignored; otherwise, the following procedure is followed when a mouse click is detected:

1. The CSwitchboard DoMouseDown function is called. It calls the DispatchClick function of the CDesktop (gDesktop) object.

2. The CDesktop DispatchClick function determines that the click was in the "content" region of the window (where a button should be located) and then tests whether the window is currently active. If not, the window is selected, bringing it to the front.

   If the window properties specify that it does not want to process the click that causes it to activate, then processing of the mouse click ends at this point. However, if the (previously unselected) window wishes to process the click that causes it to become selected, then its Activate function is called.

   In the case where the window is already active or newly activated, its `wantsClicks` property is tested to determine whether it wishes to handle mouse clicks. If not, then the click is handled the same as a click on the desktop (that is,

nothing is done). If the window wishes to receive mouse click events, the Update function is called for each of the desktop's windows and then the DispatchClick function for the window in which the mouse click occurred is called.

3. The CWindow DispatchClick function (unless you have overridden this) converts the location of the mouse click (the `where` field of the event record) to local window coordinates and then calls the CView DispatchClick function with the updated event record.

4. The CView DispatchClick function calls its FindSubview function with the location of the click to locate a subview whose `wantsClicks` property is `TRUE`. If none is found that matches the criteria, the FindSubview function returns a value of `NULL`.

   If a "willing" subview is found, its DispatchClick function is called. Because none of the CPane, CControl, or CButton derived classes of CView handles the DispatchClick function, the process continues by the subview calling the CView DispatchClick function, which it has inherited, to locate one of its "willing" subviews.

   When the CView DispatchClick function returns a value of `NULL`, then it is assumed that the current view is the one in which the click occurred. You'll find that this is a valid assumption by considering the fact that the click occurred inside the window's content region and the final view to be handled by the DispatchClick function has no subviews—or at least none that is willing to handle a mouse click. In this case, the DispatchClick function continues by calling the DoClick function for the current view (or subview) object.

5. The DoClick function is handled by the CControl class when a CButton object is clicked. DoClick calls the Mac toolbox TestControl function to determine which part of the control was clicked. In the case of stationary controls such as buttons, the TestControl function returns a non-zero "part code" (that is, inButton) if the control is active; otherwise TestControl returns zero and the click is ignored.

   If the control is active, then the TrackControl function is called to track the mouse position until the button is released.

If it is released still within the control in which it was clicked, then the TrackControl function will return a non-zero value and the DoGoodClick function will be called. If TrackControl returns a zero value, then the user had a change of heart and didn't release the button while the cursor was inside the control in which it was clicked.

6. The DoGoodClick function is overridden by the CButton class to process a valid click on an active control that wants to receive clicks. This function tests whether a `clickCmd` has been assigned to the button. If so, it calls the DoCommand function of the button's supervisor (`itsSupervisor`) with the `clickCmd` code. If the `clickCmd` is equal to the value `cmdNULL`, then the click is ignored.

The final step in the foregoing illustrates how important it is to assign "click commands" to the buttons in your dialogs and windows. If you do not do so, then mouse clicks on these controls will be ignored. Notice in Figure 6-2 that a `Command` value of `cmdPushMe` has been assigned to the button. The behavior assigned to the occurrence of this command is shown in Figure 6-3, where it is specified that the VA should generate code to call a function in the CMain class when the button is clicked. This will result in the generation of a test for the `cmdPushMe` command code in the DoCommand function of the CMain class and the generation of a DoCmdPushMe function in the CMain class to handle the command.

Examining the CRadioControl Class

Radio buttons (and checkboxes) are more complex than simple push buttons. Both of those classes inherit behavior from both the CControl and CGroupButton classes, as shown in Figure 6-4.

The CRadioControl class supports the creation and management of radio button controls. CRadioControl is derived directly from the CButton class and shares its functionality to a large extent; however, radio buttons are unique in that they are usually created in groups and only one button in the group can be turned on at a time. This behavior is a metaphor for the buttons for selecting stations in a (somewhat dated) car radio—hence the name "radio button."

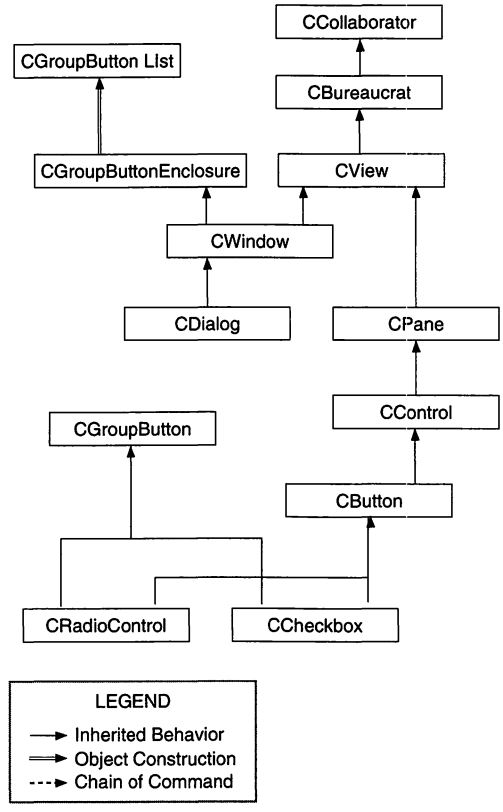

Figure 6-4
Inheritance diagram for CRadioControl and CCheckbox classes

Radio Button Properties

The "Pane Info" properties for a CRadioControl called "First" are shown in Figure 6-5. The CPane and CView class properties are the same as for the CButton object, whose properties are shown in Figure 6-2. Notice that there are no data members for the CRadioControl class, so the primary difference between the properties shown for CRadioControl and CButton are in the CControl class (where the "First" button's `contrlValue` setting is 1, indicating that it is turned on by default).

Another special property of CRadioControl is that it inherits behavior from the CGroupButton class, which is responsible for operations on the button that have relevance to the button's group. The VA assigns a group ID to each button, and you can group buttons either by placing them inside another enclosure or by selecting the set of buttons that comprise a group and then choosing the **Set Button Group** command from the VA's **Pane** menu.

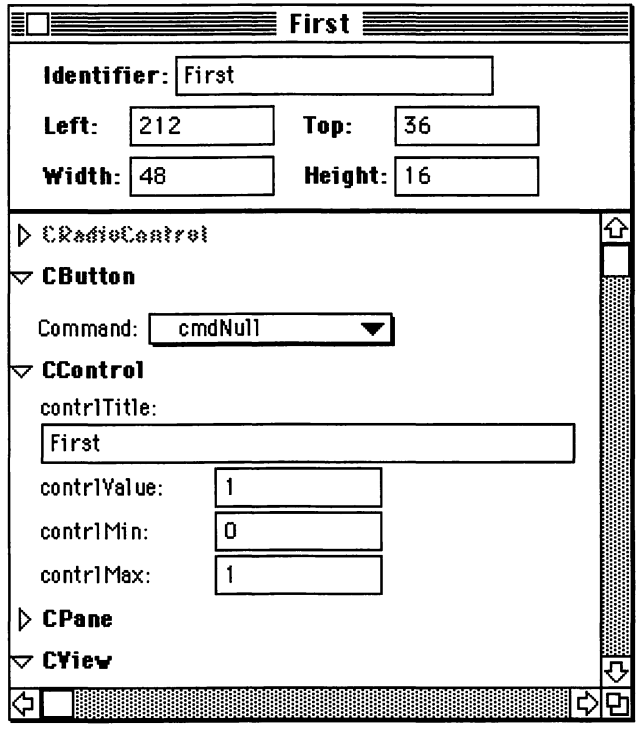

Figure 6-5
Pane Info settings for the First radio button

Radio Button Actions

Because radio buttons are derived from the CButton class as well as the CGroupButton class, the actions that are taken when a radio button is clicked are initially very similar to those for CButton objects. In fact, the main distinction from the description of "Button Actions" for the CButton class—albeit a large one—is in the override of the DoGoodClick function in the CRadioControl class (this was described for CButton on page 267). The DoGoodClick function in the CRadioControl class operates as follows:

1. The current state of the button is ascertained. If the button is currently "on," then no action will be taken and the click will be ignored.

2. If the button is currently turned "off," then the button is turned on by calling its SetValue function with `BUTTON_ON` as its argument.

3. The SetValue function sets the new control value into the radio button by calling the SetValue function of the CButton class to do so. If the new value for the control is the same as the old value (not possible with a radio button), then nothing further is done. However, if the new value is nonzero and it is different from the old value, the TellTurningOn function is called.

4. TellTurningOn is a member function of the CGroupButton class, which is multiply inherited by the CRadioControl class in the TCL. The CGroupButton class provides the specialized behavior for radio buttons, checkboxes, and other similar objects. When a CRadioControl object is constructed, its CGroupButton constructor initializer is executed.

   When the CGroupButton constructor is executed and an enclosure is specified for the button—this is always the case with VA-created radio buttons—the IGroupButton initialization function is executed. This function searches through the enclosure chain to find the window or dialog that encloses all of the buttons in that view and saves that pointer into the button's `itsGroupEnclosure` variable. When the window or dialog is located, a CGroupButtonList object is constructed (if it doesn't already exist), its pointer is saved in a member variable of the enclosure (`itsGroupButtons`), and then the newly created radio button object is added to the list.

   The TellTurningOn function calls the TurningOn function of the button's CGroupButtonEnclosure (`itsGroupEnclosure`) object.

5. The TurningOn function in the CGroupButtonEnclosure object iterates through the list of buttons in the `itsGroupButtons` list and calls the CGroupButton TurningOn function for each button in the list. If the button's group ID matches the current button's group ID and the button is not the current button, the button's TurnOff function is called. This ensures that only one button in a group is turned on at a time. If a group consists of checkbox controls, as well as radio buttons, then the code ensures that radio buttons turn off other radio buttons *and* checkboxes, but that checkboxes only turn off radio buttons.

Examining the CCheckBox Class

Checkbox objects of class CCheckbox are similar both to CRadioControl and CButton objects in the following ways:

- The objects dispatch a command (if one is associated with the object) when the checkbox is checked or unchecked. This is the primary behavior of a CButton object.
- CCheckbox objects inherit the features of the CGroupButton class and can cause radio buttons in the same group to turn off when the checkbox is checked. This is similar to how the CRadioControl object functions.

As the foregoing indicates, a checkbox is somewhat of a cross between a radio button and a push button, at least as far as its behavior is concerned. Its properties are basically the same as that of a CRadioControl object—both of which are derived from the CButton and CGroupButton classes. The main difference between checkbox and radio button controls is that multiple checkbox controls in a group can be simultaneously checked.

CCheckBox Properties

The properties for the CCheckbox class are identical to those for the CRadioControl class. If you substitute the name CCheckbox for CRadioControl in Figure 6-5, you will see the same properties when looking at the object's Pane Info inside the VA.

CCheckBox Actions

The major distinction between CCheckBox and CRadioControl objects is in their DoGoodClick functions. The DoGoodClick function for the CCheckbox object—called when the user's mouse-down and mouse-up actions for a checkbox are both inside the control's bounds—performs the following actions:

1. The state of the checkbox's value is toggled between values of 0 and 1 by calling SetValue for the checkbox.
2. If a command is associated with the checkbox, then the DoCommand function for the current gopher is called with the command code specified for the checkbox.
3. If the group ID for the checkbox is nonzero and the checkbox has just been checked, then the TellTurningOn function will

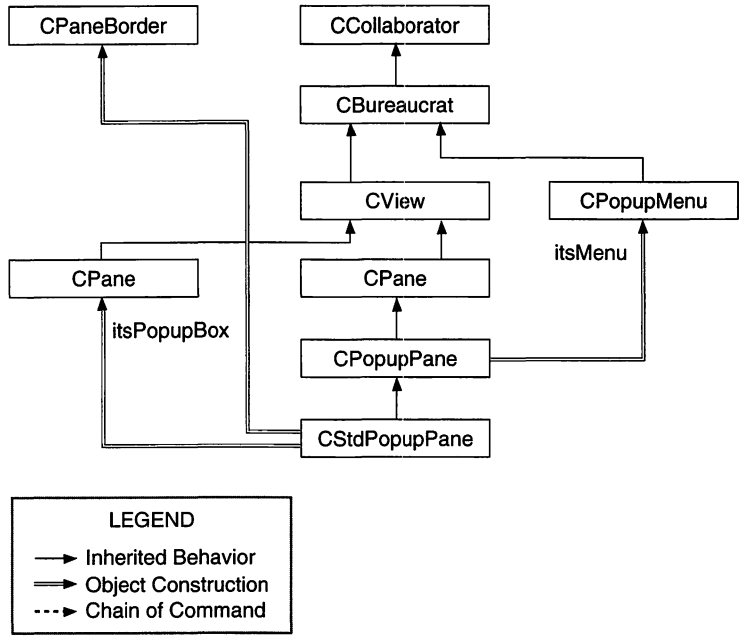

Figure 6-6
Pop-up Menu object construction and hierarchy

be called. As with the CRadioControl object, this function searches for any radio controls in the same group as the checkbox and turns them off. Therefore, when a checkbox in a mixed group of radio buttons and checkboxes is checked, the radio buttons in that same group will be turned off. Clicking one of those same radio buttons will not, however, uncheck any of the checkboxes.

Learning About Pop-Up Menus

While pop-up menus implement the functionality of menus within a window or dialog, rather than the menu bar, they are treated more like controls by those who develop and use them. The VA creates a CStdPopupPane object when you specify a pop-up menu in a window or dialog. This object consists of a small pane to display the current menu choice (`itsPopupBox`), with a border around the pane that is created by the CPaneBorder class, and is installed into the pane as `itsBorder`. The object contains a CPopupMenu object that contains the menu item entries and provides the "pull-down" menu behavior. The object relationships and hierarchy are shown in Figure 6-6.

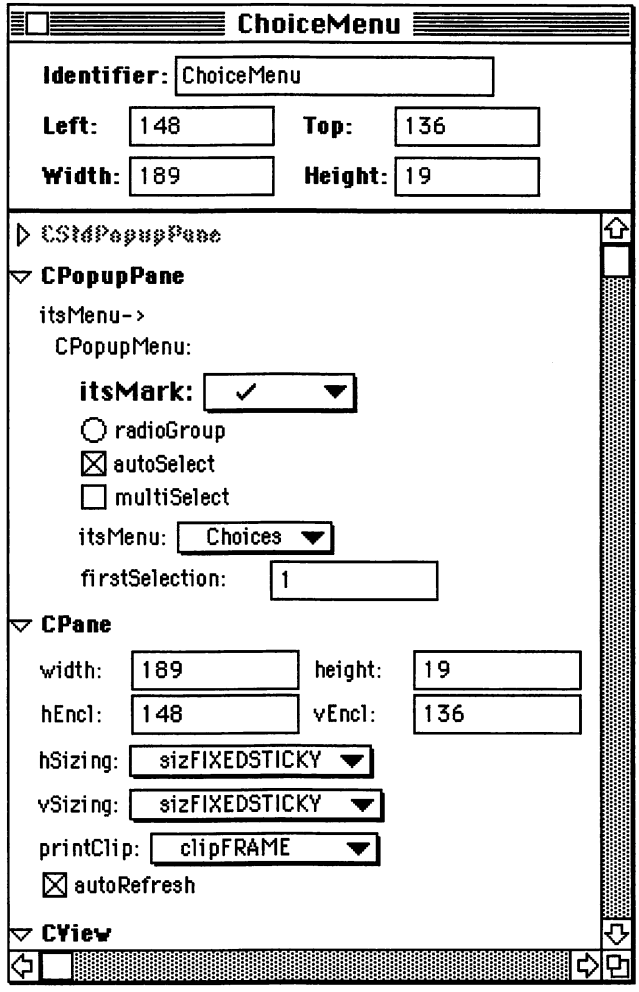

Figure 6-7
ChoiceMenu pop-up menu settings

Pop-up Menu Properties

The VA Pane Info properties for an example pop-up menu called ChoiceMenu are shown in Figure 6-7. These indicate that the control has an associated menu, whose name is **Choices**. When a menu item is chosen, it is marked with the symbol chosen from the `itsMark` pop-up menu, and the choice is displayed in the bordered pane. The Choices menu is specified to have the `autoSelect` property, which causes a previous selection to be unmarked and the newly chosen selection to be marked, automatically. The `radioGroup` property allows only one menu

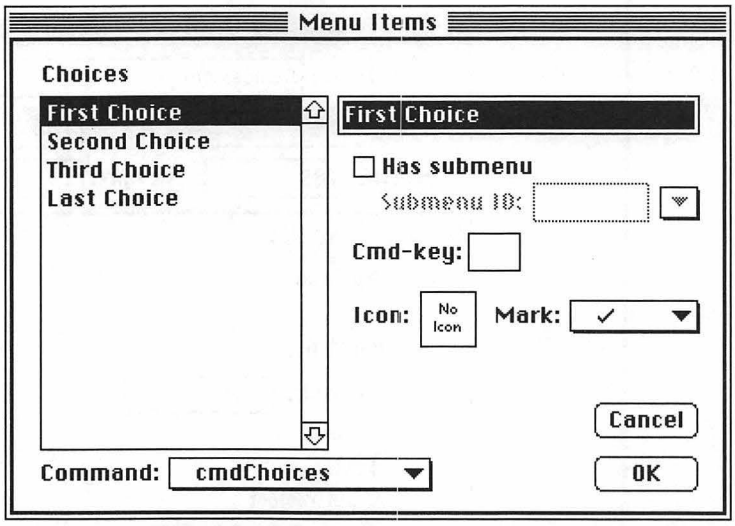

Figure 6-8
Choices menu command items

item to be checked at a time. In this case, with the `autoSelect` property, only one item can be chosen; if we had also checked the `multiSelect` property, then multiple items could be chosen. The CPane properties are as shown.

The **Choices** menu items are illustrated in Figure 6-8. Each of the choices has an associated command that specifies the code to be used when the command is dispatched. The command code (`cmdChoices`) is the same for all of the menu commands in this example, and the behavior of the `cmdChoices` command is for the DoCommand function in the CMain class to call a function when any of the items is chosen.

As is often the case with pop-up menus, we have defined each of the menu's entries when we defined it in the VA. It is also possible to populate a pop-up menu during execution of the program, but that technique is not used too frequently. If you wish to do so, you can access the menu handle and use the AppendMenu toolbox call to add items to the menu. The menu handle is stored in the CPopupMenu object as a protected member variable, so you must use the GetMacMenu member function of that class to access the handle.

When the CStdPopupMenu object is constructed, its display pane (`itsPopupBox`) and menu (`itsMenu`) objects are also constructed and initialized.

Pop-up Menu Actions

When the user clicks on the pop-up menu pane, the pane highlights and the menu pulls down to displays either the full set or a partial set of item choices. In the case where there are many choices, an arrow is drawn at the bottom of the list to indicate that more choices are available. The pop-up menu shown in Figure 6-8 has only four choices (First Choice, Second Choice, Third Choice, and Last Choice). When a choice is made, its associated command (if any) is dispatched and the choice's text is displayed in the pop-up pane. The detailed actions are as follows:

- The act of clicking on a pop-up menu pane causes the CStdPopupPane's DoClick function to be called. This function verifies that the click occurred inside the pane; if so, it inverts the title in the pane and calls the CPopupPane's DoClick function.

- The DoClick function in the CPopupPane class calls the PopupSelect function for the CPopupMenu object associated with the pane. The last argument of the PopupSelect function is the command code of the choice to which the menu should scroll when it is first displayed. In this case, the argument is a call to the CalcPopupCmd function, which returns a value of `cmdNULL`, by default. You could override this to choose another command code if you create a pop-up menu that is derived from the CStdPopupPane class.

- The PopupSelect function of the CPopupMenu class determines which entry to display as the initial selected item in the menu, either by searching the menu for a choice whose command matches the last argument to the function call, or by arbitrarily selecting the first item. The function then inserts the menu into the application's menu list and calls the PopUpMenuSelect toolbox function to handle the selection process.

- When PopUpMenuSelect returns to the PopupSelect function, the menu is removed from the menu list and then the result of the selection action is tested. If no item was selected, then `cmdNULL` is returned to the DoClick function in the CPopupPane class; however, if a choice was made and its associated command code is not `cmdNull`, the DoCommand function for the menu's supervisor (`itsSupervisor`) is called, and then one of two possible actions is taken, as follows:

- If the `autoSelect` property for the menu is not specified, then the `firstSelection` member variable is set to the selected item and the BroadcastChange function is called to report the `popupMenuNewSelection` semantic event. A pointer to the selected item is passed in the call.

- If the `autoSelect` property is specified, then the SelectItem function of the CPopupMenu class is called to handle the change in the selection, automatically. The action argument to the SelectItem function is `pmToggle`, which toggles the state of the mark for the selected item. While inside the SelectItem function, the `multiSelect` property is tested. If that property is specified, and if the `pmToggle` action type is specified, the mark for the selected item is toggled; otherwise, the selected mark is attached, or not, depending upon whether `pmForceOn` is the specified action type. If the `multiSelect` property is not specified and the action is `pmForceOff` *or* the new selection is the same as the previous selection, then the SelectItem function returns to PopupSelect; otherwise, if there is an existing selection, its mark is removed, the selected mark is attached to the newly selected item, and, finally, BroadcastChange is called with the `popupMenuNewSelection` semantic event and a pointer to the newly selected item number. SelectItem then returns to PopupSelect.

- When PopupSelect receives control once again, it returns the command code associated with the newly selected item to the DoClick function of the CPopupPane object, which ignores the returned value, but returns control to the DoClick function of the CStdPopupPane class. That function reinverts the popup menu's title, returning it to its normal appearance. Processing of the mouse click is complete at this point.

There is ample opportunity to customize the behavior of the popup menu-related functions, but you must do so by creating derived classes of the appropriate CStdPopupPane or CPopupMenu base classes in order to override any of their functions. Customization of some of the actions can be performed by carefully choosing the action types in the VA's Pane Info properties, or by calling the SetAutoSelect, SetMultiSelect, SetRadioStyle or SetMarkChar functions of the CPopupMenu class (`itsMenu`).

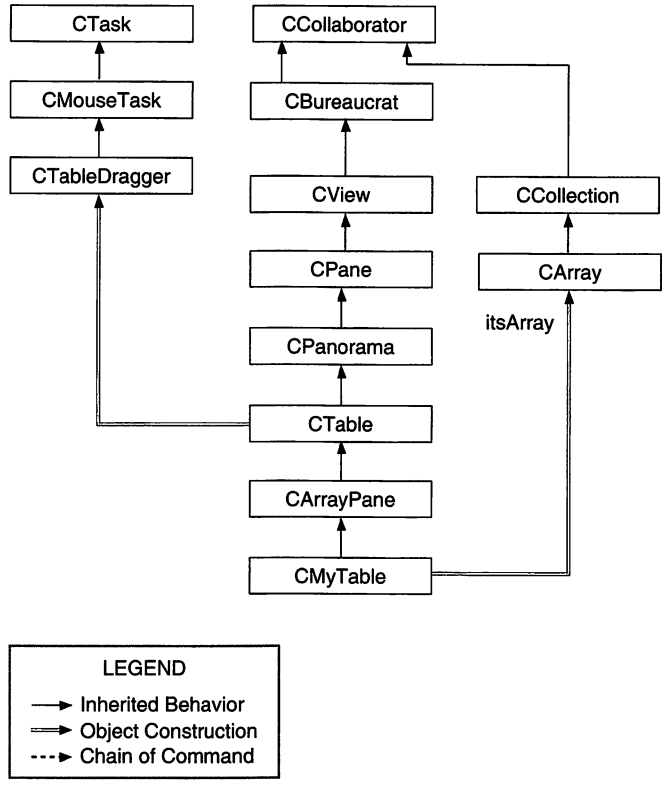

Figure 6-9
VA-created table class hierarchy and actions

Learning About Tables

The CTable class in the TCL makes provision for the creation of one- or two-dimensional tables, with a myriad of options. The CArrayPane class is derived from CTable to provide support for an associated array (if desired) that will hold the data for table objects. The class hierarchy for VA-created tables is shown in Figure 6-9. The CTableDragger object is created as a task to handle the selection criteria for table entries.

The CTable class has no storage for the data contained in a table's cells. It relies upon the GetCellText function to provide this data, and you *must* override either GetCellText or DrawCell to implement tables in your applications.

The VA-created table objects are all derived from the CArrayPane class, mainly because it is most often the case that you'll want an array to hold the table's data. Even so, to create a table, you must

first create a new class (choose **Classes** in the **Edit** menu), and specify that it is to be derived from CArrayPane. After doing so, you can create a table using the VA's List/Table tool, and set its class to the one that you created previously, by selecting the table object and then choosing your new class's name from the **Class** submenu of the VA's **Pane** menu. This is shown as CMyTable in the class hierarchy of Figure 6-9.

One of the most useful aspects of the CTable and CArrayPane classes is the variety of semantic messages that they generate. The CArray (or other CArray-derived) class, when an array is created and specified as a provider to the dependent table, calls BroadcastChange for each insertion, deletion, or change to an element of the array. In addition, the CTable class calls BroadcastChange when the user makes a selection in the table. These semantic messages can be easily processed in the window director's ProviderChanged function.

Table Properties

Lists and tables (both constructed from the CArrayPane class by the VA) can have up to 32,768 rows and 32,768 columns. A short (16-bit) integer is used to hold each of these values. Unlike the List Manager in the Macintosh toolbox, the CTable class supports the full range of columns and rows. However, because the maximum size would comprise more than a billion cells, the only practical limit to a table's size is the amount of memory you wish to devote to contain its data. A robust design could even access the table's data from disk and be able to support only the amount that could fit into memory at a given time; however, that technique is beyond the scope of this book.

Now that you're convinced that tables or lists can grow to any reasonable size, it is important to point out the other properties of these very special "controls."

I have created a new class called CMyTable, derived from CArrayPane, and have set its properties in the VA, as shown in Figure 6-10. As you can see from the figure, neither the CMyTable nor CArrayPane has any properties that you can specify. The properties of the CTable class are quite extensive, however. The various properties of the CTable class are as follows:

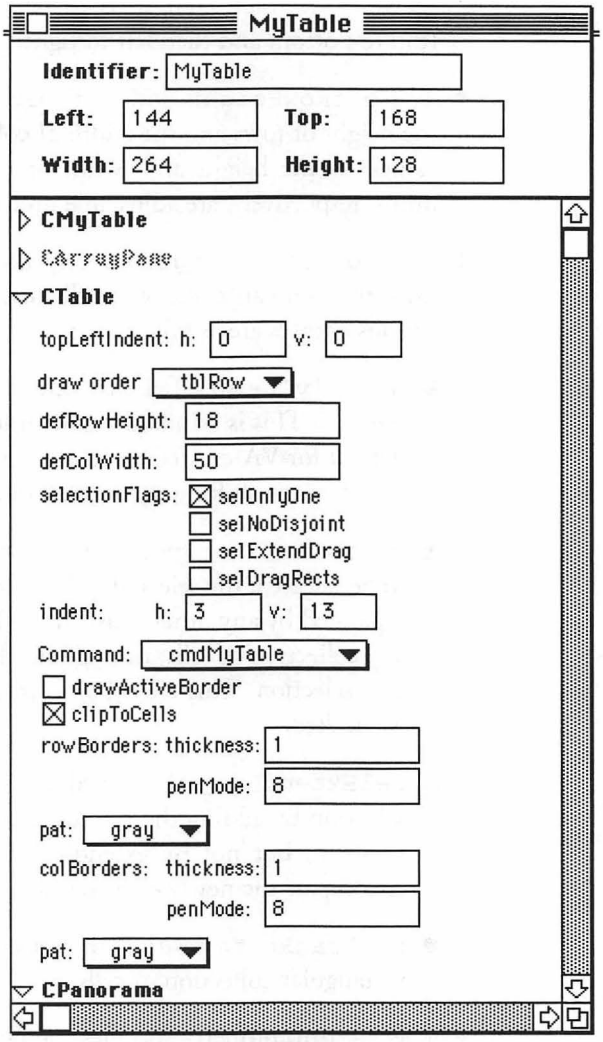

Figure 6-10
CMyTable properties, as shown in the VA's Pane Info

- The `topLeftIndent` property specifies the horizontal and vertical displacement of the entire table from its enclosure. As is usual for such values, positive values indicate horizontal indentation to the right and vertical indentation from the top of the enclosure.
- The `drawOrder` can be specified as `tblRow` or `tblCol`; the former specifies that each row is drawn before any cell in the next row (left to right and then top to bottom), and the latter

specifies that each column is drawn before the next column (top to bottom and then left to right).

- The `defRowHeight` and `defColWidth` properties specify the height of rows and the width of columns. These are default values, as the height and width of individual rows and columns, respectively, are adjustable under program control.

- The `selectionFlags` property governs the type of selections the user can make when clicking on the table's cells. The various choices are as follows:

  - `selOnlyOne` specifies that only one cell at a time can be selected. This is standard behavior for a list, but it is not the default for VA-created tables. Make sure that you check this box if you wish for only one cell to be selected.

  - `selNoDisjoint` specifies that if multiple cells are allowed to be selected, the selected cells *must* be adjacent (that is, not separated by any other cells). If the box is not checked, then any collection of cells, throughout the table, can be included in a selection. This checkbox is ineffective if `selOnlyOne` is checked.

  - `selExtendDrag`, if checked, allows the user to extend the selection by adding the cell just clicked, when the Shift key is down, but not by extending the selection rectangle to encompass the newly clicked cell.

  - `selDragRects` restricts the selection to a single cell or a rectangular collection of cells.

- The `indent` property specifies the horizontal (`H`) and vertical (`V`) indentation of each cell's contents. This is a default value and assumes that text is to be written into the cell. In that case, the indent specifies the indent for the leftmost character's baseline in the cell. For example, using a cell height of 18 pixels, the 13-pixel vertical indent shown in Figure 6-10 specifies that the baseline of the text is 5 pixels above the bottom of the cell, leaving room for the descenders in the text. The indent values are used primarily by the DrawCell function. If you override this function to draw the contents of the table's cells yourself, then you can safely ignore the value of the `indent` property and draw whatever you wish inside the table's cells. Each cell

can have its own font, size, style, justification, or any other non-text contents that you wish, if you override DrawCell.

- The `Command` property is a double-click command that you can assign when you design the table inside the VA. I have defined a command called `cmdMyTable`, whose behavior is to call a function in the CMain class when the user double-clicks a cell in the table. When the user performs that action, the DoCommand function in the CMain class is called with the `cmdMyTable` command. The VA-generated code dispatches that command to a function called DoCmdMyTable, which it has generated into the x_CMain class's source code.

- The `drawActiveBorder` property, when checked, specifies that the border of the table is to be drawn when the table is the current gopher. This behavior is standard for Standard File lists and the like, but is not part of the Apple *User Interface Guidelines* for normal tables. Check or uncheck this property as you wish.

- The `clipToCells` property, if checked, ensures that when text is drawn (using the CTable's DrawCell function) it does not exceed the boundaries of the cell being drawn. If the property is not checked, then DrawCell will draw beyond the boundaries of the cell.

- The `RowBorders` property specifies the `thickness` (in pixels), the `penMode, and the pen pattern` to be used in drawing row borders, if they are to be drawn. The values shown in Figure 6-10 indicate a row border thickness of 1 pixel, a pen mode of `patCopy (8)`, and a pen pattern of `gray`. The various pen modes and their associated values are (`patCopy=8`, `patOr=9`, `patXor=10`, `patBic=11`, `notPatCopy=12`, `notPatOr=13`, `notPatXor=14`, `notPatBic=15`). The pen patterns are chosen from a pop-up menu and the only choices are gray or black. If you wish to use some other pattern for the borders, you can call the SetRowBorders function in the CTable class.

- The `ColBorders` property is specified in the same way as the RowBorders property. To use a custom pen pattern for the border, you can call the SetColBorders function in the CTable

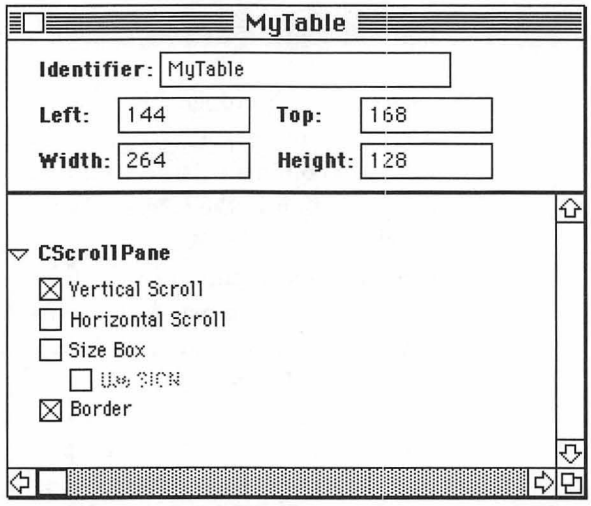

Figure 6-11
ScrollPane Info for the CMyTable object

class. The table in Figure 6-10 has a column border thickness of 1 pixel, a pen mode of `patCopy` and a pattern of gray.

In addition to the foregoing properties, a CArrayPane-derived object is enclosed by a CScrollPane, whose **ScrollPane Info** (chosen from the **Pane** menu when the table object is selected in the VA) are shown in Figure 6-11. In the case of the MyTable object, the table's enclosure has a vertical scroll bar and a border only.

Tables (and lists) also have properties for their CPanorama, CPane, and CView base classes. These are shown for the MyTable object in Figure 6-12. Although fairly extensive, the properties are fairly self explanatory.

Even though quite a number of the table properties can be specified inside the VA, there are others that must be set programmatically. The following paragraphs should help in this regard:

- Tables are created with a single column and no rows (that is, no table entries). If you want to increase the number of columns, then call the AddCol function. A very large positive value for the `afterCols` argument will ensure that columns are added *after* the last column, whereas a negative value will ensure that they are added *before* the first column.

- To add rows to the table, call the AddRow function and specify a large value for the `afterRow` argument to add the rows *after*

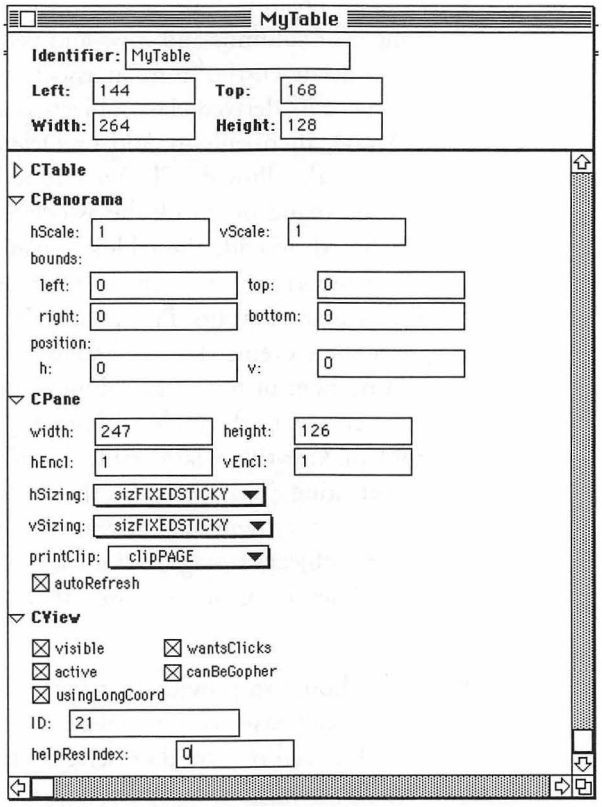

Figure 6-12
Base class properties of the MyTable object

the existing row (if any), or a negative value to add the rows *before* the existing rows.

- If you wish to change the default column width or row height, you *must* first delete the single column that is created by the VA. To do so, call the DeleteCol function and specify 1 as the value for the `numCols` argument and 0 as the value for the `startCol` argument. Table columns and rows are numbered starting with 0, so you must always translate 1-based index values in an array, for example, to 0-based table-cell numbering. After deleting the single column, you can call SetDefaults and specify the default column width and row height values.

- To change the height of a single row, or the width of a single column, call the SetRowHeight or SetColWidth functions with the appropriate arguments (remembering that rows and columns are numbered commencing with 0).

- If you are creating a spreadsheet, you may wish to create a fixed number of columns and rows and yet still retain the ability to add or delete entries from an associated array. When a CArray (or one of its derived class) objects is attached to a table (with SetArray), then calls to Add or DeleteItem for the array will result in the ProviderChanged function of the CArrayPane class to expand or shrink the associated table. To prevent this, you should override the table's ProviderChanged function and handle the `arrayInsertElement` and `arrayDeleteElement` semantic events. Proper handling in this case is to simply ignore these events. This will allow the table itself to retain the fixed number of rows and columns, but allow you to add and delete entries to the table without any expanding or shrinking problem. Creating a large table doesn't take any more memory than creating a small one. It's the *contents* of the cells that adds to the memory requirements, and these contents are not part of the table object's storage, but must be supplied by your application, either by using an associated array, or by some other means.

- If you choose to provide text font, size, style, and alignment on a per-cell basis, you may wish to store those settings in the individual cell objects themselves. Then an approach to handling the rendering of the cell's data is to override the DrawCell function for the table. In that function, you should first save the existing text properties, set the new properties and draw the text, and then reset the properties to their original values before exiting. In this way, each cell can have its own characteristics.

- For spreadsheet designs, you should create CArrayPane-derived objects for the column title and row title lists. To enable these to scroll in synchronism when the table is scrolled, override the table's Scroll function (inherited from CPanorama), and test whether a horizontal or vertical (or both) scroll was done. Call the Scroll function for the associated column title or row title lists, and then call the Scroll function for the CTable object. The code for this will look something like the following:

```
void MyTable::Scroll (long hDelta, long vDelta, Boolean redraw)
{
   if (hDelta > 0)
   {
```

```
      itsColTitles->Scroll (hDelta, 0, TRUE);
   }
   if (vDelta > 0)
   {
      itsRowTitles->Scroll (0, vDelta, TRUE);
   }
   CTable::Scroll(hDelta, vDelta, redraw);
}
```

The foregoing code is adaptable to any situation where you want to synchronize the scrolling of two or more lists or tables. The object pointers `itsColTitles` and `itsRowTitles` are assumed to be known to the table being scrolled. You can create an access function in your derived table class to set these pointers when the function is called from the director of all three of the objects.

Table Actions

When we speak of tables, we are speaking of CArrayPane-derived objects, because that is what the VA creates. In most cases, you will want to create a CArray (or CArray-derived) object to contain the data to be displayed in a table or list. It is *not* mandatory that you assign an array to a table created as a CArrayPane-derived object, but it is often convenient to do so.

You can create the array at any time. It can be a CArray or an object in any of the CArray-derived classes. The array should not be attached until the table has been created (in an override of MakeNewWindow, for example). Calling the SetArray function of the CArrayPane-derived object with a pointer to the array object and an "ownership" flag will "attach" the array to the table. If the ownership flag is `TRUE`, then the table "owns" the array and it will be disposed when the table is disposed. On the other hand, if the ownership flag is `FALSE`, then you will be responsible for deleting the array and its contents. Generally speaking, you should set the flag to `FALSE` if the table contains other dynamically allocated objects, because although the array itself is disposed (if it is "owned") when the table is disposed, any objects contained in the array are not disposed. If the array contains values or strings that have not been dynamically allocated, setting the ownership flag to `TRUE` provides a truly carefree combination.

If you refer to the code for Category list that we presented in Chapter 5, you will see that we created the array in the ICMain function of the CMain class (see page 218) and then called SetArray in the BeginData function of the CCategories class, after the

MakeNewWindow function had been called to create the array (see page 229). In this case, the document "owns" the array and the CCategories modeless dialog just "uses" the data it contains, so the ownership flag in the SetArray call is `FALSE`. The data in this case are pointers to objects in the CCat class, sorted alphabetically by category name.

When the user selects a cell (entry) in a table, the following actions are performed:

- The CTable class overrides the DoClick function of the CView class to perform the following actions:
  - The point at which the mouse click occurred is tested to determine whether it is within the bounds of the existing table cells. If not, then the ClickOutsideBounds function is called. That function calls DeselectAll, which clears any existing selections by calling DeselectRect for the entire redraw rectangle of the table. This results in BroadcastChange being called with a semantic event type of `tableSelectionChanged`.
  - If the click was within the bounds of the existing table cells, then DoClick tests whether the click is a double-click. If so, then the DoDblClick function is called. That function tests whether a double-click command is assigned to this event; if so, it calls the DoCommand function of the current object (the table). The CTable class override for the DoCommand function handles only the `cmdSelectAll` command, passing all other commands to the CPanorama (base class) to handle. Of course, commands travel up the hierarchy and the `dblClickCmd` command will be passed on to the table's supervisor (`itsSupervisor`) by the actions of the DoCommand function in the CBureaucrat class. You can either override DoCommand in your own CArrayPane-derived class or test for the `dblClickCmd` in your window or dialog director's DoCommand function.
  - If the click was not a double-click, then the DoClick function in the CTable class creates a CTableDragger object to track the mouse. The `mouseTask` object is passed to the TrackMouse function inherited from the CPane class. The TrackMouse function calls the BeginTracking function of

the `mouseTask` object, and then while the mouse button is still down, it calls the KeepTracking function for that object. When the mouse button is released, TrackMouse calls the EndTracking function of the `mouseTask` object.

- The BeginTracking function of CTableDragger determines the initial selection criteria to be associated with the mouse click, taking into account any modifier keys pressed on the keyboard at the time the mouse button was clicked. There are some rather complex criteria for determining what to do when a cell is clicked. If the Command key is down and the cell was selected already, then the DeselectCell function of the CTable class is called, resulting in the BroadcastChange function being called with a semantic event type of `tableSelectionChanged`. If the Command key is down and the cell was not already selected, then the SelectCell function of the CTable class is called to select the cell. This results in BroadcastChange being called with the semantic event code of `tableSelectionChanged`. If the Shift key is down, the Command key is not down, the table's `selDragRects` property is checked, and the `selOnlyOne` property is unchecked, then the newly clicked cell is added to the selection range, and the SelectRect function of the table object is called to select the cell and call BroadcastChange with the semantic event type of `tableSelectionChanged`. If neither the Shift nor Command keys is down, then SelectCell is called to select the table cell, and BroadcastChange is called with the `tableSelectionChanged` semantic event type.

- The KeepTracking function of CTableDragger is called continuously while the mouse button is still down. It determines whether to select or deselect cells, depending upon the state of the modifier keys and the table properties, as the user drags the mouse. Newly added selections or newly deselected cells result in BroadcastChange being called with the `tableSelectionChanged` semantic event type.

- The EndTracking function of CTableDragger is called when the mouse button is released. This function merely disposes of the current CTableDragger object (`this`) and returns.

- After the TrackMouse function returns control to the table's DoClick function, that function concludes execution.

In addition to the actions performed with the mouse and keyboard modifier keys, other actions can cause semantic events to be dispatched. These are all associated with operations on the array that is attached to the table object, and for which a dependency relationship exists. The following semantic events are created by the CArray class:

- When the array's destructor is executed, BroadcastChange is called with the semantic event type of `arrayGoingAway`.
- When you call SetArrayItem, BroadcastChange is called to report the `arrayElementChanged` semantic event.
- When you call InsertAtIndex, BroadcastChange is called to report the `arrayInsertElement` semantic event.
- When you call DeleteItem, BroadcastChange is called to report the `arrayDeleteElement` semantic event.
- When you call MoveItemToIndex, BroadcastChange is called to report the `arrayMoveElement` semantic event.
- When you call the Swap function, BroadcastChange is called twice with the `arrayElementChanged` semantic event; once for each of the elements being swapped.

You can override the ProviderChanged function in your derived table class, or you can override that function in your director class (for example, CMain) to handle any one or more of the foregoing events (including the ones dispatched by the selection and deselection functions in the CTable class). ProviderChanged is called with a pointer to the provider (the table object or the array), the event type code, and any additional information (for example, an array item index).

If you are implementing a spreadsheet-like object, you may wish to handle the `arrayInsertElement` and `arrayDeleteElement` events by ignoring them, as described on page 284.

If your ProviderChanged function is called for a table cell selection or deselecting event, you will have to determine the meaning of the event by calling GetSelection to ascertain what cells (if any) are selected. If you have specified the `selOnlyOne` property,

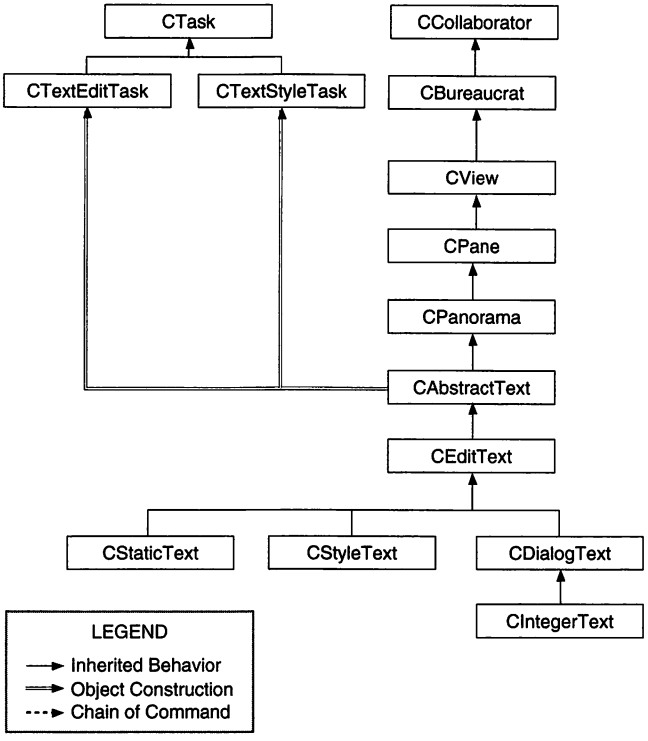

Figure 6-13
Object hierarchy of text object "controls"

then it is very easy to determine which cell (if any) has just been selected by calling the GetSelect function. In the case where you allow extended selections or disjoint selections, you will have to keep track of the selection region and compare the cells it contains to which were selected previously, if you wish to use these semantic events. The GetSelection function returns a region handle (`RgnHandle`) object.

Learning About Text

While text fields (either static or editable) are not specifically considered as "controls," per se, they can respond to various commands and create various semantic events when operated upon by the user. The object hierarchy and undoable text editing task objects for editable text objects are shown in Figure 6-13.

CDialogText and CIntegerText objects will occur only in dialogs, whereas CStaticText, CStyleText, and CEditText objects can occur in any window type. The VA does not support the creation of the CStyleText object at this time; however, you can create objects

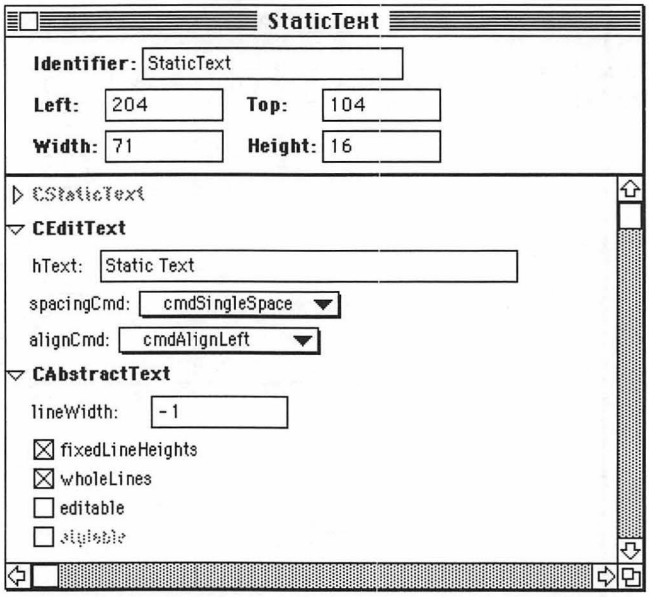

Figure 6-14
Static text object properties

of this type programmatically and use the features of the CStyleText class. All of the classes derived from the CEditText object use the Mac toolbox's TextEdit features and are subject to its limitations, as well. If you wish to create a robust text editor application, you will need to derive your custom text object from the CAbstractText class and then add all of the functionality that you require. The CAbstractText class has only a limited number of features you can use for this task, although it does support the creation of tasks to handle cut and paste, as well as text styling.

Text Properties

The VA-related properties of a static text object are shown in Figure 6-14. Note that although the CStaticText class has no explicit properties, the `editable` and `styleable` properties of the CAbstractText base class are unchecked and disabled, respectively.

A static text field usually includes the specification of a text string in the `hText` property of the CEditText class, as shown in the figure. Other properties of the static text field are as follows:

- The `spacingCmd` property has three choices in the pop-up menu. These are `cmdSingleSpace`, `cmd1HalfSpace`, and `cmdDoubleSpace`.

- The `alignCmd` property has four choices in the pop-up menu. These are `cmdAlignLeft`, `cmdAlignRight`, `cmdAlignCenter`, and `cmdAlignJustify`. While the last property might suggest that "full justification" is supported, this is not the case. The Mac toolbox does not support full justification and the choice is provided in case you wish to perform this operation in your derived class.

The CAbstractText class properties for a CStaticText (or any other derived class) are shown in Figure 6-14. These are as follows:

- The `lineWidth` property specifies the length of text lines in pixels. If the value is negative (as is shown), then the width is taken to be the width of the text pane.

- The `fixedLineHeights` property, if checked, ensures that all lines in the text field are the same height.

- The `wholeLines` property, if checked, ensures that only whole lines (relating to their height) are displayed when the field is scrolled or changed in size.

- The `editable` property applies only to CEditText or CStyleText fields, and not to CStaticText fields, for which this property is disabled when the object is initialized.

- The `styleable` property applies only to CEditText or CStyleText fields, and not to CStaticText fields, for which this property is disabled when the object is initialized.

Other properties for the CPanorama, CPane, and CView base classes for text fields are as shown in many of the other illustrations for these classes.

The properties for a CDialogText object are shown in Figure 6-15. This object is created for editable text fields in dialogs—one visible difference between these and CEditText objects is a border around the CDialogText field. Because dialog fields are meant to be validated, CDialogText fields have properties in addition to those shown for CEditText fields, as follows:

- The `maxValidLength` property specifies the maximum number of characters that the user can enter into the field. It is set to the size of a long (32-bit) variable, by default.

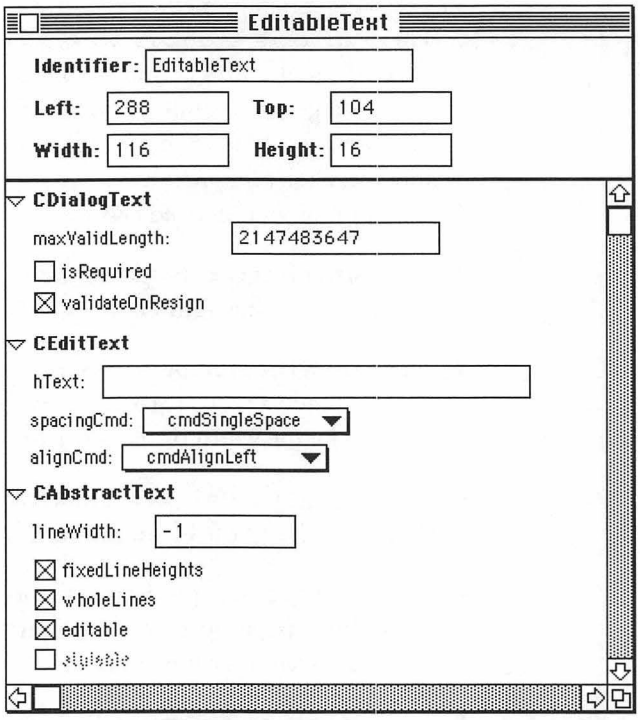

Figure 6-15
CDialogText field properties

- The `isRequired` property, if checked, specifies that the user *must* enter text into this field; otherwise, any attempt to move to another field or dismiss the dialog with the OK button will cause the TCL to display an alert, indicating that the field is a required input.
- The `validateOnResign` property determines whether validation is performed at the time the user dismisses the dialog with the OK button (no validation is performed if the dialog is cancelled, in any case). You can override the Validate function of your dialog's director (usually a CDialog-derived class) and perform your own validation of the user's inputs.

In addition to the foregoing properties, CDialogText objects also inherit all of the properties of their CEditText, CAbstractText, CPanorama, CPane, and CView base classes.

Text Actions

In addition to the normal actions associated with editable text (copying, cutting, and pasting), the DoCommand function of the

CAbstractText class supports the choice of a command from the application's **Font** (`MENUfont`) or **Size** (`MENUsize`) menus, if these exist. Both of these menus are provided in the VA's list of potential menus to add to the menu bar when you choose **Menu Bar** from the **Edit** menu. Just click on the **Add** pop-up menu at the right side of the Menu Bar dialog, and choose either **Font** or **Size** as the menu to add.

If either of the foregoing menu commands (or a command to change the text style) is sent to the DoCommand function of CAbstractText, and if the `editable` and `styleable` properties of the text field are `TRUE`, then it creates a CTextStyleTask object, calls the Notify function for the field's supervisor (usually a CDirector object) to set the "dirty" flag for the document (and intermediate directors), and then calls the Do function for the CTextStyleTask object. The Do function saves the current style settings, assigns the newly specified style, and then prepares for the user to Undo the action. If the CTextStyleTask's Undo function is called subsequent to the previous action, it swaps the new and previous style information and performs the Undo or Redo operation.

If any of the **Copy**, **Cut**, **Paste**, or **Clear** commands is chosen by the user for the text field, then the CAbstractText class creates a CTextEditTask object and calls its Do function to perform the requested action. The Do function of the CTextEditTask calls upon the text field's PerformEditCommand function to perform the actual requested action—interacting with the CClipboard object (`gClipboard`) for **Cut**, **Copy**, and **Paste** commands. The CEditText class contains the PerformEditCommand function. After the requested has been performed, ReportChange for the CTextEditTask is called—resulting in BroadcastChange being called with a semantic event type of `textValueChanged`.

If the chosen command was **Select All**, the DoCommand function of CAbstractText selects the entire contents of the text field and then calls SelectionChanged—resulting in BroadcastChange being called with a semantic event type of `textSelectionChanged`. In addition, if a "typing task" was created for the field, then its SelectionChanged function is called (typing tasks are associated with key events, which will be covered in a later chapter).

Table 6-1
Semantic event classes and code summary

Class	Type Code	Event
CAbstractText	textSelectionChanged	New selection made
CArray	arrayGoingAway	Destructor called
CArray	arrayElementChanged	Element is changed
CArray	arrayInsertElement	Element is inserted
CArray	arrayDeleteElement	Element is deleted
CArray	arrayMoveElement	Element is moved
CBureaucrat	bureaucratIsGopher	Newly assigned gopher
CBureaucrat	bureaucratIsNotGopher	Newly resigned gopher
CControl	controlValueChanged	Control value changed
CDialogText	dialogTextChanged	Text field has changed
CPopupMenu	popupMenuNewSelection	Command newly chosen
CRunArray	runArraySizeChanged	Array size changed
CRunArray	runArrayElementChanged	Element is changed
CTable	tableSelectionChanged	New selection made
CTextEditTask	textValueChanged	Text field has changed
CIconButton	controlValueChanged	Control value changed
CSwissArmyButton	controlValueChanged	Control value change

Any other command intercepted by the DoCommand function of the CAbstractText class is passed to its CPanorama base class to resolve.

Control Object Summary

This chapter has described a number of controls, their properties, and the semantic events that they create when operated by the user. I have talked about buttons, radio buttons, checkboxes, pop-up menus, tables, and text controls, how each of these is created within the VA, and how the properties of each of these can be defined or changed.

Because quite a number of semantic events are generated by a variety of classes in the TCL, I have summarized them for easier reference in Table 6-1.

Events other than commands and semantic events are handled by the normal event loop of the application. These events will be discussed in the next chapter.

Chapter 7
Handling Events

This chapter is all about how the TCL handles standard Macintosh events. I will describe how the "standard" events are dispatched from the main application's event loop and how they are handled. High-level events will be covered in a later chapter. Figure 7-1 illustrates the major components of the event mechanism in the TCL. It is supplemented by what is shown in Figure 7-2. In each of the two figures, individual functions are enclosed by rounded rectangles and function calls are indicated by directed lines from the source of the call to its destination. The thick vertical lines are used to eliminate a "rat's nest" of connections and are intended to show that calls to all of the destination functions are made from the single source. Classes are shown inside the dotted rectangles.

Examining the Main Event Loop

The main event loop in the TCL is contained in the CApplication class, in the DoRun function, as shown in Figure 7-1. That function calls the Process1Event function in the same class, which, in turn calls the ProcessEvent function in the CSwitchboard class. When control returns to the DoRun function, it loops, waiting until the value of the `running` variable becomes `FALSE` (which will occur when the user chooses the **Quit** command from the **File** menu), continuing to call Process1Event on each trip through its loop.

When a modal dialog is created and DoModalDialog is called, the DoChangeableModalDialog function in the CDialogDirector class also calls the Process1Event function, in a loop, waiting for the command to dismiss the dialog to occur.

So the focus of attention in the TCL's event mechanism is inside the Process1Event function. The steps taken to handle events in that function are as follows:

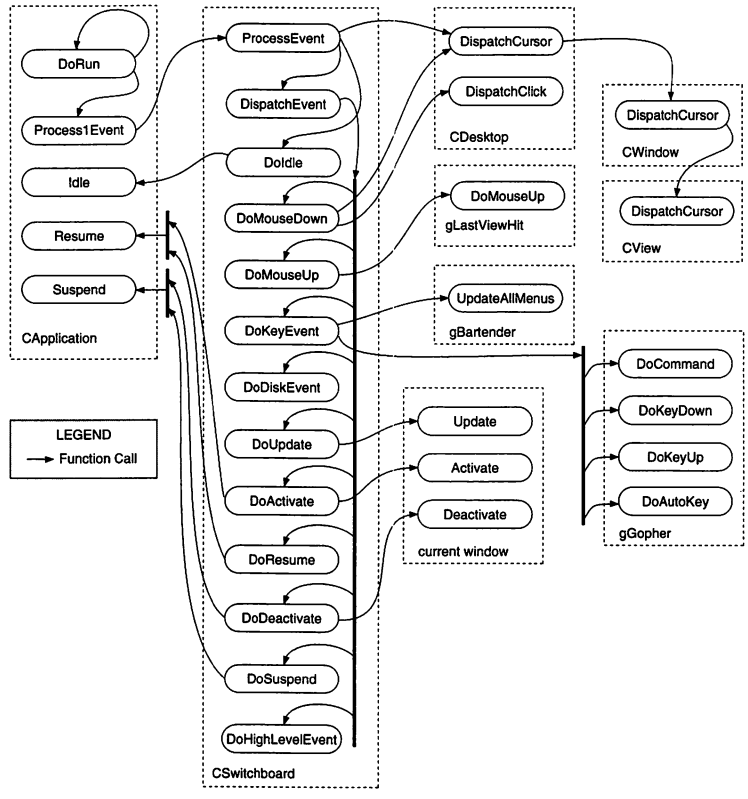

Figure 7-1
Major event-handling elements in the TCL

1. The ProcessEvent function in the CSwitchboard class is called.

2. If any urgent chores exist, each is executed, in turn. An urgent chore is entered into that list by calling the application object's AssignUrgentChore function. After the urgent chores (if any) are executed, they are removed from the list. An urgent chore is a one-shot task that must be executed as soon as possible, but only once.

3. If the front window is a "system" window (that is, for a desk accessory), then the Suspend function is called, allowing the DA to continue execution. If the front window is *not* a system window, and a desk accessory has just become inactive, then Resume is called.

It should be clear that no significant tasks were left out of this explanation, though no event has yet been processed by the

Process1Event function—that task is left to the ProcessEvent function of the CSwitchboard class.

The Process Event Function's Role

The ProcessEvent function in the CSwitchboard class is the function that calls the Mac's WaitNextEvent or GetNextEvent toolbox function to access the next event. Most modern Macintosh computers include the code for the WaitNextEvent function, which supersedes the functionality of the original GetNextEvent function. In any case, the end result of calling either function is the acquisition of a single new event (or no event). The sequence of steps taken by the ProcessEvent function is as follows:

1. The current mouse location is acquired and then the DispatchCursor function of the CDesktop class is called. That function, in turn, calls the DispatchCursor function of the CWindow class, which calls DispatchCursor for the CView class. If the view has any subviews, then DispatchCursor is called for the "hit" subview. This allows a subview to control the shape of the cursor on an event-by-event basis.

2. ProcessEvent then calls GetAnEvent, which in turn calls the appropriate toolbox function to get an event from the Mac's event queue. If the event code returned by the GetAnEvent function is nonzero, then DispatchEvent is called; otherwise, DoIdle is called.

When the foregoing steps are complete, the ProcessEvent function returns to the Process1Event function in the CApplication class, and that function returns to the loop in the application's DoRun function.

The DoIdle Function's Role

Let's talk about the case where no event was found in the Mac's event queue. In this case, the GetAnEvent function returns a zero value as the event code (`NULL` event). As mentioned previously, the ProcessEvent function calls the DoIdle function in the CSwitchboard class, which, in turn, calls the Idle function for the application object (`gApplication`). You can provide an Idle function in your application subclass to override the default be-

havior. The Idle function in the CApplication class performs the following actions:

1. If the "Rainy Day Fund" of additional memory, held by the application, has been used (`rainyDayUsed` is `TRUE`), then the Idle function attempts to replenish the used memory by allocating a new block. If the fund cannot be replenished, then the Idle function posts an alert, informing the user that memory is getting low.

2. Then the Idle function enters into a loop that calls the Dawdle function of each Bureaucrat in the chain of command, starting with the current gopher. Generally speaking, the current gopher is a pane in the frontmost window. That pane's Dawdle function (if any), its supervisor's Dawdle function, then the Dawdle function for the supervisor's supervisor, and so forth, are called, until the end of the chain of command is reached. You may choose to provide a Dawdle function for various objects in the chain of command, in order to take advantage of "idle" time, when the user has not performed any action that would cause an event to enter the queue.

3. After completing the loop which calls the various Dawdle functions, the Idle function calls a static function named DoChores to cycle through the list `itsIdleChores` and call the Perform function for each member of the list. Idle chores are entered into the list by calling the application object's AssignIdleChore function. They remain in the list and are called periodically, until they are removed by calling the application's CancelIdleChore function.

There is great merit, in certain circumstances, to using the Dawdle function or idle chores to manage a periodic process. For example, in a time-scheduling application, there is often a need to examine a list of reminders to ascertain whether it is time for the earliest reminder to be posted. Using the Dawdle function to handle a periodic task such as this is recommended highly.

If a periodic task must be performed for an object that is not directly in the chain of command, then posting an idle chore to call its Perform function is a good way to proceed. For example, if calculations are to be performed when the application is idle, then an

idle chore would be an appropriate mechanism to initiate the next cycle of processing.

The DispatchEvent Function's Role

If the result of calling the GetAnEvent function is `TRUE`, then an event has been removed from the Mac's event queue and is available for inspection in the `macEvent` record (an EventRecord structure). The primary feature of the DispatchEvent function is a `switch` statement, which determines from the type field of the record (`what`) what kind of event has been removed from the queue. As you can see from Figure 7-1, the DispatchEvent function calls quite a number of other functions, also located in the CSwitchboard class, depending upon the type of event being processed. Both the functions responsible for handling the various event types and the actions that they perform are described in the paragraphs that follow.

Handling Mouse Down Events

If the event type is determined to be `mouseDown`, then the DispatchEvent function calls the DoMouseDown function. The steps taken by the DoMouseDown function are as follows:

1. The DispatchCursor function of the CDesktop object (`gDesktop`) is called. That function calls the DispatchCursor function for the CWindow class, and then that function calls DispatchCursor for the CView class. As described earlier for the ProcessEvent function, DispatchCursor is then called for the subview that was "hit" by the mouse click (if any).

2. DoMouseDown also calls the DispatchClick function in the CDesktop object. That function performs quite a number of additional operations, as shown in Figure 7-2. We will cover these additional operations shortly.

3. The EventRecord is saved into the `gLastMouseDown` record, which is used to determine whether multiple `mouseDown` events (double-click events, for example) have occurred.

Handling Mouse Up Events

If the event type is determined to be `mouseUP`, then the DoMouseUp function is called. That function performs the following steps:

1. If the `mouseDown` event that preceded the `mouseUp` event occurred on the desktop or in the content region of a window, then the DoMouseUp function for the view that was last "hit" (`gLastViewHit`) is called.

2. The current event record is saved in the `gLastMouseUp` structure.

Handling Key Events

The `keyDown`, `keyUp`, and `autoKey` events are all handled by calling the DoKeyEvent function. Depending upon the nature of the keystroke, this function performs a number of different tasks, which are as follows:

1. If the event is `keyDown` and the Command key is also down, then the event is assumed to be a command shortcut and the following steps are taken:

   a. The UpdateAllMenus function of the CBartender object (`gBartender`) is called. That function loops through all of the menus, dimming or unchecking the commands according to the specifications for each menu, and then calls the UpdateMenus function associated with the current gopher.

   b. A `menuChoice` variable takes on the value returned by the MenuKey toolbox call, identifying the menu and to which item within that menu the command shortcut pertains. It is possible that no menu contains a command code that matches the shortcut keystroke.

   c. If a menu does contain a command that matches the keystroke, then the DoCommand function for the current gopher is called with the command code associated with that menu command.

   d. If no menu contains the matching keystroke, then the DoKeyDown function for the current gopher is called with the character, its keycode, and the event record. This permits the application to perform internal command dispatching, according to its own needs.

2. If the event is `keyDown` and the Command key is not also down, then the DoKeyEvent function tests the keystroke to

determine whether it is one of the F1–F4 function keys. If so, then one of the following steps is taken:

a. If the keystroke is F1, then the UpdateAllMenus function of the CBartender object (`gBartender`) is called, followed by calling the FindMenuItem function of the CBartender object, looking for a match for the `cmdUndo` command (Undo). If the command was found to pertain to an existing menu, then the menu title is highlighted. Whether or not the command is found to pertain to a menu, the DoCommand function for the current gopher is called with the `cmdUndo` command code, and then the highlight (if any) is removed from the menu title.

b. If the keystroke is F2, then the same steps as in the foregoing are taken, with the exception that the menus are searched for the `cmdCut` (Cut) command, and the DoCommand function for the current gopher is called with the `cmdCut` command code.

c. If the keystroke is F3, then the menus are searched for the `cmdCopy` (Copy) command, and DoCommand for the current gopher is called with the `cmdCopy` command code.

d. If the keycode is F4, then the menus are searched for the `cmdPaste` (Paste) command, and DoCommand for the current gopher is called with the `cmdPaste` command code.

3. If the event is `keyDown` and is none of the foregoing, then the DoKeyDown function of the current gopher is called with the character, its keycode, and the event record.

Handling Disk Inserted Events

If the event is `diskEvt`, then the user has inserted a disk and DoDiskEvent is called to test the result of the disk mount process. The Mac toolbox handles the event, but the TCL will display an alert if the mount failed.

Handling Update Events

If the event is `updateEvt`, then the Mac OS has determined that the contents of one or more windows needs to be updated (that is,

the update region is not empty), so it manufactures an update event, returning it as the next event in the call to GetNextEvent or WaitNextEvent. The update process is quite complex and its steps are as follows:

1. The DispatchEvent function commences processing the `updateEvt` by calling DoUpdate with the event record. The currently active window object is ascertained and its Update function is called. That function calls the toolbox BeginUpdate, DeviceLoop, and EndUpdate functions to update (draw the regions to be updated) the screen. The DeviceLoop toolbox function scans all of the active display devices, calling the specified drawing procedure (DoUpdateDraw) for each screen that intersects the drawing region (`visRgn` for the current port). The steps taken by the DoUpdateDraw function are as follows:

    a. First of all, DoUpdateDraw is a universal procedure pointer member variable of the CWindow class. The actual procedure used to perform the drawing action in the window class is sDoUpdateDraw, which is a static function of the CWindow class. That function calls the UpdateDraw function for the current window.

    b. The UpdateDraw function sets the clipping region to the full port rectangle and determines whether the user's computer has Color QuickDraw. If so, it calls GetForeColor, RGBForeColor, GetBackColor, and then it calls RGBBackColor.

    c. The update rectangle is set to the bounding box of the port's visible region, and then UpdateErase is called with that value. UpdateErase calls the EraseRect toolbox function to erase the update region, prior to causing it to be redrawn.

    d. After the update region has been erased, the Pane_Draw function for each of the window's subviews, in turn, is called.

2. The Pane_Draw function of the CPane class is called to draw the current pane to be updated. The steps that it takes are as follows:

a. The Pane_Draw function is called with a pointer to the pane to be drawn and also the update rectangle. The pane object is first tested to determine whether it is really visible (it could hidden). If it is not really visible, then the Pane_Draw function does nothing further for this subview.

b. Next the pane is tested to determine whether it has a border. If so, then the intersection of the border with the update region is computed and stored into the clipping rectangle by calling ClipRect. Then the border is drawn by calling DrawBorder for the border object. A gray border is drawn for a disabled pane object, and special attention is paid to whether the border is being drawn or printed. In the latter case, the origin is set to 0,0.

c. If the pane itself is being printed (rather than being drawn), the intersection of the pane and the update rectangle is computed, and this value is used in the call to the DrawAll function for the pane.

d. If the pane is being drawn (rather than being printed), the DrawAll function is called with the entire update rectangle as its argument (that is, the drawing is not clipped to its own pane).

3. The DrawAll function is fairly complex—it handles the drawing of the current pane and all of its subpanes (subviews). The steps it takes are as follows:

a. The first step in drawing the current pane (subview) is to convert its update area from global to frame (QuickDraw) coordinates.

b. Then it is determined whether the pane is being printed or drawn. If it is being printed, then the clipping rectangle is set to the intersection of the drawing area and the pane's aperture (area inside its borders); otherwise, the area to be drawn is not clipped.

c. Next the pane's `itsEnvironment` member variable is tested to determine whether the pane has an associated CEnvirons object. The CEnvirons class provides extended drawing facilities for a given CPane-derived

object. If the `itsEnvironment` pointer is not `NULL`, then the Background function of that object is called with the drawing area and a pointer to the current pane. In fact, because CEnvirons objects can be linked, one to another, in a chain, the Background function calls itself, recursively, for each of the objects in the chain. Obviously, if the `itsEnvironment` object is `NULL`, then the Background function is not called.

d. Then the Draw function for the pane is called with the drawing rectangle as its argument.

e. Then, once again, the `itsEnvironment` variable is tested and if it holds a non-`NULL` pointer, its Foreground function is called. This allows an object to perform more embellishment of the foreground of the pane, after the contents of the pane have been drawn.

f. After the foregoing drawing operations are complete, if the status of the pane reflects that it is disabled, then the GrayOut function is called to draw over the entire drawing area with a gray pattern, using the `patBic` transfer mode. This shows the pane as being disabled.

g. At this point, the pane has been drawn, including any background or foreground embellishments that were applied as a result of the associated CEnvirons object chain. The DrawAll function then proceeds to call Pane_Draw for each of the current pane's subviews (if any). This results in the innermost pane being drawn last, and in front of all of the rest. The description of the Pane_Draw function and the functions it calls begins on page 302.

h. After all of the current pane's subpanes (subviews) have been drawn, then the DrawAll function calls the PutBackEnvironment function, which tests whether the `itsEnvironment` object exists and calls the TearDown function for that object. The objective here is to reset the drawing environment of a pane, so that when it is redrawn, the correct beginning state will exist. This step concludes the drawing process for a pane and its enclosed panes (subviews).

It is not entirely clear what might constitute a drawing environment, but if, for example, printed pages need to contain an underlying "watermark" symbol, or some other background or foreground contents, the CEnvirons class provides the ability to associate this with panes on an individual basis.

Handling Activate and Deactivate Events

The DispatchEvent function in the CSwitchboard class determines whether the current event is the `activateEvt` (activate). When this event occurs, the function tests whether the "activebit" is set in the event. If so, then it calls the DoActivate function, the steps for which are as follows:

1. The DoActivate function first tests the `gInBackground` global variable, to determine whether the application was suspended previously. If `gInBackground` is `TRUE`, then the Resume function for the application object (`gApplication`) is called. The Resume function determines whether the front window belongs to the application. If so, the `gInBackground` variable is set to `FALSE`, and the Resume function in the CDirectorOwner class is called. That function performs the following steps:

    a. The Resume function of the CDirectorOwner class loops through its list of directors (`itsDirectors`) and calls the Resume function for each director. Generally speaking, these directors are CDocument, CDialog, or other window director (CDirector) objects.

    b. The Resume function of the CDirector class calls the Resume function of the CDirectorOwner (its base class) as its first step. (This step refers to the function mentioned in the foregoing step, which refers to the application's list of CDirector objects—CDocument objects. The step currently being discussed refers to a document's list of directors—usually CDialog objects.)

    c. After the Resume function of the base class is called, then it is determined whether the director's `active` member variable is `TRUE`. If not, then the Resume function of the CDirector class returns.

d. If the `active` variable is `TRUE`, then it is determined whether the director's window (`itsWindow`) is owned by the director (that is, whether the director is the window's supervisor) and also whether the `activateWindOnResume` variable is `TRUE`. If both of these are `TRUE`, then the `active` variable is set to `FALSE`, the Activate function for the window object is called, and the `activateWindOnResume` variable is set to `FALSE`. The functionality of the Activate function in the CWindow class is covered in the next step.

2. If the application object's `gInBackground` variable is `FALSE`, then the application's front window (if any) is located. If it exists, the Activate function of the window object (CWindow) is called. If the window is already active, the Activate function does nothing. If the window is not active, then Activate calls the HiliteWindow toolbox function to ensure that the window appears to be active (its title bar, grow, zoom, and close box icons are redrawn), and then the Activate function of the CView class is called. The Activate function in the CView class loops through all of the views in the window, calling the Activate function for each of them. When control returns to the Activate function in the CWindow class, the ActivateWind function of the window's supervisor—a CDirector object—is called. That function in the CDirector class calls its Activate function, which perform the following steps:

   a. If the director is not currently active, then execution skips to step e in the sequence that follows.

   b. The `active` member variable for the director is set to `TRUE`, and then the function tests whether a window exists and whether or not it is a floating window.

   c. If the window exists and is not floating, then the BecomeGopher function for the `itsGopher` object pointer is called to request that the object become the new current gopher. If the current gopher refuses to resign, then the Activate function of the CDirector class returns to the ActivateWind function, which returns to the Activate function of the CWindow class.

d. If the current gopher resigns as the gopher, then the object pointed to by the `itsGopher` variable has become the current gopher and the BecomeGopher function returns a `TRUE` result. In this case, the Activate function continues by setting the `gSleepTime` global variable to 0, causing the event loop to perform all of the idle tasks (if any), at the earliest possible moment. Execution continues in the next step.

e. At this point, whether or not the director's `active` variable is `TRUE`, the Activate function of the CDirector class calls the ActivateDirector function of the director's supervisor—a CDirectorOwner object—with a pointer to the current director (`this`).

f. The ActivateDirector function of the CDirectorOwner class tests whether its list of directors has at least one member. If so, it calls the BringFront function of the list of directors (`itsDirectors`). This moves the specified director's pointer to the front of the list. Then the ActivateDirector function sets the `active` member variable to `TRUE`, returns control to the Activate function of the CWindow class, and pops back through the stack of calls to the event loop, from which DispatchEvent was called to handle the Activate event.

If the "active-bit" is not set in the `activateEvt`, then it is a deactivate event, and the DispatchEvent function calls DoDeactivate to handle the event. Steps performed by the DoDeactivate function are as follows:

1. The front window is examined to determine whether it is a "system" window (usually a desk accessory). If so, then the DoDeactivate function calls the Suspend function for the application object (`gApplication`).

2. The Suspend function of the CApplication class tests whether the `gInBackground` member variable is `TRUE` (indicating that the application is already running in the background). If so, it does nothing further. If the application is not currently executing in the background, the Suspend function of the CDirectorOwner class is called, the `gInBackground` variable is set to `TRUE`, and then the BecomeGopher function is

called to make sure that the application object is the current gopher when the application's execution is suspended. (By the way, what I mean by "execution is suspended" is that the application is no longer in the foreground, although background execution can continue by virtue of the multitasking features of the Mac OS and the TCL's ability to dispatch Idle events when nothing else is happening.)

3. When, in the foregoing step, the Suspend function of the CDirectorOwner class is called, that function loops through its list of directors and calls the Suspend function for each such object in the list.

4. The Suspend function in the CDirector class tests whether the director is active. If not, it simply returns. If the director *is* currently active, then it is determined whether the director owns (is the supervisor of) the window object contained in the `itsWindow` member variable and whether that window is active. If both of these are `TRUE`, then execution continues; otherwise, the function returns.

5. If the window is owned by the director *and* it is active, then the Deactivate function for the window object is called. This action results in the following actions:

   a. The HiliteWindow toolbox function is called to cause the window to be redrawn as inactive (the title bar is redrawn in white and the grow, zoom, and close box icons are erased).

   b. Next, the DeactivateWind function of the window's supervisor (the original CDirector object) is called. That function calls the director's Deactivate function.

   c. The Deactivate function of the CDirector class first tests whether the director is active. If not, then execution continues at the next step. The function continues in the case where the director is active by setting its `active` variable to `FALSE` and then testing whether the current gopher is identical to the director object. If so, then the function calls BecomeGopher for its supervisor to ensure that an inactive director is not the current gopher, and then the function returns. If the director is not the current gopher, then execution continues at the next step.

d. Whether or not the director was active in the foregoing step, the Deactivate function of the CDirector class calls the DeactivateDirector function of its supervisor—a CDirectorOwner object. That function sets the `active` member variable to `FALSE` and returns.

6. After the window's Deactivate function is called, then the `active` variable for the director will have been set to `FALSE`, essentially deactivating the director. This is not necessary when an application has been suspended via a deactivate event, so the Suspend function in the CDirector class continues by setting the `active` variable back to `TRUE`, and then also sets the `activateWindOnResume` variable to `TRUE`. This ensures that when you switch to another application and then switch back, an active window will be reinstated as active after the switch.

Handling Suspend and Resume Events

When the user switches to another application, the current application is sent a `suspendEvt` (suspend event), and the new application is sent a `resumeEvt` (resume event). When the user switches back to the original application, the sequence of events is repeated.

The DoSuspend function is called by the DispatchEvent function when it receives a suspend event. That function simply calls the application object's Suspend function, as shown in Figure 7-1. Process steps for the Suspend function of the CApplication class are described in connection with the Deactivate event, beginning on page 307, in step 2.

The DoResume function is called by the DispatchEvent function when it receives a resume event. That function simply calls the application object's Resume function, as shown in Figure 7-1. Process steps for the Resume function of the CApplication class are described in connection with the Activate event, beginning on page 305, in step 1.

Handling High-Level Events

High-level events, also called "Apple events," are handled by installing "handlers," as described in Chapter 2, regarding the creation of the CSwitchboard object (see page 27). The whole topic of high-level events is covered in a later chapter; however, when a

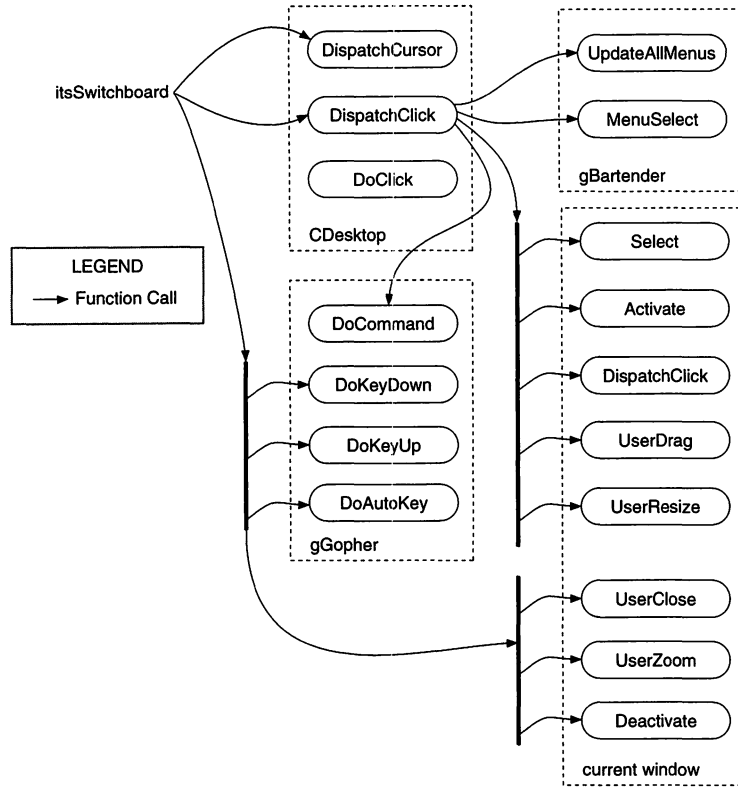

Figure 7-2
Additional event handling elements in the TCL

high-level event is input to the DispatchEvent function, its response is to call the DoHighLevelEvent function, which determines whether the user's system can handle these events. If so, it calls the AEProcessAppleEvent toolbox function, recording any error result, and then returns.

More About Mouse Down Events

The DispatchEvent function of the CSwitchboard class receives `mouseDown` events, as described earlier (see page 299). Figure 7-2 depicts more of the detail in the process of handling these events. In particular, the DispatchClick function of the CDesktop class is responsible for determining what actions need to be performed.

The first action of the DispatchClick function of the CDesktop class is to call the FindWindow toolbox function to determine whether or not some portion of an existing window was clicked—in which case it returns a pointer to the window—and also what

part of the window, desktop, or menu bar was clicked. If the return value is `inDesk` or `inMenuBar`, the window pointer will always be `NULL`. The DispatchClick function then tests whether the window kind is `OBJ_WINDOW_KIND`, which is the type created by the TCL. If so, then the function tests whether the top window is a modal dialog and the click occurred in a window or the menu bar, either of which could cause the modal dialog to become inactive. This is not allowed, so the function calls SysBeep and returns if the situation is proven to be true. Otherwise, execution continues to the section of the DispatchClick function that determines what actions to perform in response to the `mouseDown` event.

Handling inDesk Clicks

If the part code returned by the FindWindow function is `inDesk`, then the mouse click was on the desktop, and not in any specific window. The code for this situation tests whether the top window exists and whether it is a modal dialog. If so, then the function calls SysBeep and returns.

If there is no top window or it is not a modal dialog, then the DispatchClick function continues by calling the CountClicks function (a global function in the CView source file). CountClicks determines whether the current click occurred in the same view as the previous click. If so, it increments the `gClicks` global variable. If the click is in a different view, then `gClicks` is set to 1. In either case, the `gLastViewHit` global variable is set to the pointer to the current view.

The final action in handling a click on the desktop is to call the DoClick function that is inherited by the CDesktop class from the CView class. That function is empty and the click is ignored. If you wish to handle the DoClick function call, you can do so by creating a subclass of CDesktop and then override that function.

Handling inMenuBar Clicks

If the part code returned by the FindWindow function is `inMenuBar`, then the mouse click was in the menu bar. In that case, the DispatchClick function calls the UpdateAllMenus function of the CBartender object (`gBartender`). The processing steps of that function were described earlier, in connection with a `keyDown` event, where the Command key was also held down (see page 300, step 1a).

After the dimming and unchecking operations associated with the UpdateAllMenus function, and the subsequent undimming and checking functions of individual UpdateMenus functions have been performed, the DispatchClick function calls the `MenuSelect` toolbox function to determine the menu and item that was chosen by the user (if any). The high-order 16 bits of the return value reference the menu number and the low-order 16 bits refer to the item (command), within that menu, that was chosen. If a menu command was chosen, MenuSelect highlights the menu in the menu bar. The DispatchClick function calls the DoCommand function for the current gopher if a command was chosen. In either case, processing of a click in the menu bar is completed by calling `HiliteMenu` to remove the highlight from the chosen menu (if any).

Handling inSysWindow Clicks

If the part code returned by FindWindow is `inSysWindow`, then the user has clicked in a system window (that is, a desk accessory). In that case, the DispatchClick function calls the SystemClick toolbox function to handle the click.

Handling inContent Clicks

If the part code returned by FindWindow is `inContent`, then the click has occurred in the content region of a window. If the window kind is not `OBJ_WINDOW_KIND` (that is., a window created by the TCL), then the `inContent` mouse click is ignored.

If the window kind is `OBJ_WINDOW_KIND`, then the DispatchClick function continues by testing whether the window is inactive (that is, its `active` variable is `FALSE`) or whether it is a floating window that is not the front window. If either of these cases is `TRUE`, then the steps taken are as follows:

1. The Select function for the window object is called. That function calls SelectWind for the window's enclosure (which is usually the CDesktop object). The SelectWind function performs a number of steps, depending upon the type of window that was clicked and the current status of the application. These are summarized as follows:

    a. If the window is the top window, is also a floating window, and the front window is not a desk accessory window, then the function returns.

b. If the front window belongs to a desk accessory, then a click in a different window requires activating the application. In this case, the window pointer for the DA's window is placed behind the bottom window of the desktop's window list (`itsWindows`).

c. If the window in which the click occurred is a floating window and is not the "top" floating window (`topFloat`), then the function tests whether a modal dialog currently exists. If so, the floating window is placed behind the modal dialog; otherwise, it is brought to the front. In either case, the window is brought to the front of the list of floating windows, and its Show function is called to make the window visible.

d. If the window in which the click occurred is not a floating window and is not the top window, then the window list is rearranged so that any modal windows are in front, floating windows are directly behind modal windows, and any non-modal window is behind all of the rest. Then the clicked window is shown, and its `visible` variable is set to `TRUE`.

e. If, however, the window in which the click occurred is a modal dialog, is not the top window, and no modal dialogs are currently visible, then the function sets the top window to the clicked window and tests whether a desk accessory was active at the time of the click. If so, then the HiliteWindow toolbox function is called to make the DA window inactive, and the sSetInactive function is called for the old top window (this sets the `active` variable for the window's director to `FALSE` and also sets the `activateWindOnResume` variable to `FALSE`). The sSetActive function is then called for the new top window (this sets the `active` variable for the window's director to `TRUE`; if the `gInBackground` variable is true, it sets the `activateWindOnResume` variable to `TRUE`).

f. If the window in which the click occurred is not the top window, no modal dialogs were visible, this window is a modal dialog, but no desk accessory was active, then the old top window is deactivated (if the `gInBackground` variable is `TRUE`, then the sSetInactive function is called;

otherwise, the Deactivate function is called for the old top window). Then if the `gInBackground` variable is `TRUE`, the SelectWind function loops through the list of windows (`itsWindows`), calls the sSetInactive function for each, and then calls the sSetActive function for the new top window. If the `gInBackground` variable is `FALSE`, then the Activate function of the new top window is called.

The foregoing steps are rather difficult to describe in words, but I hope the steps will help you understand what is going on when you look at the code in the SelectWind function of the CDesktop class.

2. After the Select function continues (having selected the correct window as the top window), the DispatchClick function tests whether the `actClick` variable for the window object in which the click occurred is `TRUE`. If not, then processing of the click is complete; otherwise, the Activate function for the clicked window object is called and processing continues.

At this point in the DispatchClick logic, for a click that occurred in the content portion of a window, the function tests whether the `wantsClicks` variable of the clicked window object is `TRUE`. If so, then the UpdateWindows function is called, followed by calling the DispatchClick function for the clicked window. The UpdateWindows function loops through the list of windows (`itsWindows`), calling the Update function for each window whose update region is not empty.

The DispatchClick function of the CWindow class merely calls the DispatchClick function of the CView class (its base class). That function tests whether the "hit" view has a subview that contains the point on which the mouse was clicked. If so, it calls DispatchClick recursively for that subview. If no "hit" subview is found, the current subview's object pointer is used for the remaining operations, which are as follows:

1. The CountClicks function is called for this subview, to tally the number of consecutive clicks that have occurred.

2. If the `canBeGopher` variable for the object is `TRUE`, then the BecomeGopher function is called for the object. If the return value from that function is `FALSE` (indicating that the cur-

rent gopher refuses to resign), then the DispatchClick function returns; otherwise, the DoClick function for the view is called. The DoClick function is empty in the CView class; however, many other classes in the TCL implement the DoClick function (for example, see the explanation of DoClick for the CControl class in Chapter 6, on page 266).

If the `wantsClicks` variable for the clicked window is `FALSE`, then the CountClicks function is called to tally the number of consecutive clicks for the current subview, and the DoClick function for the CDesktop class (inherited from CView) is called. This function is empty, but if you subclass CDesktop, you can override this function.

Handling inDrag Clicks

If the part code returned by the FindWindow function is `inDrag`, then the click occurred on the title bar of the window, and it is assumed that the user may wish to drag the window to some other position on the screen. The function first tests whether the window kind is `OBJ_WINDOW_KIND`, indicating that it is a TCL window. If not, the function returns. If so, then the UserDrag function for the current window object is called with the event record as its argument. The steps taken by the UserDrag function are as follows:

1. The Drag function in the CWindow class is called with the current event record as its argument.

2. The Drag function of the CWindow class calls the DragWind function of the window object's enclosure (CDesktop) with a pointer to the current window and the event record as arguments. The DragWind function of the CDesktop class performs a number of steps, as follows:

    a. If the window is not the top window, is not the top floating window, and the Command key is not down, then the SelectWind function is called to select the window (see the description of the SelectWind function, beginning on page 312, in step 1). Execution continues with the next step.

    b. The state of the mouse button is tested to determine whether it is still down. If not, then the function returns.

c. The current port is saved, and then the port is set to the window's port. The PenNormal toolbox function is called to set the default pen properties, a variable named `dragBox` is set to the window's bounds, and then those values are inset by a constant `DRAG_MARGIN`, which is defined to be 4 pixels. The current clipping region is saved, the `strucRgn` (content region plus the frame) of the window is copied into a utility region (defined in the TCL as `gUtilRgn`), and then that region, the mouse position in the event record, and the `dragBox` are passed to the DragGrayRgn toolbox function to track the mouse and drag a dotted outline of the window while the mouse button is still down. The return value from the DragGrayRgn function is the new location of the mouse after the drag is complete.

d. The saved clipping region is restored, the window is moved to the final location by calling the MoveWindow toolbox function (if the window was actually moved), the saved port is restored, and the ForceNextPrepare function is called to ensure that the port is reset before any new drawing operations take place.

3. The UserDrag function continues by testing whether the result of calling the application's Factoring function is `TRUE` *and* the window is *not* modal and *not* floating. If one or more of these conditions is `FALSE`, then the function returns. If they are all `TRUE`, then the new window's bounds (in global coordinates) are passed as an argument to the SendSetPropertyToThis function of the CAppleEvent class (which is also a base class of the CWindow class) to send an Apple Event to this same application, reporting the window's new position.

The foregoing step is an example of how the TCL is factored to send events to itself, allowing your application to be scriptable and recordable. We will cover this intrinsic feature of the TCL in a later chapter.

Handling inGrow Clicks

If the part code returned by the FindWindow function is `inGrow`, then the user has clicked in the window's grow box and it is assumed that the window is going to be resized. If the window

kind is `OBJ_WIND_KIND`, then the window's UserResize function is called; otherwise the event is ignored. The steps taken by the UserResize function are as follows:

1. The UserResize function of the CWindow class begins by calling the Resize function. That function computes the current size of the window (width and height) and then calls the GrowWindow toolbox function to track the mouse and allow the user to change the size of the window. After GrowWindow returns, the function tests whether the window has been resized. If so, it calls the ChangeSize function in the CWindow class.

2. The ChangeSize function in the CWindow class calls the SizeWindow toolbox function with the new height and width for the window, as well as a `TRUE` value for the `updateFlag` argument, so that any newly exposed area of the window will be added to the update region. Then the ChangeSize function tests whether the window contains any subviews. If so, it loops through the list of subviews (`itsSubviews`) and calls Pane_AdjustToEnclosure for each. This global function in the CPane class calls the AdjustToEnclosure function for the specified pane.

3. The AdjustToEnclosure function in the CPane class has to determine whether a pane should be moved or resized when its enclosure is resized, depending upon its "sizing" characteristics. If you recall, each pane has both a horizontal and vertical sizing characteristic that can be set when the pane is designed in the VA. These must be taken into account when the pane's enclosure changes size. The action to take when the pane's enclosure is resized is determined by the AdjustHoriz and AdjustVert functions, which are called, in turn, to compute movement and resizing values. The AdjustHoriz function of the CPane class determines whether the pane's sizing characteristic is `sizFIXEDLEFT`, `sizFIXEDRIGHT`, or `sizELASTIC`, and then calculates the `offset`, `moved`, and `sized` results as follows:

    a. If the `hSizing` characteristic is `sizFIXEDLEFT` and the left edge of the enclosure has changed position, then the `offset` result is set to the value of that delta change and the `moved` result is set to `TRUE`.

b. If the hSizing characteristic is sizFIXEDRIGHT and the right edge of the enclosure has changed position, then the offset result is set to the value of that delta change and the moved result is set to TRUE.

c. If the hSizing characteristic is sizELASTIC, the delta change for both the left and right edges of the enclosure is computed, and if either value is nonzero, the sized result is set to TRUE.

4. The AdjustVert function of the CPane class determines whether the pane's sizing characteristic is sizFIXEDTOP, sizFIXEDBOTTOM, or sizELASTIC, and then calculates the offset, moved, and sized results as follows:

a. If the vSizing characteristic is sizFIXEDTOP and the top edge of the enclosure has changed position, then the offset result is set to the value of that delta change and the moved result is set to TRUE.

b. If the vSizing characteristic is sizFIXEDBOTTOM and the bottom edge of the enclosure has changed position, then the offset result is set to the value of that delta change and the moved result is set to TRUE.

c. If the vSizing characteristic is sizELASTIC, the delta change in both the top and bottom of the enclosure's edges is computed. If either value is nonzero, the sized result is set to TRUE.

5. After both the AdjustHoriz and AdjustVert functions have returned, then the AdjustToEnclosure function tests the return values to determine what actions to perform. If both the moved and sized results are TRUE, then the AdjustToEnclosure function proceeds as follows:

a. The Offset function is called with the horizontal and vertical offset values and a redraw argument of TRUE (forcing the pane to be redrawn).

b. The Offset function tests whether the redraw argument is TRUE and calls the Refresh and RefreshBorder functions to redraw the area of the window where the pane currently resides. The hOrigin and vOrigin values for the pane are modified according to the amount the pane

has moved, Offset recalculates the `hEncl` and `vEncl` values for the pane, and then calls CalcAperture to recalculate the pane's aperture. If the `redraw` argument is `TRUE` (as it is in this case), then the Refresh and RefreshBorder functions are called to update the pane's content at its new location.

c. If the pane has any subviews (`itsSubviews`) the Pane_EnclosureMoved function is called for each of these, which results in the Offset function being called for each of the subviews of the current pane. These are, in essence, recursive calls of the Offset function to enable the changes associated with changing the size of a pane's enclosure to ripple down throughout a set of nested panes, causing the innermost pane to be drawn last.

d. After the Offset function has been called, the ChangeSize function is called for the pane (it has been both moved and resized), with the delta change in size values and a `redraw` argument of `TRUE`.

e. The ChangeSize function begins by testing whether the `redraw` argument is `TRUE`. If so, then the Refresh and RefreshBorder functions are called to cause the area occupied by the pane's current position is added to the update region.

f. If either the left or top of the pane has changed in position, the Place function is called for the pane. That function determines the new location of the pane in the window and calls the Offset function to cause it to be redrawn in the new location. Then the right delta size change is decreased by the amount the left edge was moved, the bottom delta size change is decreased by the amount the top edge was moved, and then the left and top size changes are set to 0.

g. The ChangeSize function then calls ResizeFrame to adjust the width and height of the pane, change the location of its frame coordinates, and change its `hOrigin` and `vOrigin` values. CalcAperture is called to calculate the new aperture of the pane, and then the `redraw` argument is tested to determine whether it is `TRUE`. If so,

then the Refresh and RefreshBorder functions are called to force the pane (at its new size and position) to be added to the update area. If the `autoRefresh` property for the pane is `FALSE` *and* the pane is really visible, then the ValidRect toolbox function is called to revalidate the area of the pane; and if the pane has a border, then the border's rectangle is calculated and ValidRect is called to validate its rectangle. These actions prevent the pane from being drawn automatically when the enclosure's size is changed. The default setting is for `autoRefresh` to be `TRUE`, thereby causing the pane to be redrawn.

h. In any event, the ChangeSize function completes its execution by testing whether the pane has any subviews and calls the Pane_AdjustToEnclosure function for each of these (that function is described on page 317, in step 2).

6. In the AdjustToEnclosure function of the CPane class, if the `moved` result is `TRUE`, but the `sized` result is `FALSE`, then only the Offset function is called for the pane. That function is described beginning on page 318, in steps 5b and 5c.

7. In the AdjustToEnclosure function of the CPane class, if the `moved` result is `FALSE`, and the `sized` result is `TRUE`, then the ChangeSize function is called for the pane. That function is described beginning on page 319, in steps 5e through 5h.

8. If, in the AdjustToEnclosure function of the CPane class, neither the `moved` nor `sized` results is `TRUE`, then the function calls the CalcAperture function to recalculate the pane's aperture (which might have changed when the enclosure's size was changed).

9. After all of the panes in the window have been moved or resized, as necessary, the ChangeSize function of the CWindow class regains control. That function calls ForceNextPrepare to ensure that the port is reset before anything is redrawn, and then calls the Update function of the CWindow class to cause the entire window's contents to be redrawn. Prior to this, all of the calls to Refresh and RefreshBorder have only added rectangular areas to the window's update region and no drawing has actually taken place. As you saw in step 5g, when `autoRefresh` was `FALSE`, a pane and its bor-

der's areas were revalidated in the update region. All of the foregoing steps are intended to determine what really needs to be redrawn when the window's contents are updated. By calling Update directly, the ChangeSize function in the CWindow class has "faked" an update event. To reread what occurs in the processing of that event, refer to the description beginning on page 302, in step 1.

10. After the window has been redrawn, control returns to the UserResize function. The function then tests whether the `factoring` property of the application is `TRUE`, that the window is *not* a floating window, and that it is *not* a modal window. If these are all `TRUE`, then the bounds of the window are ascertained, and the SendSetPropertyToThis function in the CAppleEventObject class is called to send an Apple Event of the window's new bounds to the current application (to be acted upon or recorded, as you desire).

The foregoing is the final step in the UserResize function, which was called to handle a mouse down event in the size box of the current window.

Handling inGoAway Clicks

When the user clicks in the close (go-away) box of the current window, the DispatchClick function of the CDesktop class tests whether the window pointer is `NULL`. If so, then the click is ignored. If the window pointer is not `NULL`, then the TrackGoAway toolbox function is called to track the mouse pointer, returning `TRUE` only if the mouse button is released while still inside the close box of the window. If the result of the function is `TRUE`, then DispatchClick calls the UserClose function in the CWindow class.

The UserClose function tests whether the `factoring` property of the application is `TRUE`. If so, it sends an Apple Event of `kAEClose` to itself, indicating that the event should be executed and also recorded (if desired). If the `factoring` property is `FALSE`, then the Close function for the CWindow class is called. That function calls the CloseWind function for its supervisor (a CDirector object). The CloseWind function in the CDirector class first determines whether the window being closed is the same object as that whose pointer is stored in the `itsWindow` member

variable. If so, then the CloseWind function performs a number of actions, as follows:

1. The Close function of the director is called with a value of `FALSE` for the `quitting` argument.

2. A given director object usually overrides the Close member function to perform actions specific to the director when that function is called. In the case where the director is derived from the CDocument class, the director attempts to close the document's file (if any) and then calls the Close function in the CDirector class. Following that, the director's Close function returns `TRUE`, indicating success in the performance of the Close function.

   In the case where the director is derived from the CDialogDirector class—itself being derived from CDirector—the Close function initiates the EndDialog process, calls the Close function of the CDirector class, and then returns `TRUE`, indicating success in the performance of the Close function.

3. In any case, when the Close function of the CDirector class is called, that function tests whether the `alreadyClosing` variable is true. If so, it returns with a result of `FALSE`; otherwise, the `alreadyClosing` variable is set to `TRUE` and the function proceeds to call the Close function of the CDirectorOwner (its base class) with the value of the `quitting` argument passed to it (which is `FALSE` in this case).

4. The Close function of the CDirectorOwner class accesses the first object in its list of director objects (`itsDirectors`), which are the subdirectors for that director, and assigns this object to the `theDirector` variable and performs the following steps in a loop that terminates only when the value in the `theDirector` variable is `NULL`:

   a. It calls the Close function for the `theDirector` object.

   b. If the Close function returns a result of `FALSE` (indicating that it is unwilling to close), then the Close function of the CDirectorOwner class returns to the Close function of the CDirector class with a `FALSE` result.

   c. If the Close function returns a `TRUE` result, indicating that the close operation succeeded, the subdirector will

have been removed from the list of directors by virtue of the execution of the director's destructor function (see step 5 next), so that, in the event that the list is empty after the preceding operation is complete, the Close function of the CDirectorOwner will terminate execution and return a `TRUE` result.

d. If the list is not empty, the new first element in the list is chosen, and the loop will repeat.

5. Returning to the Close function of the CDirector class, in the case where the Close function of the CDirectorOwner class has returned a value of `TRUE`, the function disposes of its own object by calling the TCLForgetThis macro, which calls Dispose with an argument of `this`. Dispose executes a `delete this` statement, disposing of the object. When the destructor of the object is executed, if the director owns the window object stored in the `itsWindow` variable, the window is disposed by calling TCLForgetObject with the window pointer as its argument, and then the RemoveDirector function of its supervisor (a CDirectorOwner object) is called. This removes the CDirector-derived object from the list of its directors (`itsDirectors`).

Whether or not the window being closed is identical to the object whose pointer is stored in the director's `itsWindow` member variable, the CloseWind function in the CDirector class disposes of the window by calling TCLForgetObject with the window pointer as its argument. Then if both the `itsWindow` and `itsDirectors` (list of subdirectors) are `NULL`, the steps described in detail beginning on page 322, in step 1, are executed.

After calling the Close function for the CDirector object, the CloseWind function returns to the Close function of the CWindow class, which, in turn, returns control to the UserClose function in the CWindow class, which returns control to the event loop.

Handling inZoomIn and inZoomOut Clicks

When the user clicks in the "zoom" box of the window, the part code returned by the FindWindow toolbox function is either `inZoomIn` or `inZoomOut`, depending upon whether or not the window has already been expanded (zoomed) to its full size. In ei-

ther case, the DispatchClick function of the CDesktop class first tests whether the window pointer is NULL. If so, then the click is ignored; otherwise, the TrackBox toolbox function is called to track the mouse. If the button is released when the mouse pointer is outside the zoom box, then the click is ignored. If the mouse pointer is still within the zoom box when the button is released, then TrackBox returns a TRUE result, and DispatchClick calls the window object's UserZoom function.

UserZoom determines whether factoring is TRUE for the current application and that the window is *not* floating and is also *not* a modal dialog. If these are all TRUE, then the SendSetPropertyToThis function is called to send an Apple Event to the application, indicating the new zoom status of the window. It is assumed that the application will record the new status, in addition to performing the zoom operation.

If the application is not factored, or if the window is floating or a modal dialog, then the Zoom function for the CWindow class is called with a `direction` argument of TRUE for a zoom out, or FALSE for zoom in. The Zoom function performs the following actions:

1. The current window's contents are erased.
2. The window size is computed for a zoom-out operation, to be as large as possible for the current display device. There are complications to this. If the window spans more than one display device, the dominant device's characteristics are used when sizing the window.
3. The ZoomWindow toolbox function is used to perform the zoom in or zoom out function.
4. If the window's list of subviews (`itsSubviews`) is not NULL, then the Zoom function loops through the list, calling the Pane_AdjustToEnclosure function for each (the actions taken by this function are described beginning on page 317, toward the end of step 2).
5. After all of the subpanes have been adjusted to the new enclosure size, the InvalRect toolbox function is called to invalidate the port rectangle of the entire window, causing it to be redrawn when the next update event is processed.

6. Finally, if the window is floating, the SelectWind function for the window's enclosure (CDesktop) is called to make the window the current top window. (The SelectWind function's process steps are described beginning on page 312, in step 1.)

Event Processing Summary

The foregoing sections have gone into a great amount of detail to describe the actions of the TCL in response to all possible events (except for high-level events, which will be covered in a later chapter). In general, Figure 7-1 illustrates the gross logic for handling other than mouse-down events and Figure 7-2 illustrates the gross logic for handling mouse-down events.

Most of the complexity of event handling occurs when windows are moved or changed in size. This affects the positions or sizes of all of the subviews of the window. And because views can be nested inside other views, the process of resolving the new positions and sizes of these is necessarily complex. However, the good news is that the TCL does almost all of the work, and you need only perform whatever operations are needed when your own Draw functions are called. Controls and other TCL objects are drawn automatically. You needn't add code to provide these features, unless you are subclassing a control object and wish to customize a control's appearance.

Chapter 8

Examining Template and Collection Classes

In this chapter, I am going to describe the collection classes defined in the TCL. Some of these are implemented with a generic class template (CPtrArray), and they will be described also. In general, the TCL implements only arrays and lists. Although there are no stacks or dictionary-type collections, these can be implemented easily with the existing facilities.

In addition to the existing collections, we will discuss the iterator classes that provide a robust means for iterating through a collection and performing actions safely that might otherwise operate incorrectly in procedures that rely on index values for element selection. There are two iterator classes. One is CArrayIterator, and the other is CVoidPtrArrayIterator. The collection and iterator class hierarchy is shown in Figure 8-1.

Using the CArray Class

In addition to being the base class for the other collections, CArray is fully capable of being used as a concrete class. That is, you can create CArray objects and use them in your applications.

The CArray class provides the means to hold items of any type and size, as long as all of the items are the *same* size. When you create a CArray object, you specify the size of each of the items you intend to store in the array, and, optionally, the number of slots by which the array should grow when more space is needed. The default block size is three slots. If you expect to allocate a lot of array items, then you might set the block size a bit larger than this; however, each time the array is increased in size, the block size multiplied by the item size is used to allocate new storage.

The CArray class allocates contiguous storage, in the form of a handle, to contain the array's data. Access to items in the array is very efficient, because the requested item's location can be com-

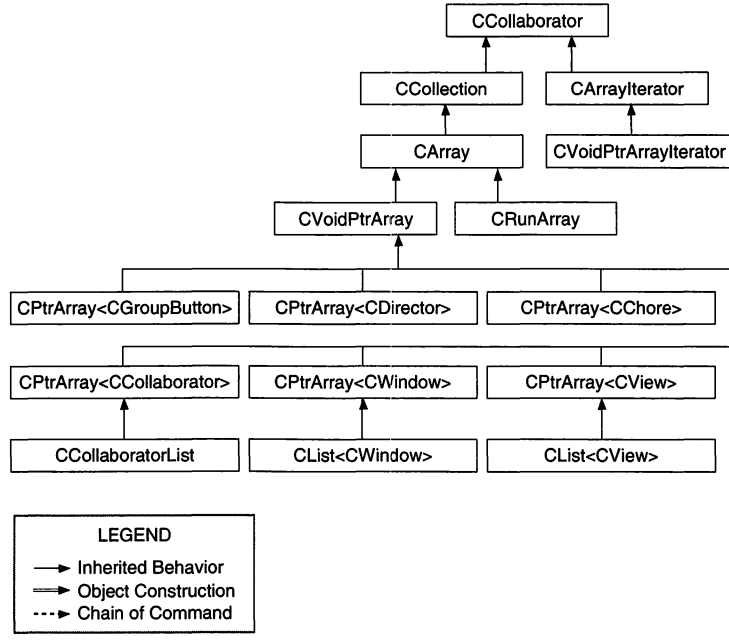

Figure 8-1
Collection and iterator class hierarchy

puted from its index value (since the items's size is known). Some operations on the array require that it be searched for an item that meets a condition specified by a `compare` function (for example, as in the Search member function). Other operations may require that array items be moved to make space for a newly inserted item (InsertAtIndex) or a deleted item (DeleteItem). Items can be added (Add) to the end of the array efficiently.

I have shown a number of examples of using the CArray class in earlier chapters. For example, in Chapter 5, I used a CArray object to hold CCat object pointers. The code to create the CArray object was placed in the ICMain function of the CMain class (the document owns the `categories` list in this case). The code to create the array is as follows:

```
void CMain::ICMain()
{
   Ix_CMain();

   // create the categories list
   categories = TCL_NEW (CArray, (sizeof (CCat *)));
}
```

As is evident in the foregoing, the creation of the CArray object is accomplished with a single statement. The `TCL_NEW` macro creates a new CArray object with items that are the size of a CCat object pointer. I could have used a CVoidPtrArray for this purpose, but I am comfortable using the CArray class.

The code for adding items to an array is very simple. Once again, this is illustrated in Chapter 5 for the `categories` array. A function called AddCategory in the CMain class was created to perform this task. The code is as follows:

```
void CMain::AddCategory (CCat *aCat)
{
   // the easiest way to add a category is to add it
   // to the end of the array and then sort the array.

   categories->Add (&aCat);
   SortCat();
}
```

The foregoing code adds a category to the end of the list (array) and then calls SortCat to sort all of the categories into order according to their names. Notice in the foregoing code that the Add function is called with a pointer to the item to be added. The item, in this case, is an object pointer, but we still pass a pointer to that pointer when we call the Add function. The most common mistake when using arrays is to forget to use a pointer to the object pointer (or other type of item) being added.

When you wish to retrieve an item from a CArray object, you must also supply a pointer to the container that is to receive the item. So, for example, the GetCategory function described in Chapter 5 shows how a CCat object pointer is retrieved from the `categories` array. The code is as follows:

```
CCat* CMain::GetCategory (long index)
{
   CCat *aCat;
   long num;

   num = categories->GetNumItems();
   if (index > num || index < 1)
   {
      return NULL;
   }
   categories->GetArrayItem (&aCat, index);
   return aCat;
}
```

In the foregoing, notice that the GetArrayItem function of the CArray class is being called with a pointer to the CCat object pointer and also an index—a `long` variable—of the item to be retrieved. Index values are positive values greater than zero (that is, they run from 1 through the number of items in the array).

It is also simple to delete an item from an array. The DelCategory function described in Chapter 5 shows how that operation is performed. The code is as follows:

```
void CMain::DelCategory (long index)
{
   long num = categories->GetNumItems();

   if (index <= num && index >= 1)
   {
      categories->DeleteItem (index);
   }
}
```

Although the foregoing code tests the index value to make sure that it is within the bounds of the array, the DeleteItem function requires only the index of the item to be deleted from the array.

Documentation for the CArray class in the TCL describes a number of other useful member functions. If you wish to change the contents of an item in a CArray object, then you can use the SetArrayItem member function. That function wasn't used for our `categories` array, but the function could be used as follows:

```
void CMain::SetCategory (CCat *aCat, long index)
{
   CCat *theCat = GetCategory (index);
   if (theCat == NULL)
   {
      ASSERT (!"Invalid category index");
   }
   categories->SetArrayItem (&aCat, index);
}
```

The foregoing code accesses a specified category by calling the GetCategory function with the item's index. After ensuring that the CCat object pointer returned by the function is not `NULL`, the function calls the SetArrayItem function to provide replacement contents for the element at the specified index in the array.

Looping Through CArray Objects with an Index

There are times when you will need to iterate through all of the items in a CArray object to find one that matches some comparison criteria. There are two ways to accomplish the iteration. If you are merely going to compare each item to determine whether it meets a specified criteria, then it is easy to access the number of items in the array, then loop through the items, one by one, until one is found (or not) that matches the criteria. For example, each CCat object has a name string that is stored in a CString object within the CCat object. If you are looking for a category whose name is "Mortgage," then you could do it as follows:

```
CCat *CMain::FindCategoryByName (CString name)
{
   CCat *aCat;
   long numItems;
   long index;

   //
   // find a category item by name and return a pointer
   // to the CCat object if found, or NULL if not found.
   //
   numItems = categories->GetNumItems();
   for (index=1; index <= numItems; index++)
   {
      categories->GetArrayItem (&aCat, index);
      if (aCat->catName == name)
      {
         return aCat;
      }
   }
   return NULL;
}
```

In the foregoing, there is no question of confusion in the array index, because the array is not being altered within the loop. However, in cases where the array *is* being altered (items added, moved, or deleted while the loop is in progress), then it is a lot safer to use a CArrayIterator object to iterate through the array.

An example of this is a function that deletes every array item whose taxable status (`catTaxable`) is `FALSE`, leaving only items that have tax consequences remaining in the array, after the loop is complete. It should be clear that if you delete a particular item in the array, then the index of the next item will be the same as the current item's index (because items beyond the deleted item are moved to occupy the empty space). While you could still cope

with this situation by using an index-oriented loop, it is much safer to use a CArrayIterator object for this purpose.

Looping Through CArray Objects with an Iterator

A CArrayIterator object uses the notion of a "cursor" to denote the current position in an array. The cursor does not point directly to an item in the array, but, rather, sits between the previous and next items in an array, as shown in Figure 8-2.

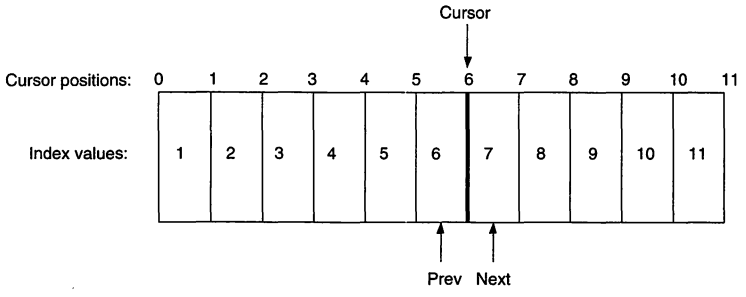

Figure 8-2
CArrayIterator cursor relationship to array index

While the foregoing figure illustrates a direct relationship to an array index and what is called the cursor and an array index, the cursor will keep track of the current position in the array, regardless of how many insertions, deletions, and movements of the data are made. An example of using a CArrayIterator object and its cursor position is illustrated by the SortCat function presented initially in Chapter 5. That function sorts an array of CCat object pointers, such that the objects are in order, from beginning to end of the array, according to the values of their `catName` fields. The code for the SortCat function is as follows:

```
void CMain::SortCat ()
{
   CCat *pCat, *nCat;
   Str255 pName, nName;

   CArrayIterator pIter (categories, kStartAtBeginning);
   CArrayIterator nIter (categories, 0);

   //
   // perform a simple N^2 sort of the categories array
   //
   while (pIter.Next (&pCat))
   {
      pCat->GetCatName(pName);
      nIter.MoveTo (pIter.GetCursor());
      while (nIter.Next (&nCat))
```

```
      {
         nCat->GetCatName(nName);
         if (IUCompString(nName, pName) < 0)
         {
            categories->Swap (pIter.GetCursor(), nIter.GetCursor());
         }
      }
   }
}
```

The foregoing code creates two CArrayIterator objects and initializes them both to point at the `categories` array. The first iterator (`pIter`) is initialized to start at the beginning of the array by using the `kStartAtBeginning` constant in its construction. The second iterator (`nIter`) is initialized with 0 as its `start` argument (it turns out that both of the `start` values are the same in this case, but I use 0 in the second iterator simply to indicate that I don't care about its start point when it is first constructed). Iteration begins by calling the Next member function of the iterator. A call to Next returns a pointer to the next item in the array (if any). I could also call Prev and it would return a pointer to the previous item in the array (if any). When there is no "next" item, Next returns `NULL`; when there is no previous item, Prev returns `NULL`. We can access the current cursor value at any time by calling GetCursor.

So the SortCat code begins iterating through the `categories` array by accessing the pointer to the item at its cursor position and then advancing the cursor. Then the CCat object's name string is acquired by calling the GetCatName access function for that object. After doing this, the code calls MoveTo for the second iterator, telling it to move its cursor to the location in the array where the first iterator's cursor now points. An inner loop iterates through the `categories` array, accessing each new item, acquiring its name, and then comparing that name with the name acquired with the first iterator. If the object associated with the second iterator has a name that sorts in front of the one for the first iterator's object, then the two array items are swapped by calling the Swap function for the `categories` object. The Swap function is called with the current cursor values for each of the iterators. The inner loop continues until the Next function returns `NULL` (indicating there is no next entry). The first iterator then calls its Next function to access the next array item, gets its name, moves the second iterator's cursor to the following location in the

array, and then runs the inner loop again. The process continues until the Next function of the first iterator returns a NULL result. When the operation is complete, the objects will have been sorted, from beginning to end of the array, in ascending alphabetical order by their category names. The sort algorithm in the foregoing is not terribly efficient. It is a simple N^2 loop within a loop; however, for a small number of category names, it will be efficient enough.

Creating a Push-Pop Stack

Creating new collection classes is quite simple. The CArray class can be used as the basis of most of these, because of its flexibility and robust behavior. In this section I will create a stack, upon which objects (or any other type of data) are pushed to be stored, and from which data are popped to be retrieved. It is a last-in first-out type of stack (a LIFO).

In order to implement the stack (CStack), I will need to derive the class from the CArray object. This is done in a header file named **CStack.h**, and whose corresponding source file is named **CStack.cp**. The header file is as follows:

```
/*************************************************************
CStack.h

     Interface for CStack class

   BASE CLASS = CArray

   Copyright© 1995 Richard O. Parker. All rights reserved.

*************************************************************/

#pragma once

#include <CArray.h>

class CStack: public CArray
{
public:

   TCL_DECLARE_CLASS

   CStack (long anElementSize, short blockSize = 3);

   virtual void Push(void *itemPtr);
   virtual void Pop (void *itemPtr);
};
```

The foregoing header file defines a constructor for the stack and two member functions (Push and Pop). This is a very simple implementation and shows only the very basic features that could be provided in a full stack implementation. For example, one might be tempted to add Forth-like Swap, Roll, Dup, and other stack operations for the new class. You are welcome to do so in your own version of the code. The source file (**CStack.cp**) that implements the foregoing declarations is as follows:

```
/************************************************************
CStack.cp
   CStack is a dynamic array that implements a conventional
   LIFO stack. Push and Pop member functions are included.

   Copyright© 1995 Richard O. Parker. All rights reserved.
************************************************************/

#ifdef TCL_PCH
#include <TCLHeaders>
#endif

#include "CStack.h"

#define TCL_ASSERT_INDEX(index)TCL_ASSERT((index > 0)\
   &&(index <= itemCount))

TCL_DEFINE_CLASS_D1(CStack, CArray);

/************************************************************
CStack

     Constructor

************************************************************/

CStack::CStack(long anElementSize, short blockSize)
   : CArray (anElementSize, blockSize)
{
   TCL_END_CONSTRUCTOR
}

void CStack::Push (void *itemPtr)
{
   Add (itemPtr);
}

void CStack::Pop (void *itemPtr)
{
   long numItems = GetNumItems();
   TCL_ASSERT (numItems > 0);
   GetArrayItem (itemPtr, numItems);
   DeleteItem (numItems);
}
```

As you can see from the foregoing, the implementation of a simple stack is quite trivial. I use member functions from the CArray

class to handle the addition of an item to the stack (Push) and then other member functions from CArray to handle the removal of an item (Pop). The Pop function includes an assertion that makes sure that one or more items is on the stack before an attempt is made to remove an item.

Using the CVoidPtrArray Class

One of the common uses of an array is to hold object pointers, as in a list. The TCL makes it easy to do so by providing a CVoidPtrArray class for this purpose. If you refer to Figure 8-1, you will see that the CVoidPtrArray class is derived directly from the CArray base class. This class provides a number of new functions that are important additions to the functionality of the CArray class, particularly for handling the storage, access, and manipulation of object pointers. The new functions are summarized in Table 8-1.

In addition to the new functions, the CVoidPtrArray object inherits all of the member functions of the CArray class. In addition to these, the CVoidPtrArray class also includes a set of built-in iterator functions, many of which are used in various areas of the TCL in manipulating lists of objects. The iterator functions included within the CVoidPtrArray class include those shown in Table 8-2.

Looping Through CVoidPtrArray Objects with an Iterator

In addition to the iterator functions built into the CVoidPtrArray class, the TCL provides a CVoidPtrArrayIterator class that provides the same functionality as the CArrayIterator class, but specifically for CVoidPtrArray objects. The CVoidPtrArrayIterator class uses the notion of a "cursor," as shown in Figure 8-2, to iterate through the array. The Next function accesses the next item (if any) in the array, returning its object pointer, and the Prev function accesses the previous item (if any) in the array, returning its object pointer. The GetCursor function obtains the current value of the cursor. The Next and Prev functions return `TRUE` if they are successful in accessing the specified object; otherwise, they return `FALSE`.

Table 8-1
New functions in the CVoidPtrArray class

Function	Description
Copy	Makes a copy of the specified object and returns its pointer.
Remove	Takes a pointer to an object, searches for the matching pointer in the array, and removes the item.
Includes	Returns TRUE if the array holds the specified pointer.
Offset	Returns the array index-1 of the specified pointer.
InsertAfter	Takes the pointer to an existing object and another pointer to a new object to be inserted after the existing object in the array.
InsertAt	Mimics the InsertAtIndex function of CArray
BringFront	Moves an object pointer to the first position in the array.
SendBack	Moves an object pointer to the end of the array.
MoveUp	Moves an object pointer one slot closer to the front of the array.
MoveDown	Moves an object pointer one slot closer to the end of the array.
MoveToIndex	Moves the specified pointer to the specified index position in the array.
FindIndex	Returns the index of the specified pointer.
FirstItem	Returns the first object pointer in the array.
LastItem	Returns the last object pointer in the array.
NthItem	Returns the Nth object pointer in the array.

Using the CRunArray Class

The TCL has defined a special array class, called CRunArray, that is derived from the CArray class, as shown in Figure 8-1. The CRunArray was defined primarily as an object for keeping track of "runs" of identical values. It is used by the CTable class to keep track of runs of identical row heights and column widths for a given table. It is also used in the CPrinter class to keep track of runs of identical "strip widths" and "strip heights."

Whenever you need to keep track of a sequence of consecutive long integer values, you can use a CRunArray object to minimize the storage required to do so.

Table 8-2
Built-in iterator functions for the CVoidPtrArray class

Function	Description
DoForEach	Iterates through the array, calling the specified function for each object in the array, with the object's pointer.
DoForEach1	Iterates through the array, calling the specified function for each object in the array with the object's pointer and one additional long integer argument.
FindItem	Iterates through the array, calling the specified function and continuing to loop until the function returns a TRUE result, in which case the object's pointer is returned; otherwise, NULL is returned.
FindItem1	Operates the same as FindItem but passes the specified function an additional long integer argument.
FirstSuccess	A synonym for FindItem.
FirstSuccess1	A synonym for FindItem1.
LastSuccess	Same as FirstSuccess, except that the loop runs from the end of the array to its beginning.
LastSuccess1	Same as FirstSuccess1, except that the loop runs from the end of the array to its beginning.

The CRunArray object contains a structure, consisting of a long integer `runLength` and a long integer `value`, as a member variable of the class. Therefore, each item in the array is the size of that structure. The functions in the CRunArray class are summarized in Table 8-3.

Using the CPtrArray Template to Create Collections

The CPtrArray class consists of a generic template that provides the ability to construct arrays of various specific kinds of pointers and then operate on those items. The CPtrArray class is declared as follows:

```
template <class T>
class CPtrArray : public CVoidPtrArray
{
   //
   // member function declarations
   //
};
```

In order to use the CPtrArray class, you need to create a specific instance of it. This is done in the TCL for all of the concrete classes

Table 8-3
CRunArray class function summary

Function	Description
GetNumItems	Returns the number of runs in the array.
InsertValue	Inserts a run of values into the array, at the specified item index.
SetValue	Sets the value of the item at the specified index to the specified value.
GetValue	Returns the value in the array, at the specified index.
DeleteValue	Decrements the run length for the specified index. If the run length becomes zero, it deletes the array entry.
DisposeAll	Removes all of the entries from the array.
SumRange	Computes the sum of the values within the start and end index values.
FindSum	Returns the index of the item in the array for which the sum of values from the beginning of the array to that item equals or exceeds the specified value.
FindRun	Returns the number of the run that contains the entry at the specified index.
InsertRun	Inserts a new run, consisting of value and run length, into the array at the specified index.
DeleteRun	Deletes an entire run at the specified index.

that are derived from CPtrArray, as shown in Figure 8-1. One example of these is the code for the CPtrArray<CDirector> class, whose source code is in the file **CPtrArray<CDirector>.cpp**, and is as follows:

```
#include "CPtrArray.h"
#include "CDirector.h"

#pragma template_access public

#pragma template CPtrArray<CDirector>

TCL_DEFINE_CLASS_M1(CPtrArray<CDirector>, CVoidPtrArray);

#include "CPtrArray.tem"
```

As you can see in the foregoing, the header files for both the CPtrArray and CDirector classes are included by `#include` statements in the source. In addition, the `#pragma template_access public` statement specifies that the scope of the instantiation of the class and its member functions

is public. The actual source code for the CPtrArray class is contained in the **CPtrArray.tem** file.

Let us assume that you wish to create a list of pointers to CWidget objects in your CMain (document class). In order to accomplish this, you will need to provide a declaration of the list within the class declaration in your **CMain.h** header file. Following is the way it would be done in the TCL:

```
typedef CPtrArray<CWidget> CWidgetList;

class CMain : public x_CMain
{
   CWidgetList *itsWidgets;

   // other member variable and function declarations
};
```

In addition to the foregoing, you would need to create a source file called something like **CPtrArray_CWidget.cpp** and add the code to instantiate the template for your class, as follows:

```
#include <CPtrArray.h>
#include "CWidget.h"

#pragma template_access public
#pragma template CPtrArray<CWidget>

TCL_DEFINE_CLASS_M1(CPtrArray<CWidget>, CVoidPtrArray);

#include <CPtrArray.tem>
```

The foregoing is all that is required to create a list of pointers to your CWidget class. Now that this has been accomplished, let's turn our attention to the functions available in the CPtrArray class. These are summarized in Table 8-4. In addition to functions for adding, removing, and inserting items into the list, the table also includes iteration functions that are built into the CPtrArray class. Of all of the functions listed, only the DisposeAll and DisposeItems functions (in addition to the constructor) are included in the **CPtrArray.tem** source file. The **CPtrArray.h** class declaration includes in-line definitions of the remaining functions, in the form of specialized calls to the corresponding functions in the CVoidPtrArray class.

Figure 8-1 shows a number of classes that are derived from the CPtrArray template class. In addition, there are three classes that

Table 8-4
CPtrArray class function summary

Function	Description
DisposeAll	Disposes all of the items in the list and then deletes the list itself.
DisposeItems	Disposes all of the items in the list.
Add	Adds the specified object pointer to the end of the list.
Remove	Removes the specified object pointer from the list.
SetArrayItem	Stores the object pointer at the specified index in the list.
Append	Adds the specified object pointer to the end of the list.
Prepend	Adds the specified object pointer to the front of the list.
InsertAfter	Inserts the specified object pointer after another specified object pointer in the list.
InsertAt	Inserts the specified object pointer at the specified index.
BringFront	Moves the specified object pointer to the front of the list.
SendBack	Moves the specified object pointer to the end of the list.
MoveUp	Moves the specified object pointer one slot closer to the front of the list.
MoveDown	Moves the specified object pointer one slot closer to the end of the list.
DoForEach	Performs a specified function for each item in the list.
DoForEach1	Performs a specified function for each item in the list, passing it an additional long integer argument.
First Item	Returns the first object pointer in the list.
LastItem	Returns the last object pointer in the list.
FirstSuccess	Performs a specified function for each item in the list and returns the pointer to the item for which the function first returns a TRUE result.
FirstSuccess1	Same as FirstSuccess, but calls the test function with one additional long integer argument.
LastSuccess	Same as FirstSuccess, but loops from the end to the beginning of the list.
LastSuccess1	Same as LastSuccess, but calls the test function with one additional long integer argument.

are derived from these. The three newly derived classes are CCollaboratorList, CList<CWindow>, and CList<CView>. All three of these are based upon the CList class, which is based upon the CPtrArray class, but adds PutItems and GetItems functions for writing and reading list items of the specified type to/from an

output or input stream. Both the CList<CWindow> and CList<CView> classes are defined in the **CList_CView.cpp** source file, whose contents are as follows:

```
#include "CList.h"
#include "CWindow.h"

#pragma template_access public

#pragma template CList<CView>
#pragma template CList<CWindow>

TCL_DEFINE_TEMPLATE_CLASS_D1(CList, CView, CPtrArray<CView>)
TCL_DEFINE_TEMPLATE_CLASS_D1(CList, CWindow,
CPtrArray<CWindow>)

#include "CList.tem"
```

The `TCL_DEFINE_TEMPLATE_CLASS_D1` macros in the foregoing code define dynamic template classes. The first argument is the variant class name, the second argument is the item's class name, and the third argument is the base class name. These templates work only for a single variant class, but are used quite a bit in the TCL to define lists of objects for which Object I/O is to be performed. I will discuss Object I/O in a later chapter.

There are two ways that template classes can be instantiated. One is by usage, and the other is by applying the `#pragma template` directive. The TCL uses this second method for all of the template classes. In fact, the compiler is instructed that all template accesses are external in the **TCL #includes.cpp** predefined headers source file for VA projects.

Collection and Template Summary

This chapter has described a number of collection classes, some of which are based upon templates that you must instantiate explicitly with the `#pragma template` directive to use with your data types. The Symantec C++ product also provides access to the "Standard Template Library" developed by Hewlett Packard, which offers queues, vectors, maps, stacks, trees, and a variety of other collection classes. You can also create quite a variety of collection classes by basing these on the CArray class itself.

The next chapter covers the intricacies of the Object I/O features of the TCL.

Chapter 9
Understanding and Using Object I/O

This chapter is all about input and output techniques. In addition to the Object I/O facilities for persistent object storage that is built into the TCL, I will cover a related capability that is an alternative to Object I/O for saving and restoring data in a more conventional manner or in a user-specified file format.

In regard to what I call a more conventional manner, I showed you how to implement simple text file input and output in Chapter 3, concerning the CTextData object and its member functions. This chapter will show you how to improve upon that method for saving your document's data in an arbitrary file format. Prior to showing the simplified techniques for writing and reading data files, I will discuss the TCL's built-in facilities for Object I/O.

What Is Object I/O and How Is It Used?

The short answer to the question posed by the title of this section is that Object I/O is a means for saving objects in an external data file, and in such a manner that, when the file is read at a later time, the objects can be reconstructed to create the same application state as when the objects were written. This means that whatever values the member variables of a given set of objects contain when the objects are written, those same values will be restored when the objects are recreated as the external file is read. It also means that if a member variable of an object points to some other object at the time both objects are written to the external file, that both objects will be restored fully when the external file is read. In other words, Object I/O is persistent storage and retrieval of object-oriented data.

Creating a User Interface View

Object I/O is central to the creation of the user interface objects that are specified by the developer using the VA tool. The VA writes a file called **Visual Architect.rsrc**, in standard resource format, that contains the information needed to recreate each of the windows, dialogs, and their subview objects, including all of the property settings defined by the developer, in 'CVue' resources.

In addition to the use of Object I/O by the TCL, the developer can also make use of these same features to save and restore objects that pertain to the application's own data. The remainder of the chapter is devoted to descriptions of both the interface and data-oriented Object I/O facilities.

Creating the Categories Dialog Object

Although menus—and, perhaps a few other user-interface features—are described by conventional resource types in the application file, all of the windows, dialogs, and the views and subviews contained within these are described by special 'CVue' resources. These resources cannot be read by programs like Apple's ResEdit, because their contents are too complex. Instead, they are read and written by the VA resource editor and code generation tool. They are also read by code in the TCL that has been fashioned to both read and write these data.

The best way to understand how the user interface is recreated by the TCL is to use an example such as the Categories dialog that was designed, implemented, and described in Chapter 5 for this purpose. The dialog, as it appears inside the VA, is shown in Figure 9-1. As is evident in the figure, the dialog contains a panorama (every VA view contains one of these as the top-level element), a list (CCatTable) with an accompanying scroll pane, and four buttons, one of which is the default button. The Categories dialog is created by calling the MakeNewWindow function in the x_CCategories class, whose code is as follows:

```
void x_CCategories::MakeNewWindow(void)
{
   itsWindow = TCLGetNamedWindow("\pCategories", this);
```

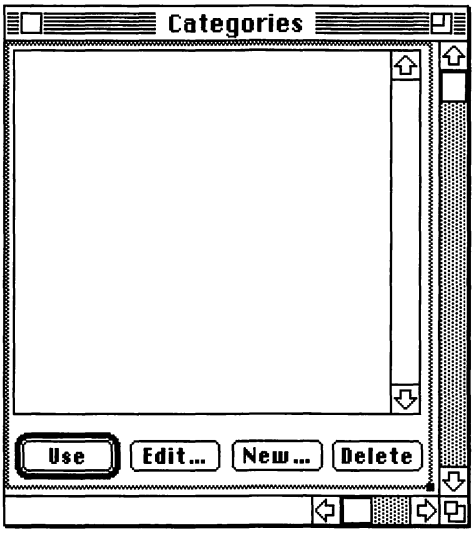

Figure 9-1
Categories dialog as it appears inside the VA

The foregoing is all that is needed to create the dialog and all of its subviews and controls. The MakeNewWindow function continues execution by accessing the pointers to the various objects created by the TCL; however, the acquisition of these pointers is not a necessary part of the dialog's creation.

You'll note in the foregoing MakeNewWindow code that to create the dialog, one need only call the TCLGetNamedWindow function with a Pascal string that identifies the window (in this case, it is a modeless dialog) by name and an object pointer that is taken to be the window's supervisor (shown as `this` in the code).

Accessing the View Resource

The TCLGetNamedWindow function can be found in the **ViewUtilities.cp** file in the section of VA Library source files of the project. The sequence of events in recreating the Categories dialog window, starting with the execution of the TCLGetNamedWindow function, is as follows:

- The function calls GetNamedResourceCanFail, passing it the Pascal name of the window and the `'CVue'` resource type. A handle to the resource is returned.
- The function then calls ReadAndReleaseViewResource with arguments of `gDesktop`, the supervisor passed to the function

(`this`), the handle to the view resource, the `kTCLViewWindowKind` constant (indicating this is a window and not a subview), and a pointer to a Point variable in which the called function returns the window position.

- The ReadAndReleaseViewResource function starts the process of parsing the view resource and creating the objects it specifies. The first action of the function is to call NewInputHandleStream, with the handle to the view resource, to create an input stream to read the contents of the view resource data, using its handle. That function can be found in the CHandleStream class.

- After the input handle stream has been created, the ReadAndReleaseViewResource function reads one byte from the stream. This first byte is the version of the Object I/O code that wrote the stream. The function then tests to ensure that the version is within the range of the minimum and maximum versions supported by the version of the Object I/O code being used to read the stream. If the test fails, then a failure is posted and the process terminates. If the view resource was written by a compatible version of the TCL, then the function reads one more byte and tests it to determine whether it matches the view type that is to be created (`kTCLViewWindowKind`). Once again, a failure condition is created if the view kind does not match the expected view kind. The next operation of the function is to call the Get function of the stream object and access the initial position of the window, storing it into the initialPosition argument to the ReadAndReleaseViewResource function. The next operation is to call the GetView function of the stream, with arguments of the enclosure (CDesktop) and the view's supervisor (`this`). That function call begins the process of accessing and creating all of the user interface objects associated with the view. The call returns a CView object, and then the function finishes its execution by deleting the stream object, releasing the view resource, and returning the CView object to its caller (TCLGetNamedWindow).

Creating and Initializing the CDialog Object

The process of accessing the object descriptions stored in the view resource is fairly complex. The values of member variables of the

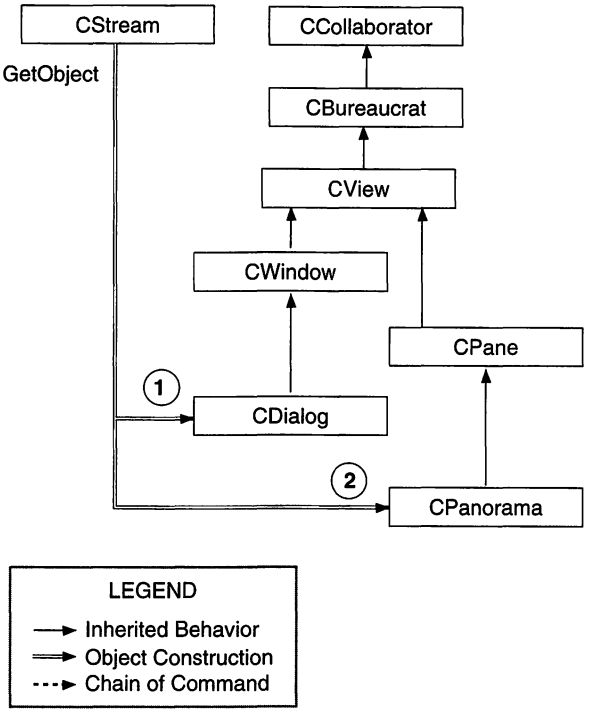

Figure 9-2
Creation of CCategories dialog object

objects, as well as their names and position information are stored in the resource. A schematic of the creation of the CCategories dialog object is shown in Figure 9-2. The circled numbers are object numbers in the stream's list of objects (`checklist`). The operation of the GetView function for the specific example of the Categories dialog is as follows:

- The GetView function is in the CStream class. The function begins by saving the enclosure object's pointer into a variable named `gIOEnclosure`, whose value, in this case, is the CDesktop object. The function then calls GetBureaucrat for the stream, returning the result as the requested view pointer.

- GetBureaucrat (also in the CStream class) saves the pointer to the view's supervisor (CMain in this case) in a variable named `gIOSupervisor` and then calls GetObject with a reference to the stream and a reference to a pointer to the specified object class (CCollaborator in this case). The GetObject function is in the **CStream.tem** file (it's a template-derived class). You will see that several classes are derived from these templates, including

CBitMap, CCollaborator, CEnvironment, CEnvirons and CPaneBorder. All of these are CStream-oriented classes whose template directives can be found in the files beginning with the name **CStream_** and ending with the **.cpp** extension. The contents of the file is as follows:

```
#include "CStream.h"
#include "CWindow.h"

#pragma template_access public

#pragma template PutObject(CStream&, CCollaborator*)
#pragma template GetObject(CStream&, CCollaborator*&)
#pragma template PutObject(CStream&, CView*)
#pragma template GetObject(CStream&, CView*&)
#pragma template PutObject(CStream&, CWindow*)
#pragma template GetObject(CStream&, CWindow*&)

#include "CStream.tem"
```

Notice in the foregoing that PutObject and GetObject functions are being created for the specified classes (CCollaborator, CView, and CWindow).

- The GetObject function begins by calling the GetClassName function of the CStream class. In the case of our example, the class name is CDialog. The name is stored as either a C or Pascal string in the stream. In either case, the GetClassName function returns the name as a C string, which the GetObject function uses in a call to the new_by_name function, creating a CDialog object (in this case) and casting it as a CCollaborator object. The newly constructed object's pointer is passed in a call to the stream's AddReference function, which keeps a list of objects that may be referenced by other objects in the stream. The CDialog object is the first object in the list for this stream.

- The GetObject function then calls the GetFrom function for the object that was just constructed (CDialog), passing it the stream as an argument. The GetFrom function of the CDialog class begins by accessing the `scrollable` (Boolean) value from the stream. This is `FALSE` for our Categories dialog. Then GetFrom calls the GetFrom function of CWindow (its base class). That function accesses the following member variable values from the stream:

  - `procID = 4`

- `theSizeRect = { 40, 40, 342, 512}`
- `floating = FALSE`
- `isColor = TRUE`
- `isModal = FALSE`
- `actClick = FALSE`
- `portRect = { 0, 0, 230, 221 }`
- `helpResID = 0`
- `visible = FALSE`
- `goAwayFlag = TRUE`
- `title = Categories`

Initializing the Dialog's CWindow Class Variables

Following the acquisition of the foregoing member variables from the stream, the GetFrom function of the CWindow class determines whether the `isColor` variable is `TRUE` and whether the user's machine has Color QuickDraw installed. Depending upon whether both of these are `TRUE`, the function calls the NewCWindow or NewWindow toolbox functions to create either a color or monochrome window with the `portRect`, `title`, `procID`, `goAwayFlag`, and an indication of whether the window should be in the front (in case the `floating` variable is `TRUE`) or behind all other windows (in case `floating` is `FALSE`). The window pointer is stored into the `macPort` variable of the CDialog object.

At this point, the GetFrom function of the CWindow class continues by setting the `windowKind` field of the `macPort` variable to `OBJ_WINDOW_KIND`, setting the current port to the `macPort` value, setting the CDialog object's `itsEnclosure` to the `gDesktop` object (CDesktop), calling the AddWind function of the CDesktop class to add the current window to its list, calling the window's SetSizeRect function with the `theSizeRect` value, and then, finally, calling the GetFrom function for the CView class (its base class).

Initializing the Dialog's CView Class Variables

Initialization of the CDialog (CCategories) object continues in the GetFrom function of the CView class. The purpose here is to

access the values that pertain to the CView base class from the stream and store these into the appropriate member variables of the object. The GetFrom function accesses the following member variable values from the stream:

- `visible = FALSE`
- `active = FALSE`
- `wantsClicks = TRUE`
- `canBeGopher = FALSE`
- `ID = 0`
- `usingLongCoord = FALSE`
- `(reserved) = 0`
- `helpResIndex = 0`
- `numSubviews = 1`

After accessing the foregoing values and storing them into the corresponding member variables, the function tests whether the value in the `itsEnclosure` variable is equal to the object pointer stored in the `gDesktop` variable (CDesktop). If not, then the AddSubview function of the object pointed to by the `itsEnclosure` variable is called to add the current object to the enclosure's list of subviews (The enclosure for the CCategories (CDialog) object is the CDesktop object, so the foregoing step is not taken for that object).

At this point the GetFrom function of the CView class calls the GetFrom function for the CBureaucrat class (its base class), which accesses a zero-length string as the name of the object's supervisor. After accessing one more byte and finding that it is also zero, a `NULL` pointer is returned as the object's supervisor. Therefore, the `itsSupervisor` member variable is set to the value of the `gIOSupervisor` variable (CMain, as described on page 347). The GetFrom function in the CView class continues by commencing a loop, whose number of iterations is specified by the `numSubviews` variable, accessing the stream to create new subviews by calling the GetObject function in the **CStream.tem** file.

Creating and Initializing the CPanorama Object

The single subview of the CDialog view is a CPanorama object. The GetObject function accesses that class name from the stream, constructs the object using the new_by_name function, and then calls the GetFrom function in the newly created CPanorama object. That function accesses the various values from the stream, storing them into corresponding member variables, as follows:

- `hScale = 1`
- `vScale = 1`
- `bounds.top = 0`
- `bounds.left = 0`
- `bounds.bottom = 342`
- `bounds.right = 512`
- `position.v = 0`
- `position.h = 0`

Following the acquisition of the foregoing values, the function calls the GetFrom function of the CPane class.

Initializing the CPanorama's CPane Class Variables

The GetFrom function of the CPane class accesses the stream to acquire the values for storage in various of its member variables, as follows:

- `width = 222`
- `height = 231`
- `hEncl = -1`
- `vEncl = -1`
- `hSizing = 5 (sizELASTIC)`
- `vSizing = 5 (sizELASTIC)`
- `printClip = 2 (clipPAGE)`
- `frame.top = -1`
- `frame.left = -1`
- `frame.bottom = 230`

- `frame.right = 221`
- `autoRefresh = TRUE`
- `flags = 3`

After the foregoing variable's values have been acquired, the `flags` variable is tested to determine whether it equals the value of the `disabled_flags` constant (1). Because it is not equal, the value of the `enabled` member variable is set to `TRUE`.

At this point in the GetFrom function in the CPane class, the code attempts to access the name of the enclosure from the stream by calling GetObject once again. Because the enclosure name is a zero-length string, the GetClassName function accesses the very next byte and finds that it is nonzero. This indicates that the following four-byte value is the reference number of the object in the stream's reference list (`checklist`). In this case, because the reference number is 1, the referenced object is CDialog. Therefore, the pointer to the CDialog object (CCategories) is stored into the `itsEnclosure` variable.

The GetFrom function in the CPane class continues by calling the SubpaneLocation function for the object whose pointer is stored in the `itsEnclosure` variable. The `hEncl` and `vEncl` values are passed to the function, which calculates `hloc` and `vloc` positions for the CPanorama subview, converted to window coordinates. Then the `hOrigin` and `vOrigin` values are calculated and CalcAperture is called to calculate the CPanorama's aperture. Then the GetFrom function of the CView class (its base class) is called to continue the initialization process.

Initializing the Panorama's CView Class Variables

The initialization sequence for the CView class was described earlier, beginning on page 349. The process for parsing the stream for the member variables of the CPanorama object follows the same logic; however, the values obtained from the stream, and the variables in which they are stored, are as follows:

- `visible = TRUE`
- `active = TRUE`
- `wantsClicks = TRUE`
- `canBeGopher = FALSE`

- `ID = 0`
- `usingLongCoord = FALSE`
- `(reserved) = 0`
- `helpResIndex = 0`
- `numSubviews = 5`

As was described in the earlier narrative for the GetFrom function of the CView class, the function tests whether the object's enclosure is the CDesktop object, and the function calls AddSubview to add the new object (CPanorama) to its list of subviews. Then the function calls the GetFrom function of the CBureaucrat class.

The GetFrom function of the CBureaucrat class attempts to acquire a pointer to the CPanorama object's supervisor (`itsSupervisor`) by accessing its name from the stream. Instead, it retrieves a reference to the first object in the stream's list of objects (`checklist`) and returns it (the CDialog object) as the CPanorama object's supervisor. Following that, the GetFrom function of the CCollaborator class is called, which merely initializes the values of `itsProviders` and `itsDependents` to `NULL` and then returns control to the GetFrom function in the CBureaucrat class. That function returns to the GetFrom function of the CView class, which continues by starting a loop, whose number of iterations is set to the value contained in the `numSubviews` variable, and then calls the GetObject function for each of the subviews. The dialog's CPanorama object has five subviews.

Creating and Initializing the CScrollPane Object

The next object in the '`CVue`' resource for the Categories dialog is a CScrollPane. This is the scrollpane for the table of categories, but we will come to that shortly. A schematic diagram that shows the creation of the CScrollPane, its CColorTextEnvirons, and its CPaneBorder objects is shown in Figure 9-3. After the CScrollPane object has been created by calling the new_by_name function, the object is added to the stream's list of object pointers (`checklist`), and then the GetFrom function for the object is called to initialize the object. The GetFrom function for the CScrollPane class accesses the stream to obtain various values, which it stores into member variables. The member variables and the values that are assigned for this example are as follows:

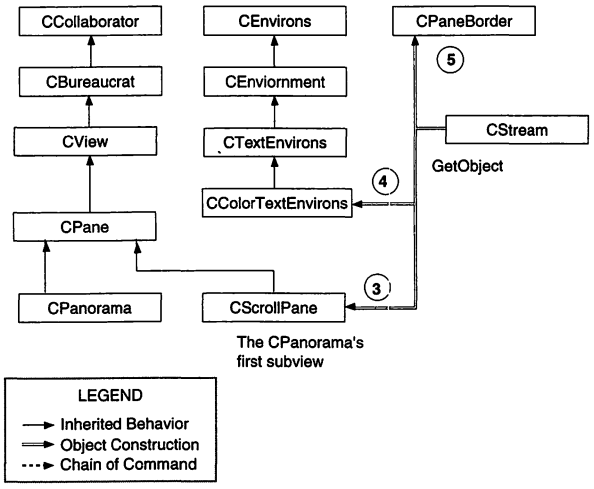

Figure 9-3
Creation of the CScrollPane and its associated objects

- `hasHoriz = FALSE`
- `hasVert = TRUE`
- `hasSizeBox = FALSE`
- `useSICN = FALSE`
- `hStep = 50`
- `vStep = 18`
- `hOverlap = 50`
- `vOverlap = 18`

At this point, the GetFrom function calls the GetFrom function of the CPane class (its base class).

Initializing the Scroll Pane's CPane Class Variables

As was described earlier (see page 351), the GetFrom function in the CPane class accesses the stream to acquire a number of values that it stores in member variables for the object. The member variables and their contents for this example are as follows:

- `width = 212`
- `height = 188`
- `hEncl = 4`
- `vEncl = 4`

- `hSizing = 4(sizFIXEDSTICKY)`
- `vSizing = 4(sizFIXEDSTICKY)`
- `printClip = 1(clipFRAME)`
- `frame.top = 0`
- `frame.left = 0`
- `frame.bottom = 188`
- `frame.right = 212`
- `autoRefresh = TRUE`
- `flags = 3`

At this point, the GetFrom function of the CPane class calls the GetObject function, with the stream, to access the CScrollPane object's enclosure pointer. The 'CVue' resource contains an object reference, rather than an object name, for this object. The reference is to the second object in the stream's `checklist`. That object is the CPanorama. So the CPanorama object's pointer is used as the value for the CScrollPane's `itsEnclosure` member variable. The SubpaneLocation function is called to compute the location of the CScrollpane within its enclosure, and the values of its top and left positions within the enclosure (in window coordinates) are returned in the `hloc` and `vloc` arguments to the function. Then the aperture of the CScrollpane object is computed by calling CalcAperture. With those tasks accomplished, the GetFrom function of the CPane class calls the GetFrom function of the CView class (its base class) to access additional values from the stream.

Initializing the Scroll Pane's CView Class Variables

As for the CDialog and CPanorama objects, the member variables associated with the CView class follow those for the CPane class in the stream. The member variables and their values for the example CScrollPane object are as follows:

- `visible = TRUE`
- `active = TRUE`
- `wantsClicks = TRUE`
- `canBeGopher = FALSE`
- `ID = 0`

- `usingLongCoord = FALSE`
- `(reserved) = 0`
- `helpResIndex = 0`
- `numSubviews = 0`

At this point, the GetFrom function for the CView class tests whether the CScrollPane object's enclosure is the CDesktop object. Because this is `FALSE`, the port for the CScrollPane object (`macPort`) is set to the port of its enclosure object (CPanorama), and the AddSubview function for its enclosure class is called to add the CScrollPane object as a subview of that object. Then the GetFrom function calls the GetFrom function of the CBureaucrat class (its base class) to access additional member variable values from the stream.

The GetFrom function of the CBureaucrat class attempts to get a pointer for the CScrollPane object's supervisor (`itsSupervisor`) by accessing its name from the stream. Instead, it retrieves a reference to the second object in the stream's list of objects (`checklist`) and returns it (the CPanorama object) as the CScrollPane object's supervisor. Following that, the GetFrom function of the CCollaborator class is called, which merely initializes the values of `itsProviders` and `itsDependents` to `NULL` and then returns control to the GetFrom function in the CBureaucrat class. That function returns to the GetFrom function of the CView class, which continues by attempting to create all of the subviews of the object, in a loop. Because the `numSubviews` value is 0 for the CScrollPane object, the loop exits immediately, causing the GetFrom function of the CView class to return to the GetFrom function of the CPane class.

Creating and Initializing the CColorTextEnvirons Object

When the GetFrom function of the CView class returns to the GetFrom function of the CPane class, that function continues execution by calling the GetObject function to access the object that establishes the value of the `itsEnvironment` member variable for the CScrollPane object. In the case of the Categories dialog example, the class name next in the stream is CColorTextEnvirons.

The CColorTextEnvirons object establishes the text environment for the CScrollPane object. After the CColorTextEnvirons object

has been created and added to the stream's list of objects (`checklist`), the GetFrom function for that object is called to access a number of values that are placed into variables associated with the object. The GetFrom function accesses the stream to acquire the values for various member variables that are stored in the form of a structure whose type is SaveColorTextEnvirons. The various fields of the structure and the values obtained are as follows:

- `foreColor = { 0, 0, 0 }`
- `backColor = { 65535, 65535, 65535 }`
- `penSize = { 1, 1 }`
- `changeForeColor = FALSE`
- `changeBackColor = FALSE`
- `changeForePattern = FALSE`
- `changeBackPattern = FALSE`
- `resPatterns = TRUE`
- `penMode = 8 (patCopy)`
- `(reserved) = 0`

After the foregoing are retrieved, they are stored in member variables of the same names as the fields. Because the `resPatterns` variable is `TRUE`, the GetFrom function continues by accessing the following variables from the stream:

- `forePatID = 0`
- `backPatID = 0`
- `forePatInd = 0`
- `backPatInd = 0`

Then the GetFrom function of the CColorTextEnvirons class completes its execution by calling the GetFrom function of the CTextEnvirons class.

Initializing the CTextEnvirons Class Variables

The GetFrom function of the CTextEnvirons class accesses various values from the stream and places them into member vari-

ables. The variables and their values for this example are as follows:

- `textInfo.theSize = 12`
- `textInfo.theStyle = 0 (normal)`
- `textInfo.theMode = 0 (srcCopy)`
- `fontName = Chicago`

The value in the `fontName` variable is used to look up the corresponding font number by calling the GetFNum toolbox function, and the number is placed into the `textInfo.fontNumber` member variable for the object. The GetFrom function then calls the GetFrom function in the CEnvironment class (its base class) to access additional values from the stream.

Initializing the CEnvirons Class Variables

The CEnvironment class has no data members, so the GetFrom member function in the CEnvirons class (its base class) is called instead. That function accesses the value of the `moreEnvirons` pointer from the stream, which is `NULL` in this case. Execution continues in the GetFrom function of the CPane class, after the environment data has been accessed.

Creating and Initializing the CPaneBorder Object

After the GetFrom function of the CColorTextEnvirons class returns to the GetFrom function in the CPane class, that function calls GetObject to acquire the pointer to the `itsBorder` object. The GetObject function accesses the stream to find an object name of CPaneBorder in this case. After the CPaneBorder object is constructed and added to the stream's list of objects (`checklist`), the GetFrom function for the CPaneBorder object is called. That function accesses the stream to acquire a number of values that are placed into the object's member variables. The variables and values for this example are as follows:

- `borderFlags = 0x0000000F`
- `borderPen = { 1, 1 }`
- `shadowOffset = { 2, 2 }`
- `shadowPen = { 0, 0 }`
- `roundDiameter = { 16, 16 }`

- `penPat = 0xFFFFFFFF FFFFFFFF`
- `margin = { 1, 1, -1, -1 }`

After filling in the values of the foregoing variables, the GetFrom function of the CPaneBorder object returns to the GetFrom function of the CPane object. At this point, execution continues in the GetFrom function of the CScrollPane object. That function continues by calling GetObject to access the panorama object that is controlled by the scroll pane. In this case, the panorama object is a subclass of CArrayPane (CCatTable), whose construction is described in the next section. After the CCatTable object has been constructed, the GetFrom function in the CScrollPane class finishes by installing the CCatTable object as the scroll pane's panorama. Then the loop in the CView class continues by accessing the stream to create the first subview of the dialog's CPanorama object.

Creating and Initializing the CCatTable Object

The CCatTable object was created in the VA as a derived class of the CArrayPane class. A schematic of the recreation of this, its CRunArray, and CTextEnvirons objects is shown in Figure 9-4. After the CCatTable object has been created in the GetObject function, its GetFrom function is called. That function merely calls the GetFrom function of the CArrayPane class (its base class).

Initializing the CCatTable's CArrayPane Class Variables

The GetFrom function in the CArrayPane class accesses the stream to acquire the value of the `ownsArray` variable, which is found to be `FALSE`. The function continues by calling the GetFrom function of the CTable class (its base class) to access additional values from the stream. The first set of variables is accessed in the form of a structure, whose fields and the corresponding values are as follows:

- `numRows = 0`
- `numCols = 1`
- `topLeftIndent = { 0, 0 }`
- `drawOrder = 0 (tblRow)`
- `defRowHeight = 18`

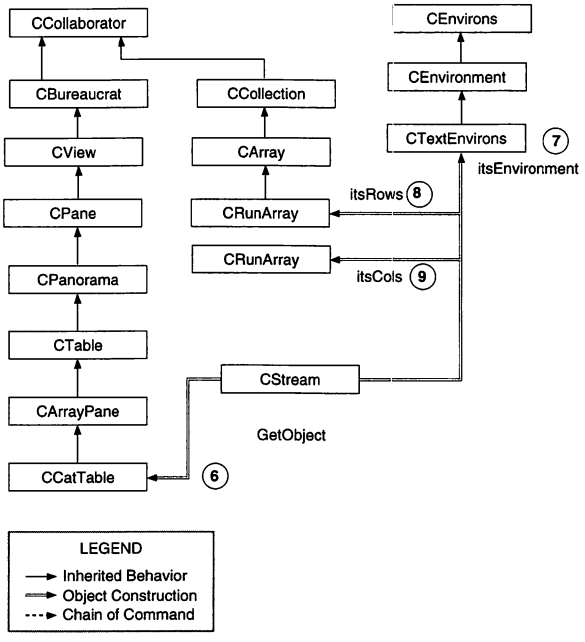

Figure 9-4
Creation of the CCatTable and its associated objects

- `defColWidth = 50`
- `selectionFlags = 0x00000001`
- `indent = { 13, 3 }`
- `dblClickCmd = 512`
- `drawActiveBorder = FALSE`
- `clipToCells = FALSE`
- `rowBorders` = (a structure with fields as follows)
  - `thickness = 0`
  - `penMode = 0 (patCopy)`
  - `pat = 0xFFFFFFFF FFFFFFFF`
- `colBorders` = (a structure with fields as follows)
  - `thickness = 0`
  - `penMode = 0 (patCopy)`
  - `pat = 0xFFFFFFFF FFFFFFFF`

After the foregoing structure has been read from the stream, the fields are stored into correspondingly named member variables in the object, and then the GetFrom function of the CPanorama class (its base class) is called.

Initializing the CCatTable's CPanorama Class Variables

The GetFrom function in the CPanorama class was described beginning on page 351. That function accesses the stream to acquire various values that are stored into the object's member variables, as follows:

- `hScale = 1`
- `vScale = 1`
- `bounds.top = 0`
- `bounds.left = 0`
- `bounds.bottom = 0`
- `bounds.right = 0`
- `position.v = 0`
- `position.h = 0`

Following the initialization of the foregoing member variables, the GetFrom function of the CPane class is called.

Initializing the CCatTable's CPane Class Variables

The GetFrom function in the CPane class accesses a number of values from the stream that it places in member variables for the object, as follows:

- `width = 195`
- `height = 186`
- `hEncl = 1`
- `vEncl = 1`
- `hSizing = 5 (sizELASTIC)`
- `vSizing = 5 (sizELASTIC)`
- `printClip = 2 (clipPAGE)`
- `frame.top = 0`
- `frame.left = 0`

- `frame.bottom = 186`
- `frame.right = 195`
- `autoRefresh = TRUE`
- `flags = 3`

After the foregoing values have been stored into the CCatTable's member variables, the GetFrom function in the CPane class accesses the stream to acquire the pointer to the table's enclosure. In this case, the stream contains a reference to the third object (the CScrollPane object) in the stream's `checklist`. The CCatTable object is installed in the CScrollPane object's enclosure when the GetFrom function calls the SubpaneLocation function, returning the `hloc` and `vloc` values that establish its horizontal and vertical position (in window coordinates). The table's aperture is calculated with a call to CalcAperture and then the GetFrom function of the CView class (its base class) is called.

Initializing the CCatTable's CView Class Variables

The member variables of the CView class are the same as what was presented earlier for other objects. The values for the variables for the CCatTable object are as follows:

- `visible = TRUE`
- `active = TRUE`
- `wantsClicks = TRUE`
- `canBeGopher = TRUE`
- `ID = 2049`
- `usingLongCoord = TRUE`
- `(reserved) = 0`
- `helpResIndex = 0`
- `numSubviews = 0`

At this point, the GetFrom function for the CView class tests whether the CScrollPane object's enclosure is the CDesktop object. Because this is `FALSE`, the port for the CCatTable object (`macPort`) is set to the port of its enclosure object (CPanorama), and the AddSubview function for its enclosure class is called to add the CCatTable object as a subview of that object. Then the

function calls the GetFrom function of the CBureaucrat class (its base class) to access additional member variable values from the stream.

Initializing the CCatTable's CBureaucrat Class Variables

The GetFrom function of the CBureaucrat class first accesses the stream to determine the object pointer for the CCatTable object's supervisor. In the case of this example, the `itsSupervisor` pointer is stored as a reference to the CPanorama object. The next step of the GetFrom function is to call the GetFrom function of the CCollaborator class, which merely sets the values of `itsProviders` and `itsDependents` list pointers to `NULL` values.

Execution continues in the GetFrom function of the CView class. Because the CCatTable object has no subviews, the loop in that function is skipped and the `itsSubviews` list pointer is set to `NULL`. Then execution continues in the GetFrom function of the CPane class.

Creating and Initializing the CTextEnvirons Object

When the GetFrom function in the CView class returns to the GetFrom function in the CPane class, that function calls the GetObject function in the CStream class to create what will become the CCatTable's `itsEnvironment` object pointer. In the case of our example, the VA has written the CTextEnvirons name into the stream, and so an object of that class is created. After being created, the CTextEnvirons object is added to the stream's list of objects (`checklist`) and then its GetFrom function is called.

The GetFrom function of the CTextEnvirons class accesses the stream to acquire the values of various member variables. The variables and their values, for our example, are as follows:

- `textInfo.theSize = 12`
- `textInfo.theStyle = 0 (normal)`
- `textInfo.theMode = 1(srcOr)`
- `fontName = Geneva`

After the foregoing values have been extracted from the stream, the GetFrom function for the CTextEnvirons class uses the `fontName` string in a call to the GetFNum toolbox function to get the font number for that typeface. After that, the GetFrom

function calls the GetFrom function in the CEnvironment class (its base class). Because the CEnvironment class has no GetFrom function, the one inherited from the CEnvirons class is called.

The GetFrom function of the CEnvirons class accesses the stream to acquire the value of the `moreEnvirons` object pointer. This is a `NULL` value in the stream. After this, execution continues in the GetFrom function of the CPane class, where the GetObject function is called to create the `itsBorder` object for the CCatTable object. This object is also `NULL`, so execution continues in the GetFrom function of the CTable class, where the GetObject function is called to access the `itsRows` object. In this case, a CRunArray object is created.

Creating and Initializing the CRunArray Object

The CRunArray object for the CCatTable's `itsRows` variable is used to contain the heights of all of the rows in the table. The positive attributes of the CRunArray object were discussed in Chapter 8, where it was shown that these objects minimize the amount of array storage needed to represent "runs" of identical values. After the CRunArray object is created, it is added to the stream's list of objects (`checklist`), and then its GetFrom function is called to initialize the object.

The GetFrom function of the CRunArray class accesses the stream to acquire the value of the `itemCount` variable. This value is 0 (no rows) in our example. The function then calls the GetFrom function of the CArray class (its base class).

Initializing the CRunArray's CArray Class Variables

The GetFrom function in the CArray class accesses the stream to acquire the values for several of the CRunArray object's member variables. The variables and their values for our example dialog are as follows:

- `blockSize = 3`
- `elementSize = 8`

After the foregoing values have been accessed and stored in member variables, the GetFrom function for the CArray class calls the GetFrom function in the CCollection class (its base class).

Initializing the CRunArray's CCollection Class Variables

The GetFrom function in the CCollection class accesses the stream to acquire the value of the `numItems` variable, whose value is 0 for our example. Execution continues in the GetFrom function in the CArray class.

Continuing the CArray Class Variable Initialization

When control returns to the GetFrom function in the CArray class, that function calls the GetItems function in the same class to complete the initialization process. The GetItems function accesses the stream to acquire the value of the `slots` variable, which is 0 for our example. Then the GetHandle function of the CStream class is called to access the handle to be stored into the `hItems` variable. In this case, the GetHandle function accesses the stream to acquire the handle. In the process of doing so, the handle length is acquired, the handle is created, and then the contents of the handle are read from the stream. The length and contents are as follows:

- `len = 8`
- `h = {0, 0}`

The values in the handle 'h' consist of two long (4-byte) fields of a single structure. The first field is the length of the run and the second field is the value of the run. Because the value of the `itemCount` variable is 0 for this object, the foregoing handle's contents are irrelevant. After the foregoing have been accessed and stored, execution continues in the GetFrom function of the CTable class, where it calls the GetObject function to create the object whose pointer is stored into the `itsCols` variable of the CCatTable object.

Creating and Initializing the CRunArray Object

As was the case with the CRunArray object that specifies "runs" of row heights, the new CRunArray object is created to hold "runs" of column widths in the table. After the object has been created and added to the stream's list of objects (`checklist`), the object's GetFrom function is called.

The GetFrom function of the CRunArray class accesses the stream to acquire the value of the `itemCount` variable, which is 1 (one column) in our example. Following this, the function

calls the GetFrom function of the CArray class to access additional values from the stream.

Initializing the CRunArray's CArray Class Variables

The GetFrom function in the CArray class accesses the stream to acquire the values for several of the CRunArray object's member variables. The variables and their values for our example dialog are as follows:

- `blockSize = 3`
- `elementSize = 8`

After the foregoing values have been accessed and stored in member variables, the GetFrom function for the CArray class calls the GetFrom function in the CCollection class (its base class).

Initializing the CRunArray's CCollection Class Variables

The GetFrom function in the CCollection class accesses the stream to acquire the value of the `numItems` variable, whose value is 1 for our example. Execution continues in the GetFrom function in the CArray class.

Continuing the CArray Class Variable Initialization

When control returns to the GetFrom function in the CArray class, that function calls the GetItems function in the same class to complete the initialization process. The GetItems function accesses the stream to acquire the value of the `slots` variable, which is 3 for our example (because the array contains one column and the `blockSize` variable has a value of 3, three array slots are allocated in this case). Then the GetHandle function of the CStream class is called to access the handle to be stored into the `hItems` variable. In this case, the GetHandle function accesses the stream to acquire the handle. In the process of doing so, the handle length is acquired, the handle is created, and then the contents of the handle are read from the stream. The length and contents are as follows:

- `len = 32`
- `h = {1, 195L}, {0, 0}, {0, 0}, {0, 0}`

The values in the handle 'h' consist of two long (4-byte) fields of a single structure. The first field is the length of the run and the second field is the value of the run. Because the value of the `item-`

`Count` variable is 1 for our example, only the first entry in the foregoing handle is relevant. The remaining three entries can be ignored and are allocated merely to fill the array's slots. After the foregoing have been accessed and stored, execution continues in the GetFrom function of the CArrayPane class.

Continuing the Initialization of the CArrayPane Variables

When the GetFrom function of the CTable class returns to the GetFrom function in the CArrayPane class, that function calls the GetObject function to access the object whose pointer will be stored into the `itsArray` member variable of the CCatTable object. The value returned by the GetObject function, for this example, is a `NULL` pointer. In our application, the array is created in the ICMain function of the CMain class and is installed in the BeginData function of the CCategories class (see page 229).

After the CCatTable and its associated objects have been fully constructed, execution continues in the GetFrom function of the CScrollPane class to install the CCatTable object as the CScrollPane's panorama. Control of the execution then returns to the GetFrom function in the CView class to access the CPanorama object's next (second) subview.

Creating and Initializing the CButton Objects

The GetFrom function of the CView class continues its loop, calling the GetObject function in the CStream class to create the remaining subviews of the CPanorama object. In the case of our example, these are the four CButton objects (Use, Edit, New, and Delete). Each of these has an associated CColorTextEnvirons object, and the first of them also has a CPaneBorder object to serve as the default button for the dialog. A schematic of the construction of the remaining four subviews is shown in Figure 9-5. The first CButton object is labeled with a circled number 10. Its CColorTextEnvirons object is number 11, and its CPaneBorder object is number 12. This is the button named "Use" in the dialog. The remaining CButton objects (Edit, New, and Delete) and their associated CColorTextEnvirons objects are numbered in pairs: 13 and 14, 15 and 16, and then 17 and 18.

The GetObject function creates the first CButton object, adds it to the stream's list of objects (`checklist`), and then calls the GetFrom function for the object. That function accesses the

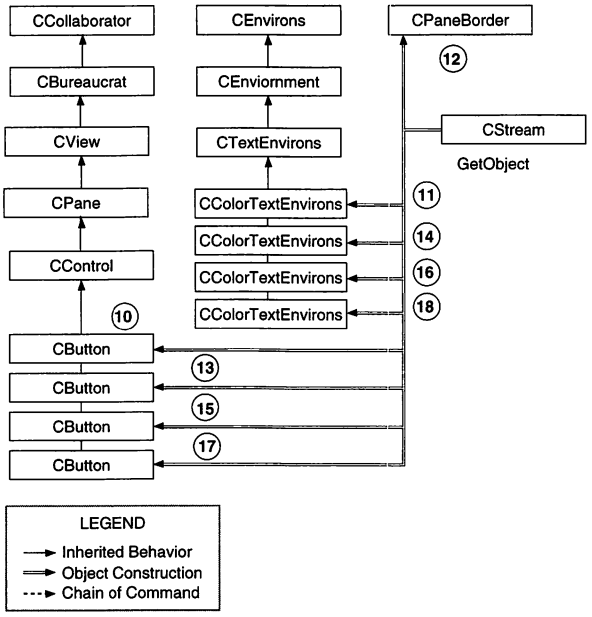

Figure 9-5
Creation of the CButton and associated objects

stream to acquire the values for two member variables of the object, as follows:

- `procID = 8`
- `clickCmd = 512 (cmdUseCat)`

After the foregoing values have been stored, the function calls the GetControl function that it inherits from the CControl class. When that function returns, the NewControl toolbox function is called to create the button using the values of the button's member variables.

Initializing the Use Button's CControl Class Variables

The GetControl function of the CControl class accesses the stream to get the values of four variables, as follows:

- `value = 0`
- `cmin = 0`
- `cmax = 1`
- `title = Use`

Following the acquisition of the foregoing values, the GetControl function calls the GetFrom function of the CPane class.

Initializing the Use Button's CPane Class Variables

The GetFrom function in the CPane class has been described for many other objects. In the case of the Use button, the values accessed from the stream are as follows:

- `width = 47`
- `height = 16`
- `hEncl = 9`
- `vEncl = 205`
- `hSizing = 4(sizFIXEDSTICKY)`
- `vSizing = 4(sizFIXEDSTICKY)`
- `printClip = 1(clipFRAME)`
- `frame.top = 0`
- `frame.left = 0`
- `frame.bottom = 16`
- `frame.right = 47`
- `autoRefresh = TRUE`
- `flags = 3`

After the foregoing values have been stored into the Use button's member variables, the GetFrom function in the CPane class accesses the stream to acquire the pointer to the button's enclosure. In this case, the stream contains a reference to the second object (the CPanorama object) in the stream's `checklist`. The function continues by calling the SubpaneLocation and GetAperture functions. Then the function calls the GetFrom function in the CView class.

Initializing the Use Button's CView Class Variables

The GetFrom function of the CView class has been discussed previously. The member variables and the values accessed from the stream for the Use button are as follows:

- `visible = TRUE`
- `active = FALSE`
- `wantsClicks = TRUE`
- `canBeGopher = FALSE`

- ID = 2050
- usingLongCoord = FALSE
- (reserved) = 0
- helpResIndex = 0
- numSubviews = 0

After the foregoing values have been acquired, the port for the view is set to the port of its enclosure (CPanorama), and the view is added to its enclosure's list of subviews (itsSubviews). The fact that the value of numSubviews variable is 0 causes the loop to be skipped and the GetFrom function of the CBureaucrat class (its base class) to be called.

Initializing the Use Button's CBureaucrat Class Variables

The GetFrom function of the CBureaucrat class accesses the stream to get the object which is to be assigned as the button's supervisor. In the case of the Use button, the itsSupervisor is set to a pointer to the CPanorama object (a reference in the stream). Following this, the GetFrom function of the CCollaborator class is called. That function merely sets the values of the itsProviders and itsDependents lists to NULL pointers.

After the foregoing operations are complete, execution control returns to the GetFrom function in the CPane class, where GetObject is called to create the object that is to serve as the Use button's environment (itsEnvironment).

Creating and Initializing the CColorTextEnvirons Object

The CColorTextEnvirons object has been discussed previously. The values of the member variables that apply to the Use button's environment are as follows:

- foreColor = { 0, 0, 0 }
- backColor = { 65535, 65535, 65535 }
- penSize = { 1, 1 }
- changeForeColor = FALSE
- changeBackColor = FALSE
- changeForePattern = FALSE

- `changeBackPattern = FALSE`
- `resPatterns = TRUE`
- `penMode = 8 (patCopy)`
- `(reserved) = 0`

After the foregoing are retrieved, they are stored in member variables of the same names as the fields. Because the `resPatterns` variable is `TRUE`, the GetFrom function continues by accessing the following variables from the stream:

- `forePatID = 0`
- `backPatID = 0`
- `forePatInd = 0`
- `backPatInd = 0`

Then the GetFrom function of the CColorTextEnvirons class completes its execution by calling the GetFrom function of the CTextEnvirons class.

Initializing the CTextEnvirons Class Variables

The GetFrom function of the CTextEnvirons class accesses various values from the stream and places them into member variables. The variables and their values for this example are as follows:

- `textInfo.theSize = 10`
- `textInfo.theStyle = 0 (normal)`
- `textInfo.theMode = 0 (srcCopy)`
- `fontName = Chicago`

The value in the `fontName` variable is used to look up the corresponding font number by calling the GetFNum toolbox function, and the number is placed into the `textInfo.fontNumber` member variable for the object. The GetFrom function then calls the GetFrom function in the CEnvironment class (its base class) to access additional values from the stream.

Initializing the CEnvirons Class Variables

The CEnvironment class has no data members, so the GetFrom member function in the CEnvirons class (its base class) is called

instead. That function accesses the value of the `moreEnvirons` pointer from the stream, which is `NULL` in this case. Execution continues in the GetFrom function of the CPane class, after the environment data has been accessed.

Creating and Initializing the CPaneBorder Object

After the GetFrom function of the CColorTextEnvirons class returns to the GetFrom function in the CPane class, that function calls GetObject to acquire the pointer to the `itsBorder` object. The GetObject function accesses the stream to find an object name of CPaneBorder in this case. After the CPaneBorder object is constructed and added to the stream's list of objects (`checklist`), the GetFrom function for the CPaneBorder object is called. That function accesses the stream to acquire a number of values that are placed into the object's member variables. The variables and values for this example are as follows:

- `borderFlags = 0x00000020`
- `borderPen = { 3, 3}`
- `shadowOffset = { 2, 2 }`
- `shadowPen = { 0, 0 }`
- `roundDiameter = { 16, 16 }`
- `penPat = 0xFFFFFFFF FFFFFFFF`
- `margin = { -1, -1, 1, 1 }`

After filling in the values of the foregoing variables, the GetFrom function of the CPaneBorder object returns to the GetFrom function of the CPane object. At this point, the GetFrom function of the CPane class has completed its task of creating the Use button object, and the loop in the CView class continues by accessing the stream to create the next subview of the CPanorama object.

Creating and Initializing the Edit CButton Object

The GetObject function is called inside the GetFrom function of the CView class to access and create the next subview of the CPanorama object. This is the Edit button shown in Figure 9-1, and whose components are shown in Figure 9-5. The values for the CButton class variables are as follows:

- `procID = 8`

- `clickCmd = 514 (cmdEditCat)`

After the foregoing values have been stored, the function calls the GetControl function that it inherits from the CControl class. When that function returns, the NewControl toolbox function is called to create the button using the values of the button's member variables.

Initializing the Edit Button's CControl Class Variables

The GetControl function of the CControl class accesses the stream to get the values of four variables, as follows:

- `value = 0`
- `cmin = 0`
- `cmax = 1`
- `title = Edit…`

Notice the ellipsis character at the end of the button's title. Following the acquisition of the foregoing values, the GetControl function calls the GetFrom function of the CPane class.

Initializing the Edit Button's CPane Class Variables

The GetFrom function in the CPane class has been described for many other objects. In the case of the Use button, the values accessed from the stream are as follows:

- `width = 47`
- `height = 16`
- `hEncl = 65`
- `vEncl = 205`
- `hSizing = 4(sizFIXEDSTICKY)`
- `vSizing = 4(sizFIXEDSTICKY)`
- `printClip = 1(clipFRAME)`
- `frame.top = 0`
- `frame.left = 0`
- `frame.bottom = 16`
- `frame.right = 47`
- `autoRefresh = TRUE`

- `flags = 3`

After the foregoing values have been stored into the Edit button's member variables, the GetFrom function in the CPane class accesses the stream to acquire the pointer to the button's enclosure. In this case, the stream contains a reference to the second object (the CPanorama object) in the stream's `checklist`. The function continues by calling the SubpaneLocation and GetAperture functions. Then the function calls the GetFrom function in the CView class.

Initializing the Edit Button's CView Class Variables

The GetFrom function of the CView class has been discussed previously. The member variables and the values accessed from the stream for the Edit button are as follows:

- `visible = TRUE`
- `active = FALSE`
- `wantsClicks = TRUE`
- `canBeGopher = FALSE`
- `ID = 2051`
- `usingLongCoord = FALSE`
- `(reserved) = 0`
- `helpResIndex = 0`
- `numSubviews = 0`

After the foregoing values have been acquired, the port for the view is set to the port of its enclosure (CPanorama), and the view is added to its enclosure's list of subviews (`itsSubviews`). The fact that the value of `numSubviews` variable is 0 causes the loop to be skipped and the GetFrom function of the CBureaucrat class (its base class) to be called.

Initializing the Edit Button's CBureaucrat Class Variables

The GetFrom function of the CBureaucrat class accesses the stream to get the object which is to be assigned as the button's supervisor. In the case of the Edit button, the `itsSupervisor` is set to a pointer to the CPanorama object (a reference in the stream). Following this, the GetFrom function of the CCollabora-

tor class is called. That function merely sets the values of the `itsProviders` and `itsDependents` lists to `NULL` pointers.

After the foregoing operations are complete, execution control returns to the GetFrom function in the CPane class, where GetObject is called to create the object that is to serve as the Edit button's environment (`itsEnvironment`).

Creating and Initializing the CColorTextEnvirons Object

The CColorTextEnvirons object has been discussed previously. The values of the member variables that apply to the Edit button's environment are as follows:

- `foreColor = { 0, 0, 0 }`
- `backColor = { 65535, 65535, 65535 }`
- `penSize = { 1, 1 }`
- `changeForeColor = FALSE`
- `changeBackColor = FALSE`
- `changeForePattern = FALSE`
- `changeBackPattern = FALSE`
- `resPatterns = TRUE`
- `penMode = 8 (patCopy)`
- `(reserved) = 0`

After the foregoing are retrieved, they are stored in member variables of the same names as the fields. Because the `resPatterns` variable is `TRUE`, the GetFrom function continues by accessing the following variables from the stream:

- `forePatID = 0`
- `backPatID = 0`
- `forePatInd = 0`
- `backPatInd = 0`

The GetFrom function of the CColorTextEnvirons class then completes its execution by calling the GetFrom function of the CTextEnvirons class.

Initializing the CTextEnvirons Class Variables

The GetFrom function of the CTextEnvirons class accesses various values from the stream and places them into member variables. The variables and their values for this example are as follows:

- `textInfo.theSize = 10`
- `textInfo.theStyle = 0 (normal)`
- `textInfo.theMode = 0 (srcCopy)`
- `fontName = Chicago`

The value in the `fontName` variable is used to look up the corresponding font number by calling the GetFNum toolbox function, and the number is placed into the `textInfo.fontNumber` member variable for the object. The GetFrom function then calls the GetFrom function in the CEnvironment class (its base class) to access additional values from the stream.

Initializing the CEnvirons Class Variables

The CEnvironment class has no data members, so the GetFrom member function in the CEnvirons class (its base class) is called instead. That function accesses the value of the `moreEnvirons` pointer from the stream, which is `NULL` in this case. Execution continues in the GetFrom function of the CPane class, after the environment data has been accessed.

Continuing the Edit Button's CPane Class Initialization

The Edit button has no associated border object, so the object returned by GetObject in the GetFrom function of the CPane class returns a `NULL` pointer for the `itsBorder` member variable.

Creating and Initializing the New CButton Object

The GetObject function is called inside the GetFrom function of the CView class to access and create the next subview of the CPanorama object. This is the New button shown in Figure 9-1, whose components are shown in Figure 9-5. The values for the CButton class variables are as follows:

- `procID = 8`
- `clickCmd = 516 (cmdNewCat)`

After the foregoing values have been stored, the function calls the GetControl function that it inherits from the CControl class. When that function returns, the NewControl toolbox function is called to create the button using the values of the button's member variables.

Initializing the New Button's CControl Class Variables

The GetControl function of the CControl class accesses the stream to get the values of four variables, as follows:

- `value = 0`
- `cmin = 0`
- `cmax = 1`
- `title = New...`

Notice the ellipsis character at the end of the button's title. Following the acquisition of the foregoing values, the GetControl function calls the GetFrom function of the CPane class.

Initializing the New Button's CPane Class Variables

The GetFrom function in the CPane class has been described for many other objects. In the case of the Use button, the values accessed from the stream are as follows:

- `width = 47`
- `height = 16`
- `hEncl = 118`
- `vEncl = 205`
- `hSizing = 4(sizFIXEDSTICKY)`
- `vSizing = 4(sizFIXEDSTICKY)`
- `printClip = 1(clipFRAME)`
- `frame.top = 0`
- `frame.left = 0`
- `frame.bottom = 16`
- `frame.right = 47`
- `autoRefresh = TRUE`
- `flags = 3`

After the foregoing values have been stored into the New button's member variables, the GetFrom function in the CPane class accesses the stream to acquire the pointer to the button's enclosure. In this case, the stream contains a reference to the second object (the CPanorama object) in the stream's `checklist`. The function continues by calling the SubpaneLocation and GetAperture functions. Then the function calls the GetFrom function in the CView class.

Initializing the New Button's CView Class Variables

The GetFrom function of the CView class has been discussed previously. The member variables and the values accessed from the stream for the New button are as follows:

- `visible = TRUE`
- `active = TRUE`
- `wantsClicks = TRUE`
- `canBeGopher = FALSE`
- `ID = 2052`
- `usingLongCoord = FALSE`
- `(reserved) = 0`
- `helpResIndex = 0`
- `numSubviews = 0`

After the foregoing values have been acquired, the port for the view is set to the port of its enclosure (CPanorama) and the view is added to its enclosure's list of subviews (`itsSubviews`). The fact that the value of `numSubviews` variable is 0 causes the loop to be skipped and the GetFrom function of the CBureaucrat class (its base class) to be called.

Initializing the New Button's CBureaucrat Class Variables

The GetFrom function of the CBureaucrat class accesses the stream to get the object that is to be assigned as the button's supervisor. In the case of the New button, the `itsSupervisor` is set to a pointer to the CPanorama object (a reference in the stream). Following this, the GetFrom function of the CCollaborator class is called. That function merely sets the values of the `itsProviders` and `itsDependents` lists to `NULL` pointers.

After the foregoing operations are complete, execution control returns to the GetFrom function in the CPane class, where GetObject is called to create the object that is to serve as the New button's environment (`itsEnvironment`).

Creating and Initializing the CColorTextEnvirons Object

The CColorTextEnvirons object has been discussed previously. The values of the member variables that apply to the New button's environment are as follows:

- `foreColor = { 0, 0, 0 }`
- `backColor = { 65535, 65535, 65535 }`
- `penSize = { 1, 1 }`
- `changeForeColor = FALSE`
- `changeBackColor = FALSE`
- `changeForePattern = FALSE`
- `changeBackPattern = FALSE`
- `resPatterns = TRUE`
- `penMode = 8 (patCopy)`
- `(reserved) = 0`

After the foregoing are retrieved they are stored in member variables of the same names as the fields. Because the `resPatterns` variable is `TRUE`, the GetFrom function continues by accessing the following variables from the stream:

- `forePatID = 0`
- `backPatID = 0`
- `forePatInd = 0`
- `backPatInd = 0`

The GetFrom function of the CColorTextEnvirons class completes its execution by calling the GetFrom function of the CTextEnvirons class.

Initializing the CTextEnvirons Class Variables

The GetFrom function of the CTextEnvirons class accesses various values from the stream and places them into member variables. The variables and their values for this example are as follows:

- `textInfo.theSize = 10`
- `textInfo.theStyle = 0 (normal)`
- `textInfo.theMode = 0 (srcCopy)`
- `fontName = Chicago`

The value in the `fontName` variable is used to look up the corresponding font number by calling the GetFNum toolbox function, and the number is placed into the `textInfo.fontNumber` member variable for the object. The GetFrom function then calls the GetFrom function in the CEnvironment class (its base class) to access additional values from the stream.

Initializing the CEnvirons Class Variables

The CEnvironment class has no data members, so the GetFrom member function in the CEnvirons class (its base class) is called instead. That function accesses the value of the `moreEnvirons` pointer from the stream, which is `NULL` in this case. Execution continues in the GetFrom function of the CPane class, after the environment data has been accessed.

Continuing the New Button's CPane Class Initialization

The New button has no associated border object, so the object returned by GetObject in the GetFrom function of the CPane class returns a `NULL` pointer for the `itsBorder` member variable.

Creating and Initializing the Delete CButton Object

The GetObject function is called inside the GetFrom function of the CView class to access and create the next subview of the CPanorama object. This is the Delete button shown in Figure 9-1, whose components are shown in Figure 9-5. The values for the CButton class variables are as follows:

- `procID = 8`
- `clickCmd = 515 (cmdDeleteCat)`

After the foregoing values have been stored, the function calls the GetControl function that it inherits from the CControl class. When that function returns, the NewControl toolbox function is called to create the button using the values of the button's member variables.

Initializing the Delete Button's CControl Class Variables

The GetControl function of the CControl class accesses the stream to get the values of four variables, as follows:

- `value = 0`
- `cmin = 0`
- `cmax = 1`
- `title = Delete`

Following the acquisition of the foregoing values, the GetControl function calls the GetFrom function of the CPane class.

Initializing the Delete Button's CPane Class Variables

The GetFrom function in the CPane class has been described for many other objects. In the case of the Use button, the values accessed from the stream are as follows:

- `width = 47`
- `height = 16`
- `hEncl = 170`
- `vEncl = 205`
- `hSizing = 4(sizFIXEDSTICKY)`
- `vSizing = 4(sizFIXEDSTICKY)`
- `printClip = 1(clipFRAME)`
- `frame.top = 0`
- `frame.left = 0`
- `frame.bottom = 16`
- `frame.right = 47`
- `autoRefresh = TRUE`
- `flags = 3`

After the foregoing values have been stored into the Delete button's member variables, the GetFrom function in the CPane class accesses the stream to acquire the pointer to the button's enclosure. In this case, the stream contains a reference to the second object (the CPanorama object) in the stream's `checklist`. The function continues by calling the SubpaneLocation and GetAperture functions. Then the function calls the GetFrom function in the CView class.

Initializing the Delete Button's CView Class Variables

The GetFrom function of the CView class has been discussed previously. The member variables and the values accessed from the stream for the New button are as follows:

- `visible = TRUE`
- `active = FALSE`
- `wantsClicks = TRUE`
- `canBeGopher = FALSE`
- `ID = 2053`
- `usingLongCoord = FALSE`
- `(reserved) = 0`
- `helpResIndex = 0`
- `numSubviews = 0`

After the foregoing values have been acquired, the port for the view is set to the port of its enclosure (CPanorama) and the view is added to its enclosure's list of subviews (`itsSubviews`). The fact that the value of `numSubviews` variable is 0 causes the loop to be skipped and the GetFrom function of the CBureaucrat class (its base class) to be called.

Initializing the Delete Button's CBureaucrat Class Variables

The GetFrom function of the CBureaucrat class accesses the stream to get the object that is to be assigned as the button's supervisor. In the case of the Delete button, the `itsSupervisor` is set to a pointer to the CPanorama object (a reference in the stream). Following this, the GetFrom function of the CCollaborator class is called. That function merely sets the values of the `itsProviders` and `itsDependents` lists to `NULL` pointers.

After the foregoing operations are complete, execution control returns to the GetFrom function in the CPane class, where GetObject is called to create the object that is to serve as the New button's environment (`itsEnvironment`).

Creating and Initializing the CColorTextEnvirons Object

The CColorTextEnvirons object has been discussed previously. The values of the member variables that apply to the New button's environment are as follows:

- `foreColor = { 0, 0, 0 }`
- `backColor = { 65535, 65535, 65535 }`
- `penSize = { 1, 1 }`
- `changeForeColor = FALSE`
- `changeBackColor = FALSE`
- `changeForePattern = FALSE`
- `changeBackPattern = FALSE`
- `resPatterns = TRUE`
- `penMode = 8 (patCopy)`
- `(reserved) = 0`

After the foregoing are retrieved they are stored in member variables of the same names as the fields. Because the `resPatterns` variable is `TRUE`, the GetFrom function continues by accessing the following variables from the stream:

- `forePatID = 0`
- `backPatID = 0`
- `forePatInd = 0`
- `backPatInd = 0`

The GetFrom function of the CColorTextEnvirons class completes its execution by calling the GetFrom function of the CTextEnvirons class.

Initializing the CTextEnvirons Class Variables

The GetFrom function of the CTextEnvirons class accesses various values from the stream and places them into member variables. The variables and their values for this example are as follows:

- `textInfo.theSize = 10`
- `textInfo.theStyle = 0 (normal)`
- `textInfo.theMode = 0 (srcCopy)`
- `fontName = Chicago`

The value in the `fontName` variable is used to look up the corresponding font number by calling the GetFNum toolbox function, and the number is placed into the `textInfo.fontNumber` member variable for the object. The GetFrom function then calls the GetFrom function in the CEnvironment class (its base class) to access additional values from the stream.

Initializing the CEnvirons Class Variables

The CEnvironment class has no data members, so the GetFrom member function in the CEnvirons class (its base class) is called instead. That function accesses the value of the `moreEnvirons` pointer from the stream, which is `NULL` in this case. Execution continues in the GetFrom function of the CPane class, after the environment data has been accessed.

Continuing the Delete Button's CPane Class Initialization

The Delete button has no associated border object, so the object returned by GetObject in the GetFrom function of the CPane class returns a `NULL` pointer for the `itsBorder` member variable.

Completing the Creation of the CCategories CDialog Object

After the foregoing objects have been created and the loop in the CView class of the CPanorama object has finished the creation of all of the panorama's subviews, control returns to the GetFrom function in the CPane class to finish the creation of the CPanorama object. The GetObject function is called to acquire and create the object that will serve as the panorama's environment (`itsEnvironment`). The function accesses the stream to find

and create a CColorTextEnvirons object. After the object has been created and added to the stream's list of object (`checklist`), the object's GetFrom function is called to initialize the object.

Creating and Initializing the CColorTextEnvirons Object

The CColorTextEnvirons object has been discussed previously. The values of the member variables that apply to the CPanorama object's environment are as follows:

- `foreColor = { 0, 0, 0 }`
- `backColor = { 65535, 65535, 65535 }`
- `penSize = { 1, 1 }`
- `changeForeColor = FALSE`
- `changeBackColor = FALSE`
- `changeForePattern = FALSE`
- `changeBackPattern = FALSE`
- `resPatterns = TRUE`
- `penMode = 8 (patCopy)`
- `(reserved) = 0`

After the foregoing are retrieved, they are stored in member variables of the same names as the fields. Because the `resPatterns` variable is `TRUE`, the GetFrom function continues by accessing the following variables from the stream:

- `forePatID = 0`
- `backPatID = 0`
- `forePatInd = 0`
- `backPatInd = 0`

The GetFrom function of the CColorTextEnvirons class completes its execution by calling the GetFrom function of the CTextEnvirons class.

Initializing the CTextEnvirons Class Variables

The GetFrom function of the CTextEnvirons class accesses various values from the stream and places them into member vari-

ables. The variables and their values for this example are as follows:

- `textInfo.theSize = 12`
- `textInfo.theStyle = 0 (normal)`
- `textInfo.theMode = 0 (srcCopy)`
- `fontName = Chicago`

The value in the `fontName` variable is used to look up the corresponding font number by calling the GetFNum toolbox function, and the number is placed into the `textInfo.fontNumber` member variable for the object. The GetFrom function then calls the GetFrom function in the CEnvironment class (its base class) to access additional values from the stream.

Initializing the CEnvirons Class Variables

The CEnvironment class has no data members, so the GetFrom member function in the CEnvirons class (its base class) is called instead. That function accesses the value of the `moreEnvirons` pointer from the stream, which is `NULL` in this case. Execution continues in the GetFrom function of the CPane class, after the environment data has been accessed.

The CPanorama object has no border, so when the GetObject function is called to access and create a border, a `NULL` pointer is returned.

Finishing the Creation of the CDialog Object

After the CPanorama object (the single subview of the CDialog object) has been created and all of its subviews have been created, control returns to the GetFrom function of the CDialog class to access the stream and create the objects that will serve as the dialog's default button (`itsDefaultButton`) and its panorama (`itsPanorama`). Both of these objects have been created previously, so the stream contains references to them both. The values accessed for these variables are as follows:

- `itsDefaultButton = object #10 (Use CButton)`
- `itsPanorama = object #2 (CPanorama)`

After the foregoing have been accessed from the stream, the creation of the Categories dialog view is complete and control returns to the user's MakeNewWindow function.

Using Object I/O to Save and Restore Data Objects

In addition to the features to recreate user interface elements using the Object I/O capabilities of the TCL, many of these same features can be used to save and restore data objects.

The bulk of the code to save and restore objects from an external file is contained in the **CSaver.tem** file. That file implements the functionality for handling the **New**, **Open**, **Save**, **Save As**, and **Revert** commands. The **CStream.tem** file implements the stream-oriented I/O that is used for reading and writing objects. Both of these files contain functions that pertain to the class for which their template is expanded. And the templates must be expanded explicitly in the code, some of which is generated automatically by the VA, and some of which you may have to write. The base class for the CSaver class is CDocument.

Object I/O Code That the VA Generates

When you create a VA application and then check the Use File checkbox in the View Info dialog for its main window, the VA will generate several pieces of important data.

Assuming that your main window is named Main, the VA generates a file called **CSaver_CMain.cpp** whose contents are as follows:

```
#include <CSaver.h>

#pragma template_access public

#define GENERATE_TEMPLATE
#include "ItsContents_CMain.h"

#include <CSaver.tem>
```

The foregoing code includes the contents of the **CSaver.h** header file, which declares the CSaver class as a template class. Then the `#pragma` statement declares that the scope of the instantiation of the class and its member functions is public. The `#define` is referenced in the **itsContents_CMain.h** file to force the CSaver class

template definitions to be generated. The contents of the **itsContents_CMain.h** file are as follows:

```
#pragma once

#include "CCollaborator.h"

#define ITSCONTENTS_CMain CCollaborator

#ifdef GENERATE_TEMPLATE

#pragma template CSaver<CCollaborator>

#endif
```

In the foregoing, the contents of the **CCollaborator.h** header file defines the variables and member functions of that class. Then the `ITSCONTENTS_CMain` symbol is defined to be CCollaborator, and then if the `GENERATE_TEMPLATE` symbol is defined (which it is), the `#pragma` statement causes the compiler to instantiate a CCollaborator version of the CSaver template.

The inclusion of the **CSaver.tem** file, by means of the `#include` statement at the end of the **CSaver_CMain.cpp** file (shown earlier) brings in the definitions for all of the member functions of the CSaver class, instantiated for the CCollaborator object.

The CCollaborator class includes GetFrom and PutTo functions for reading from and writing to a stream. As the previous section of this chapter demonstrates, the CStream_CCollaborator class provides the ability to read CCollaborator objects from a stream. It also allows you to write a CCollaborator object to a stream.

However, the VA-generated code is really not capable of saving or reading the data for even a CCollaborator-derived data class. The general prescription for doing so is as follows:

1. Define your data contents class and create both source and header files for it.
2. Modify the **itsContents_CMain.h** file (if your document class is CMain) to refer to your data contents class instead of CCollaborator.
3. Expand the templates for the GetObject, PutObject, and PutObjectReference functions in a **CStream_Contents.cpp** file (or equivalent).

4. Create and initialize the `itsContents` variable in the MakeNewContents function of your CSaver-derived document class (for example, CMain).

5. Call `TCL_FORCE_REFERENCE()` for your data contents class, and any other classes whose data it reads or writes, either directly or indirectly, in its GetFrom or PutTo functions.

6. Add code to the ContentsToWindow and WindowToContents functions, if necessary.

The next section contains an example that follows the foregoing prescription.

Reading and Writing the Notebook Contents

In Chapter 5, I described a typical application that included a text window and a modal dialog for changing the font, size, style, and justification of the text. The description of that application begins on page 171. While that chapter was devoted primarily to the discussion of various types of dialogs, the application itself is perfect for illustrating how to read and write data using the Object I/O features of the TCL. The following sections describe how each of the steps of the foregoing prescription is followed to implement the Object I/O features for the Notebook application.

Step 1: Define the Data Contents Class

In what I'll call the Notebook application, we need to create a handle that can contain all of the text typed into the main window of the application. In addition to the contents of the handle, we need to store information that will enable us to recreate the text font, size, style, and justification features of the text, as the user has specified them.

I have created a new class, called CNoteContents to hold these data. The header file for the class is named **NoteContents.h**, and it was created with the Symantec C++ editor. The contents of this file are as follows:

```
/************************************************************
 NoteContents.h

        Header File For CNoteContents class

   Copyright © 1995 Richard O. Parker. All rights reserved.
************************************************************/
```

```
#pragma once

#include "CTextSettings.h"

class CNoteContents TCL_AUTO_DESTRUCT_OBJECT
{
   public:

      TCL_DECLARE_CLASS

      CNoteContents();
      virtual ~CNoteContents();

      void     PutTo(CStream&);
      void     GetFrom(CStream&);

      Handle   GetTextHandle();
      void     SetTextHandle(Handle t);
      CTextSettings    GetTextSettings();
      void     SetTextSettings (CTextSettings& settings);

   protected:

   CTextSettings    itsSettings;
   Handle           itsText;

};
```

The foregoing class declaration contains a constructor, destructor, and a number of member functions. In addition, it contains two sets of data. The first is a CTextSettings structure, whose contents define the text font, size, style, and justification information. These data are stored in the `itsSettings` variable. In addition, a handle is provided to contain the ASCII text. The **CTextSettings.h** header file contents are as follows:

```
#pragma once

struct CTextSettings
{
   Str255   spFontName;
   Str255   spFontSize;
   short    nFontStyle;
   short    nFontJust;

};
```

In the foregoing, the font name, font size, font style, and font justification data formats are described. The font size is stored as a string, even though it is a numeric value. This facilitates its use in the remainder of the application. The contents of the **NoteContents.cp** source file are as follows:

```
/************************************************************
 NoteContents.cp

        Source File For CNoteContents class

   Copyright © 1995 Richard O. Parker. All rights reserved.
************************************************************/

#include "NoteContents.h"

TCL_DEFINE_CLASS_D0(CNoteContents);
```

The first section of the file includes the class declaration header file and defines the class to the TCL so that the run-time type identification features can be used to recreate the class.

```
CNoteContents::CNoteContents()
{
   itsText = 0L;
   TCLpstrcpy (itsSettings.spFontName, "\pChicago");
   TCLpstrcpy (itsSettings.spFontSize, "\p12");
   itsSettings.nFontStyle = normal;
   itsSettings.nFontJust  = teFlushLeft;

   TCL_END_CONSTRUCTOR
}
```

The foregoing is the default constructor that is executed when the object is first created.

```
CNoteContents::~CNoteContents()
{
   TCL_START_DESTRUCTOR
   DisposHandle (itsText);
}
```

The foregoing is the destructor for the CNoteContents object. It disposes of the handle to the object's text. The `itsSettings` variable does not need to be disposed. It will disappear when the object is deleted.

```
void CNoteContents::PutTo (CStream& stream)
{
   stream << itsSettings.spFontName;
   stream << itsSettings.spFontSize;
   stream << itsSettings.nFontStyle;
   stream << itsSettings.nFontJust;
   stream << itsText;
}
```

The foregoing is the entire code that is needed to write out the contents of the CNoteContents object. As is evident, the features of the CStream class to write out strings, short variables, and the text handle are used.

```
void CNoteContents::GetFrom (CStream& stream)
{
   stream >> itsSettings.spFontName;
   stream >> itsSettings.spFontSize;
   stream >> itsSettings.nFontStyle;
   stream >> itsSettings.nFontJust;
   stream >> itsText;
}
```

As for the PutTo function, the foregoing GetFrom function is very simple. It too uses the features of the CStream class to read strings, short variables, and the text handle.

```
Handle CNoteContents::GetTextHandle()
{
   return itsText;
}
```

The foregoing function is an "access function" used by the application to get the handle to the object's text. You will see this function being used later in the code.

```
void CNoteContents::SetTextHandle(Handle t)
{
   itsText = t;
}
```

As for the GetTextHandle function, this is an "access function" that is used by the application to specify a handle for the text. The function is referenced later in the code.

```
CTextSettings CNoteContents::GetTextSettings()
{
   return itsSettings;
}

void CNoteContents::SetTextSettings (CTextSettings& settings)
{
   itsSettings = settings;
}
```

The foregoing two functions are also "access functions" used by the application to get and set the font settings of the CNoteContents object. The functions are referenced later in the code.

Step 2: Modify the itsContents_CMain.h File

The prescription for implementing Object I/O specifies that we should modify the **itsContents_CMain.h** file to reference our new data contents class (CNoteContents in this case). The modified contents of that file are as follows:

```
#pragma once

#include "NoteContents.h"

#define ITSCONTENTS_CMain CNoteContents

#ifdef GENERATE_TEMPLATE

#pragma template CSaver<CNoteContents>

#endif
```

Notice that the foregoing is remarkably similar to what was shown for this file in the VA-generated code (see page 388). We include the **NoteContents.h** header file and then define the contents variable, `ITSCONTENTS_CMain` to `CNoteContents`. That establishes the class name of the contents object. Then the `#pragma` statement tells the compiler to instantiate the CSaver template for the CNoteContents class.

Step 3: Expand the CStream Class Templates

The third step of the Object I/O prescription calls for expanding the templates for GetObject, PutObject, and PutObjectReference for the CStream-based template. In our case, this is a file we have created, called **CStream_CNoteContents.cpp**. The contents of that file are as follows:

```
#include "CStream.h"
#include "NoteContents.h"

#pragma template_access public
#pragma template PutObject(CStream&, CNoteContents*)
#pragma template GetObject(CStream&, CNoteContents*&)
#pragma template PutObjectReference(CStream&, CNoteContents*)

#include "CStream.tem"
```

As is evident from the foregoing, I have included the header file for our CNoteContents class, have declared that the scope of the class and member functions of the CStream<CNoteContents> template are public, and then have expanded the templates for the specified functions. Including the **CStream.tem** file provides the source code from which the template-derived classes are created.

Step 4: Create and Initialize the itsContents Variable

This step requires that we provide code in our CSaver-derived class—CMain in this case—to create and initialize the `itsContents` variable that contains a pointer to the data contents object. The code to do this for our example is in the MakeNewContents function in our CMain class. The code is as follows:

```
void CMain::MakeNewContents()
{
   itsContents = new CNoteContents;
}
```

The foregoing code creates a new CNoteContents object and the constructor of that object fills in default values for both the data handle and the text font, size, style, and justification settings. Refer to page 391 for the listing of the CNoteContents constructor function.

Step 5: Call TCL_FORCE_REFERENCE

The fifth step in the Object I/O prescription indicates that we should call the `TCL_FORCE_REFERENCE` macro for our data contents class, to ensure that it is linked into the final object file. Because our data contents object is being created implicitly, there is no way that the compiler can know to include its code, other than by our making an explicit reference to it. I have included the necessary code in the ForceClassReferences function of the **CApp.cp** source file, as follows:

```
void CApp::ForceClassReferences(void)
{
   x_CApp::ForceClassReferences();

   TCL_FORCE_REFERENCE (CNoteContents);
}
```

The foregoing code calls the ForceClassReferences function in the base class first and then uses the `TCL_FORCE_REFERENCE` macro to force the reference to our CNoteContents class.

Step 6: Add Code to Transfer the Data to/from Windows

The final step in the Object I/O prescription is to add code to the ContentsToWindow and WindowToContents functions, if necessary, to display the data in a window or take the data from a window for storage. The additional code for the ContentsToWindow function is as follows:

```
void CMain::ContentsToWindow()
{
   // Transfer data from itsContents to itsWindow.
   // See Chapter 8, Using Object I/O

   fMain_TextPane->SetTextHandle (itsContents->GetTextHandle());
   settings = itsContents->GetTextSettings();
   UpdateWindowText();
}
```

As the foregoing code illustrates, the process of transferring the contents of the data handle to the window is accomplished by using the GetTextHandle access function in the CNoteContents class (see page 392) and then using the pointer to the CEditText pane to call its SetTextHandle function to store the handle we just accessed. We also access the GetTextSettings function to retrieve the font, size, style, and justification data, storing these into the `settings` variable. UpdateWindowText is a function that I have provided to update the appearance of the text, using the data in the `settings` variable. The code for this function is as follows:

```
void CMain::UpdateWindowText ()
{
   short nFontNum, nFontSize;

   GetFNum (settings.spFontName, &nFontNum);
   nFontSize = atoi ((const char *)&settings.spFontSize[1]);
   fMain_TextPane->SetFontNumber (nFontNum);
   fMain_TextPane->SetFontSize (nFontSize);
   fMain_TextPane->SetFontStyle (normal);
   fMain_TextPane->SetFontStyle (settings.nFontStyle);
   fMain_TextPane->SetAlignment (settings.nFontJust);
}
```

The foregoing code translates the font name into a font number, converts the font size string into a numeric value, and then calls

the various member functions of the CEditText class to set the attributes of the text pane to the values in the `settings` variable. This results in an update event being created for the window, causing the TCL to redraw the text in the specified font, size, style, and justification.

The code to transfer the contents of the window back to the data contents object, in preparation for storing the data and its specifications into a file, is held in the WindowToContents function, which is as follows:

```
void CMain::WindowToContents()
{
   // Transfer data from itsWindow to itsContents

   itsContents->SetTextHandle (fMain_TextPane->GetTextHandle());
   itsContents->SetTextSettings (settings);
}
```

The foregoing code merely calls the SetTextHandle (see page 392) and SetTextSettings (see page 392) functions of the CNoteContents class to transfer the data and its settings from the window to the data contents object. The data and settings are written out using the PutTo function for the CNoteContents class that was shown on page 391.

Reading and Writing the Categories List Contents

Our next example relates to the categories list that was shown in connection with the CCategories class modeless dialog presented in Chapter 5 (see page 208). The discussion with regard to the categories list defines it as a CArray object. In order to show you how to use the Object I/O facilities of the TCL to read and write the members of a CList object, I have modified the code in both the CMain and CCategories classes and have added new code to the GetFrom and PutTo functions in the CCat class (the class associated with individual category objects). The changes to the foregoing code are not significant; however, they will be presented here, in the interests of completeness.

Defining the Categories List

The **CMain.h** header file contains the definition of the `categories` variable. Instead of defining it as a CArray object, I have

changed it to a CList object. The first part of the modified header file is as follows:

```
#include "x_CMain.h"
#include <CList.h>

class CList;
class CCat;

typedef CList<CCat> CCatList;

class CMain : public x_CMain

{
public:

   TCL_DECLARE_CLASS

   CCatList   *categories;  // list of category names
```

I've added a `typedef` statement that declares a CCatList object is really a CList<CCat> object. In other words, the CList template is going to be expanded to handle CCat objects in the list.

The New Prescription for Object I/O With Lists

Because CList is a template class and because the expansion of templates both for the CList and its base class (CPtrArray) is necessary, the prescription for creating Object I/O code for these objects is somewhat more complex. The steps for including the necessary code to perform the I/O are as follows:

1. Define your contents class (CCat, in our case).
2. Modify the itsContents_CMain.h file to use your CList class instead of CCollaborator (CList<CCat>, in our case).
3. Expand the templates for PutObject and GetObject functions for your contents class and your list class.
4. Initialize the `itsContents` pointer in the MakeNewContents function of your generated contents class (CMain).
5. Call the `TCL_FORCE_REFERENCE` macro for your contents class and any other classes it reads, directly or indirectly, in its GetFrom function and call the `TCL_FORCE_TEMPLATE_REFERENCE` macro for your list class.
6. Expand the templates for the CList and CPtrArray classes.

7. Expand the template for the PutObject1 function for your contents class (CCat).

8. Implement the ContentsToWindow and WindowToContents functions in your contents class, if necessary.

The foregoing steps are illustrated with code exhibits in the next several sections.

Step 1: Define Your Contents Class

I have defined the CCat class as the class that implements the functionality of an individual category list object. To this end, I have added code to its GetFrom and PutTo functions to read and write the data associated with one of these objects. The code for the PutTo and GetFrom functions is as follows:

```
void CCat::PutTo (CStream& stream)
{
   stream << catName << catDescrip << catType << catTaxable;
}

void CCat::GetFrom (CStream& stream)
{
   stream >> catName >> catDescrip >> catType >> catTaxable;
}
```

The foregoing code writes (PutTo) and reads (GetFrom) four member variables of the class that describe the information pertaining to a category. There's the category name (`catName`), the category description string (`catDescrip`), the category type code (`catType`) and the tax-related status of the category (`catTaxable`). The code to perform the I/O is quite straightforward.

Step 2: Modify the itsContents_CMain.h File

This step requires identifying the nature of the `itsContents` variable to other files which include this header file, including the **CSaver_CMain.cpp** and **x_CMain.h** files. The contents of the **itsContents_CMain.h** file are as follows:

```
#pragma once
#include "CCat.h"
#define ITSCONTENTS_CMain CList<CCat>
#ifdef GENERATE_TEMPLATE
#pragma template CSaver<CList<CCat>>
#endif
```

The foregoing code defines the `ITSCONTENTS_CMain` symbol as the CList<CCat> object and then permits the compiler to create an instance of the CList<CCat> object for the CSaver template class.

Step 3: Expand the GetObject and PutObject Templates

The GetObject and PutObject functions are members of the CStream class and so to expand the functions for our contents class, I created a new file called **CStream_CCat.cpp**. The contents of that file are as follows:

```
#include "CStream.h"
#include "CCat.h"

#pragma template_access public
#pragma template PutObject(CStream&, CCat*)
#pragma template GetObject(CStream&, CCat*&)

#pragma template PutObject(CStream&, CList<CCat>*)
#pragma template GetObject(CStream&, CList<CCat>*&)
#include "CStream.tem"
```

The foregoing code includes the header files for both the CStream and CCat classes, declares that the scope of the template member functions and variables is to be public, and then explicitly expands templates for the PutObject and GetObject functions, for both the CCat and the CList<CCat> classes. The **CStream.tem** file supplies the source code to perform the GetObject and PutObject I/O for the objects.

Step 4: Initialize the itsContents Pointer

When the TCL creates a new window, it also calls the MakeNewContents function in the CMain (or your document class). The revised code for the MakeNewContents function is as follows:

```
void CMain::MakeNewContents()
{
   // Initialize document contents and itsWindow here
   itsContents = TCL_NEW (CCatList, ());
}
```

The foregoing code creates a new instance of the CCatList object. The `typedef` statement equates the name CCatList to refer to a CList<CCat> object.

Step 5: Call TCL_FORCE_REFERENCE

In order to ensure that the contents list and the objects stored within that list are linked into the final application code, you have to modify the CApp.cp file to incorporate these references. There is one concern with regard to this, particularly with respect to template-derived objects. You *must* also include, at the beginning of the file, a statement like the following:

```
//
// the following statement is required to
// make the Symantec C++ compiler happy.
//
typedef CList<CCat> dummytype;
```

The foregoing indicates that some "dummytype" is equivalent to typing CList<CCat>. This is an "occult" feature of the compiler, and you need only follow this example, using your own object type in substitution for CList<CCat>. In addition to the foregoing, the code in the ForceClassReferences function of the CApp class includes the following code:

```
void CApp::ForceClassReferences(void)
{
   x_CApp::ForceClassReferences();
   TCL_FORCE_REFERENCE(CCat);
   TCL_FORCE_TEMPLATE_REFERENCE(CList, CCat);
}
```

As is evident, the TCL_FORCE_REFERENCE macro forces the linker to include the CCat class. The additional macro call (TCL_FORCE_TEMPLATE_REFERENCE) forces the CList<CCat> code to be linked as well.

Step 6: Expand the Templates for CList and CPtrArray

The CList class is derived from the CPtrArray class and provides the stream input/output features for lists. We expand the templates for these two classes, with regard to our contents object, by creating a new file named **CList_CCat.cpp**, whose contents are as follows:

```
#include "CCat.h"

#pragma template_access public

#pragma template CPtrArray<CCat>
```

```
#pragma template CList<CCat>

TCL_DEFINE_TEMPLATE_CLASS_D1(CList, CCat, CPtrArray<CCat>)
TCL_DEFINE_TEMPLATE_CLASS_D1(CPtrArray, CCat, CVoidPtrArray)

#include <CList.tem>
#include <CPtrArray.tem>
```

The foregoing code includes the header file for the CCat class, declares that the scope of the expanded template code and member variables is public, then expands the templates for the CPtrArray<CCat> and CList<CCat> classes. In addition to this, the TCL_DEFINE_TEMPLATE_CLASS macros allow the dynamic creation of template-based objects with the new_by_name facilities of the TCL. These macros work only when there is a single variant class. The first parameter is the class name, the second parameter is the template type name, and the third argument is the base class name. Therefore, the first of these macros provides for creation of the CList<CCat> class, which is based upon the CPtrArray<CCat> class. The second provides for creation of the CPtrArray<CCat> class, based upon the CVoidPtrArray class.

Inclusion of the **Clist.tem** and **CPtrArray.tem** files provides the source code, which implements the functionality of the expanded templates.

Step 7: Expand the Template for the PutObject1 Function

The PutObject1 function is called in a loop by the PutItems function of the CList<CCat> class, for each object in the list. The PutObject1 function calls PutObject, with the stream and the object type we specify as its arguments. We need to expand the template for the PutObject1 function so that it will refer to our CCat objects when it is called. We accomplish this by creating another new file named **PutObject1_CCat.cpp**. The contents of this file are as follows:

```
#include "CCat.h"
#pragma template_access public
#pragma template PutObject1(CCat*, long)
#include <PutObject1.tem>
```

The foregoing code includes the header file for the CCat class, declares the scope of the expanded template code and variables to be public, and expands the template for the PutObject1 function

with arguments of our CCat class pointer and a long variable, which will hold the stream pointer. The **PutObject1.tem** file contains the source code for the PutObject function.

Step 8: Implement the Contents Transfer Functions

The ContentsToWindow function is called when a file is read and the contents are to be transferred to a window. In our case, the contents are not transferred, but we have used this function to copy the `itsContents` pointer to our local document's `categories` pointer. The code is as follows:

```
void CMain::ContentsToWindow()
{
   categories = itsContents;
}
```

Similar to the foregoing, the code for the WindowToContents function transfers the local pointer to the `itsContents` variable, as follows:

```
void CMain::WindowToContents()
{
   itsContents = categories;
}
```

The foregoing sections demonstrate the complete implementation of Object I/O facilities for our categories list. While there are a number of steps to follow in implementing this functionality, each of the steps is rather simple.

When You Don't Want to Use Object I/O

There are times when you need to read and write files in a specific format that is different from what is produced by the Object I/O features of the TCL. I gave you an example of this in Chapter 3, where I introduced the CTextData class to implement simple text input and output.

The TCL also includes a generic facility for input and output of data in an arbitrary format. This facility is encapsulated in the CSimpleSaver class, whose base class is CDocument.

Creating a Simple User Interface

To illustrate the use of the features of the CSimpleSaver class, I have created a simple project, like the Notebook example presented earlier, including a main window that contains a CPanorama as its only object. In this case, the VA will create a window that appears like what is shown in Figure 9-6.

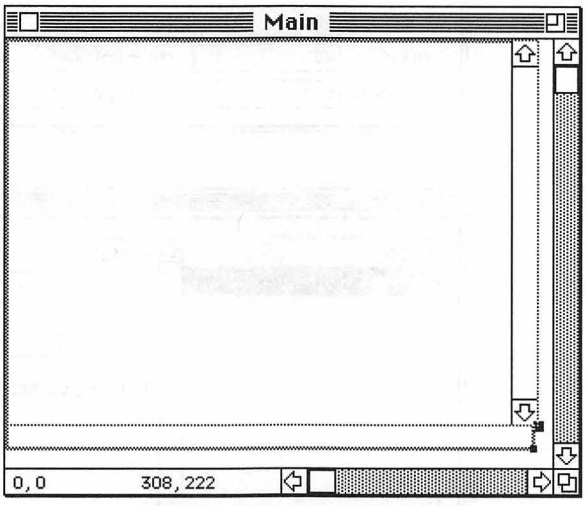

Figure 9-6
A simple main window containing a CEditText pane

The foregoing figure shows the design of the Main document's window, indicating that it contains a CPanorama object, with a vertical scroll bar, that will be stored as a CEditText object in the "Main" window's `'CVue'` resource. In order to make use of the CSimpleSaver features, rather than the somewhat more complex features of Object I/O, it is necessary to remove the check from the "Use File" checkbox in the View Info dialog. This is shown in Figure 9-7. Note that I have also unchecked the Vert Scroll and Horiz Scroll checkboxes for the window.

Because the Use file checkbox is not checked, the VA would normally generate the x_CMain class as being derived directly from the CDocument class. Because we want it to be derived from the CSimpleSaver class instead (which itself is derived from CDocument), we must choose the Classes command from the Edit menu, select the CMain class in the left-hand list, and then enter the CSimpleSaver class name as the Library Class for the CMain class, as shown in Figure 9-8.

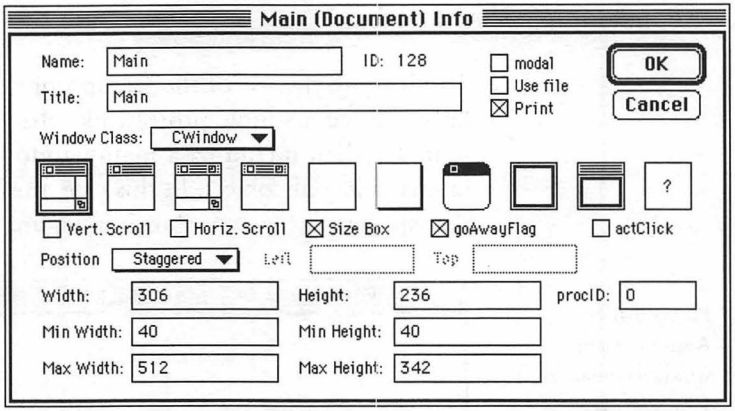

Figure 9-7
View Info dialog for Main window

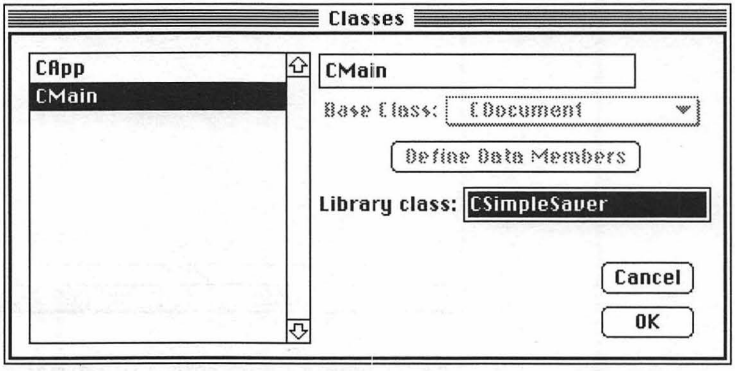

Figure 9-8
CSimpleSaver specified as the Library class

Writing the Code to Read and Write the File

By using CSimpleSaver as the "Library Class" for the CMain class, you are telling the VA to generate the code such that the base class for the x_CMain class will be CSimpleSaver. This is illustrated by the **x_CMain.h** header file, whose contents are as follows:

```
#pragma once

#include "CSimpleSaver.h"
class CEditText;

#define x_CMain_super   CSimpleSaver

class CFile;

class x_CMain : public x_CMain_super
{
public:
```

```
   TCL_DECLARE_CLASS

   // Pointers to panes in window
   CEditText  *fMain_TextPane;

   void       Ix_CMain(void);


protected:
   virtual void MakeNewWindow(void);

   virtual void NewFile(void);
   virtual void OpenFile(SFReply *macSFReply) {}
   virtual void MakeWindowName(Str255 newName);
   virtual void MakeNewContents(void);

};

#define CVueCMain  128
```

Note in the foregoing that the symbol `x_CMain_super` is being defined as `CSimpleSaver` and that the symbol is used as the base class name for the x_CMain class. In effect, CSimpleSaver is the base class for the x_CMain class.

Now, after generating the code for this project, you must also define the ReadContents and WriteContents member functions for the CMain class. This is because the CSimpleSaver class (which is a base class for CMain) declares those functions (as well as the MakeNewWindow function) as pure virtual functions. If you don't at least create declarations in the **CMain.h** and stubs for these in the **CMain.cp** file, then you will get compile errors in the x_CApp class, which attempts to construct CMain objects in its CreateDocument and OpenDocument member functions.

The contents of the **CMain.h** header file, to which the declarations for the ReadContents and WriteContents functions were added, are as follows:

```
#pragma once

#include "x_CMain.h"

class CMain : public x_CMain
{
public:

   TCL_DECLARE_CLASS

   void          ICMain(void);

   virtual void MakeNewContents(void);
   virtual void ContentsToWindow(void);
   virtual void WindowToContents(void);
```

```
   virtual void ReadContents(CFileStream *aStream);
   virtual void WriteContents(CFileStream *aStream);
};
```

ReadContents Function Code

The custom code to read the contents of a text file and store the data into the CEditText pane's handle is as follows:

```
void CMain::ReadContents(CFileStream *aStream)
{
   Handle h = aStream->GetHandle();

   HLock(h);
   try_
   {
      fMain_TextPane->SetTextPtr(*h, GetHandleSize(h));
      DisposeHandle(h);
   }
   catch_all_()
   {
      DisposeHandle(h);
   }
   end_try_
}
```

The foregoing code reads the contents of the selected file (using the GetHandle function of the CFileStream object) into a temporary handle, locks the handle, and then calls the SetTextPtr function for the CEditText pane (`fMain_TextPane`). The reason for using a temporary handle and placing the SetTextPtr call inside a "try" block is that the SetTextPtr function makes a copy of the data in the pointer that is passed, so it is possible that lack of sufficient memory to do so could cause an exception. If the call to SetTextPtr is successful, then the temporary handle is disposed.

WriteContents Function Code

The code to write out the CEditText pane's contents is quite simple and is as follows:

```
void CMain::WriteContents(CFileStream *aStream)
{
   aStream->PutHandle(fMain_TextPane->GetTextHandle());
}
```

The foregoing code calls the PutHandle function of the CFileStream object, using the handle returned by calling the GetTextHandle function for the CEditText pane (`fMain_TextPane`).

And that's all there is to using CSimpleSaver as the base class for your document class. Obviously, if you have more complex data, you will need to output and input it using individual put and get calls for the appropriate object types, but the CStream class offers a great number of I/O functions for this purpose. You can't construct objects using the features of CSimpleSaver, but you can read and write data that are stored in a complex form.

Object I/O and CSimpleSaver Summary

This chapter describes two major features of the Object I/O support that is provided within the TCL. The first of these is the ability of the framework to create a complete user interface by reading the `'CVue'` resourced with each window and dialog, and then creating the views and subviews in those objects.

The chapter also shows how these same Object I/O features can be used to read and write data in the form of persistent objects in your application. Two prescriptions for adding Object I/O features to your applications were described.

When an application needs to read and write data files in a specific format, then you can take advantage of the CSimpleSaver class whose features are described in this chapter.

We described the handling of standard events in Chapter 7. The next chapter discusses Apple events in detail.

Chapter 10

Apple Events, Factoring, and Recording

This chapter focuses on the high-level events known as Apple events. It covers how the events are handled if received by your application, how you can arrange to receive new types of events, how you can factor your application to make it scriptable, and how to prepare for recording and playback of Apple events.

The TCL implements the four "required" Apple events. These are Open Application, Open Document, Print Document, and Quit. When you create any stand-alone application, these events will be handled automatically.

When Apple Computer defined the "Object Model" (also referred to as the OM in this text), it also segregated groups of common application-oriented events into "suites." Each suite offers either fundamental support that most applications will want to implement (for example, the "Core Suite"), as well as application-type-specific suites, such as the "Table Suite" or "Text Suite." The TCL contains classes and member functions that implement much of the "Core Suite" of the Apple event Object Model. All of the suites that are currently defined are documented in the *Apple Event Registry* publication of Apple Computer. The registry is updated periodically and contains definitions of all of the standard events.

I described the installation of the TCL's Apple event handlers in Chapter 2, beginning on page 27, in the section titled "Creating the Switchboard Object." In addition, in that same chapter, I talked briefly about the various installed handlers and in what suite they belonged (see page 42). I also touched briefly on the topic of high-level events when describing event handling in Chapter 7 (see page 309). This chapter delves more deeply into the existing installed handlers and provides insight into how they can be used and supplemented to provide the scope of high-level event handling that is applicable to your application's needs.

Support for Receiving Apple Events in the TCL

As described previously, handling of Apple events in the TCL is separated into three classes. The Required Suite is fully implemented and will be described shortly. Portions of the Core Suite are implemented and they too will be described. Several aspects of the Miscellaneous Suite are also implemented and will be described in the sections that follow.

Handling the Required Events

Handlers for the four required events (Open Application, Open Document, Print Document, and Quit) are installed at the time the InstallHandlers function of the CSwitchboard class is executed, when the application is initialized. The code to install these handlers is as follows:

```
InstallEventHandler(kCoreEventClass,
            kAEOpenApplication,
            GenericAppHandlerUPP);

InstallEventHandler(kCoreEventClass,
            kAEOpenDocuments,
            GenericAppHandlerUPP);

InstallEventHandler(kCoreEventClass,
            kAEPrintDocuments,
            GenericNoResultHandlerUPP);

InstallEventHandler(kCoreEventClass,
            kAEQuitApplication,
            GenericAppHandlerUPP);
```

The foregoing statements install handlers for the required events by calling the InstallEventHandler function in the TCL to perform the installation. The arguments are the event class (`kCoreEventClass`), event type (`kAEOpenApplication`, `kAEOpenDocuments`, `kAEPrintDocuments`, and `kAEQuitApplication`), and a pointer to the handler for the event (GenericAppHandlerUPP). The symbols commencing with the letter 'k' are defined in the *Apple Event Registry* for the corresponding constant values, and the definitions for the TCL's use of these constants are in the **AppleEvents.h** header file.

InstallEventHandler is a function in the CSwitchboard class that interfaces with the Apple Event Manager in the Macintosh OS. The code for the InstallEventHandler function is as follows:

```
void CSwitchboard::InstallEventHandler(
   AEEventClass theAEEventClass,
   AEEventID theAEEventID, AEEventHandlerUPP theHandler)
{
   FailOSErr( AEInstallEventHandler(
      theAEEventClass, theAEEventID, theHandler, 0, FALSE) );
}
```

The foregoing code calls the AEInstallEventHandler function, using the supplied arguments. If the function returns a nonzero result, then the FailOSErr function will raise an exception.

Each of the required events specified GenericAppHandlerUPP as the pointer to the handler for the event. This pointer is defined in the CSwitchboard class as a pointer to the GenericAppHandler function in the CAppleEventObject class.

When one of the required events occurs, the event-handling code in the CSwitchboard class accesses the event from the application's event queue and calls the DoHighLevelEvent function to process the event. The code for that function is as follows:

```
void CSwitchboard::DoHighLevelEvent(const EventRecord *theEvent)
{
   OSErrerr;
   if (gSystem.hasAppleEvents)
   {
      err = AEProcessAppleEvent(theEvent);
   }
}
```

As is evident in the foregoing, the event is processed only if the user's machine handles Apple events (`hasAppleEvents`). If so, then the AEProcessAppleEvent toolbox function is called to process the event. Processing the event consists of looking in the application's event dispatch table for a handler whose event class and event ID match those of the received event. If such a handler is not found, then an error is returned; otherwise, the handler is called with a pointer to the event descriptor, a pointer to a default reply descriptor, and a "reference constant" value that was defined when the handler was installed (this is 0 in the case of the required event handlers).

The GenericAppHandler function is called by the Apple Event Manager when one of the required events is sent to the application. The code for that function is as follows:

```
pascal OSErr CAppleEventObject::GenericAppHandler(
   const AppleEvent *theEvent,
   AppleEvent *theReply, long refCon)
{
   return GenericHandler(theEvent, theReply, refCon, DispatchApp);
}
```

The foregoing code calls the GenericHandler function to perform the operation, passing it a pointer to the event descriptor, a pointer to the reply descriptor, the value of the reference constant, and a pointer to a static function called DispatchApp.

The GenericHandler function is used to handle a variety of Apple event types, including the required events. The operations performed by the code in that function are as follows:

- The PackageAppleEvent function of the CApplication object (`gApplication`) is called to construct a CAppleEvent object with the event, reply, and reference constant values. The constructor for the CAppleEvent class initializes a number of member variables and calls the BeginEvent function of the CAppleEventObject class to initialize the stack of CAppleEventObject instances that may be created during processing of the current event. Then the IAppleEvent function is called to initialize the `theEvent`, `theReply`, and `theRefCon` variables to the values supplied to the constructor. Finally, the GetAttributePtr function is called to get a pointer into a buffer containing the event class and event ID values. The pointers are stored for later reference into the `eventClass` and `eventID` variables. The packaged CAppleEvent object pointer is returned to the GenericHandler function.

- After the Apple event has been packaged into a CAppleEvent object, the specific handler function (that is, DispatchApp) is called, with the pointer to the packaged object and a pointer to the result descriptor, to handle the event.

- The DispatchApp function, in turn, calls the DoAppleEvent function in the application object (`gApplication`) to perform the requested function.

- The DoAppleEvent function in the CApplication class handles quite a number of events. The first action of that function is to access the event's class and then determine what to do based

upon that code. For the required events in the core class, the actions are as follows:

- For the Open Application event, if the `newWindowOnStartup` flag is `TRUE`, the function calls the DoCommand function for the current gopher, passing it the `cmdNew` command code.
- For either the Open Documents or Print Documents events, the function calls the DoOpenOrPrintDocEvent function. That function calls OpenDocument to open each specified document and then optionally prints the document. If the document is printed, it is closed after the operation is complete.
- For the Quit Application event, the function calls the DoQuitApplicationEvent function, which accesses the save options in the event and then calls the Quit function. That function sets the `running` variable to `FALSE`, then calls the Quit function for each open window's director, allowing the contents of the window to be saved, if the option to do so was set. If any director refuses to quit, then the `running` variable is set back to `TRUE` and execution continues; otherwise, the event loop is terminated when the state of the `running` variable is tested and found to be `FALSE`, causing the application to quit.

Handling Other Core and Miscellaneous Suite Events

The DoAppleEvent function of the CApplication class handles a number of other application-wide events. These include the NotifyStartRecording and NotifyStopRecording events in the core suite, as well as the Get AETE event in the AppleScript suite and the Activate event in the miscellaneous suite.

In addition to and to amplify the foregoing, during the application's initialization phase, handlers are installed for quite a number of different event types. The TCL does not completely implement many of the events for which handlers are installed; however, it calls a function for which you can provide an override in many of these cases. The various events and what handling is performed are as follows:

- Clone — An event in the Core Suite that can apply to a number of different object types. Functions are provided in the

CDocument, CWindow, and CAppleEventObject classes for this event. None implements the operation, but the opportunity to override the function leaves the functionality open to the developer.

- Close — An event in the Core Suite that is implemented in several of the TCL classes, most notably in CDocument and CWindow, as follows:
  - In CDocument, the DoCloseEvent function is called to close the document and, optionally, save the document's data either in the default file or a file specified in the event. If a file is specified in the event and the `itsFile` object exists, the existing file is closed and the new file is opened before the Close function in the CDocument class is called.
  - In CWindow, the DoCloseEvent function of the window's director is called. That function calls CloseWind for the window. If the window's director is the document, then the DoCloseEvent in the CDocument class is called.
- Count Elements — An event in the Core Suite that is intended to return the number of elements of a particular class in the specified object.
  - An override for the DoCountElementsEvent function is provided in the CWindow class; however, the default behavior is to return an event-not-handled error. You can override this function to provide whatever functionality you desire.
  - The CAppleEventObject class calls the CountObjects function, which returns an event-not-handled error; however, in any class that inherits behavior from the CAppleEventObject class, you can implement the CountObjects function and supply the desired functionality for this event.
- Create Element — An event in the Core Suite that is implemented in the DoCreateElementEvent function of the CApplication class to allow the creation of a new document object. The CAppleEventObject class implements the event by overriding the DoCreateElementEvent, but returns an event-not-handled error. You can override this function to provide the desired functionality.

- Delete — An event in the Core Suite that is intended to allow one or more elements to be deleted from an object that contains them. An example of this might be deleting one or more paragraphs displayed in a particular window.

  - An override for the DoDeleteEvent function is provided in the CWindow class; however, that function returns an event-not-handled error. You can override the function to provide the desired behavior.

  - The CAppleEventObject class also contains the DoDeleteEvent function. It also returns the event-not-handled error; however, you can override that function in any class that inherits behavior from the CAppleEventObject class and implement the Delete event functionality.

- Do Objects Exist — An event in the Core Suite that specifies a set of objects and asks whether they all exist. It is intended to be handled by the application object (`gApplication`). The CApplication class in the TCL does not implement this event; however, if your application class overrides the DoAppleEvent function, you can choose to handle the event. Proper handling of the event requires that you return an Apple event object of `typeBoolean` that has a value of `TRUE` if and only if all of the specified objects exist; otherwise, a value of `FALSE` must be returned.

- Get Data — An event in the Core Suite that requests that the data associated with a set of objects be returned. The "data" in this case depend upon the object to which the event is addressed. The DoGetData function in the CProperty class handles requests for various object properties, as follows:

  - The CWindow class creates CProperty objects to hold properties of type `cBoolean`, with `TRUE` or `FALSE` values, such as `pIsModified`, `pHasCloseBox`, `pHasTitleBar`, `pIsFloating`, `pVisible`, or `pIsZoomed`; of type `cQDRectangle` for the `pBounds` property; of type `cQDPoint` for the `pPosition` property; and a type of `cObjectSpecifier` for the `pSelection` property.

  - The CDocument class creates a CProperty object to hold the `cBoolean` type `pIsModified` property.

- The CApplication class creates a CProperty object to hold the `cIntlText` type `pName` property, and the `cLongInteger` type of `pVersion` property.

The DoGetData function is overridden in the CAppleEventObject class to return the properties of the specified object by calling the MakeSelfSpecifier function in that same class.

- Get Data Size — An event in the Core Suite that is implemented in the CAppleEventObject class by the DoGetDataSizeEvent function. That function returns the size of the object that is addressed by the event.

- Move — An event in the Core Suite that requests that a set of objects be moved to another location. The event is handled by the DoMoveEvent function of the CAppleEventObject class, which returns an event-not-handled error result. You can override this function in any class that inherits behavior from the CAppleEventObject class and performs the requested action.

- Save — An event in the Core Suite that requests that a set of objects be saved in the current or specified file. The TCL implements saving of the current document by recognizing a direct object of the event as pertaining to the CDocument class and calls its DoAppleEvent function to implement the Save operation. If the event contains an optional file specifier, then the DoSaveEvent function in the CDocument class tests whether the `itsFile` object exists, closes the file if it is open, and then opens the specified file before saving the document's contents into the file.

- Set Data — An event in the Core Suite that requests that a set of objects be set to the specified data value. The TCL implements the DoSetDataEvent function as follows:

  - The CClipboard class implements the DoSetDataEvent for a set of "specified types" of objects, including `TEXT`, `styl`, and `PICT`. You can override the GetSupportedTypes function in a class derived from the CClipboard class and provide your own set of supported data types. The DoSetDataEvent function will probably provide the necessary functionality for most applications.

- The CProperty class implements the DoSetDataEvent function for classes that create CProperty objects. These were described with regard to the Get Data event on page 415 and include properties for the CWindow and CDocument classes. The DoPropertySetDataEvent function is called for this purpose.

- The CAppleEventObject class returns an event-not-handled error code for this event type. You can override the DoSetDataEvent function in any class that inherits functionality from the CAppleEventObject class.

♦ Notify Start Recording — An event in the Core Suite that calls upon the application object (`gApplication`) to handle the event in its DoAppleEvent function. That function calls the SetRecording function with an argument of `TRUE` to set the `recording` variable to `TRUE`.

♦ Notify Stop Recording — An event in the Core Suite that calls upon the application object (`gApplication`) to handle the event in its DoAppleEvent function. That function calls the SetRecording function with an argument of `TRUE` to set the `recording` variable to `FALSE`.

♦ Application Died — An event in the Core Suite for which a handler is installed, but no code is provided. When the event occurs, an event-not-handled error code will be returned to the sender. If you wish to handle this event, you can override the DoAppleEvent function in your CApplication-derived class and provide the functionality you require.

♦ Begin Transaction — An event in the Miscellaneous Suite that provides the means to specify a transaction ID to a server application, follows that event with a series of other events bearing the same transaction ID, and then terminates the transaction by sending an End Transaction event for that transaction ID. The TCL recognizes the Begin Transaction event; however it does not process it and returns an event-not-handled error code to the sender. If you wish to handle transactions, then you can override the DoAppleEvent function in your CApplication-derived class and provide the necessary functionality. In so doing, your application must not allow events with other than

the specified transaction ID to be processed, until the End Transaction event has been received.

- Copy — An event in the Miscellaneous Suite that requests that the current selection be transferred to the clipboard. The process of determining the current selection begins in the application object (`gApplication`), which determines which of its CDirector objects is currently active. The GetSelection function of the active director object is called to ascertain which of its windows is active. That window searches through its list of subviews (`itsSubviews`) and accesses the subview that is the current gopher. The gopher is cast as a CAppleEventObject instance, and its GetSelection function is called to access the current selection. The current gopher need not be the selection in Apple event terms, but to implement the event, it must know how to return a CAppleEventObject pointer that represents the current selection object. Copy is not implemented in the TCL; however, you can do so by creating a subclass of any of the CView-derived classes and implementing GetSelection and DoAppleEvent functions for it.

- Cut — An event in the Miscellaneous Suite that requests that the current selection be removed and stored into the clipboard. As for the Copy event, Cut is not implemented in the TCL; however, you may do so by following the prescription given for the Copy event.

- DoScript — An event in the Miscellaneous Suite that requests that the application object perform the specified script. This event is not implemented in the TCL; however, if you wish to do so, you can override the DoAppleEvent function in your CApplication-derived class to do so.

- End Transaction — An event in the Miscellaneous Suite that requests that the server application terminate a transaction (a series of events with a common transaction ID). See the Begin Transaction event for a description of how this works.

- Is Uniform — An event in the Miscellaneous Suite that asks whether a set of objects has the same value for a specified property. The TCL does not implement this event; however, you can do so by overriding the DoAppleEvent function in your CApplication-derived class.

- Paste — An event that requests that a copy of the contents of the clipboard be stored into the current selection or be inserted at the current insertion point. This event, like Copy and Cut, is not implemented by the TCL, but you can do so by providing the GetSelection and DoAppleEvent functions for subviews that need to handle this event.

- Redo — An event in the Miscellaneous Suite that requests that the results of a previous undo operation be reversed. The TCL does not implement this event; however, you can do so by implementing the GetSelection and DoAppleEvent functions for subviews that are able to handle Undo/Redo events.

- Revert — An event in the Miscellaneous Suite that restores a set of objects with the version of the objects that was most recently saved. This event is not implemented in the TCL; however, you can do so by implementing the methodology for accessing the application's objects and returning the token (object class and pointer information) pertaining to the most specific object indicated in the event. We will cover this topic shortly.

- Transaction Terminated—An event in the Miscellaneous Suite that informs an application that a transaction in progress (one that has been begun with a Begin Transaction event and has not yet been ended with an End Transaction event) has been terminated. This event is intended to merely inform the server application that the event has occurred. No action is required, other than forgetting that a transaction was in progress. The event is not implemented in the TCL.

- Undo—An event in the Miscellaneous Suite that requests that the results of the last action on a set of objects be undone. The TCL does not implement this event; however, you can do so by implementing the GetSelection and DoAppleEvent functions for subviews that are able to handle Undo/Redo events.

- Get AETE — An event in the AppleScript Suite that requests that the application return its AETE resource to the requesting application. The TCL implements this function in the DoGetAETEEvent function of the CApplication class by accessing

the AETE resource of the application and returning the contents of the resource in the reply.

Handling Object Specifiers in Events

Many Apple events require an "object specifier" as the direct object (or other parameter) of the event. Such an object specifier might refer to *the title of the first document in the application.* In this case, the Apple event would contain descriptors that require the application to return a token that references the *first document* object, then that object would be requested to return a token representing its *title* property. This would satisfy the request and complete the handling of the event.

It is important to point out that the method by which a particular event is dispatched to a specific object in your application is accomplished with a combination of facilities that are present in the Apple Event Manager, the Object Support Library, and also by means of "helper" functions that you install to aid the Apple Event Manager's AEResolve function to perform this task.

Installing an Object Accessor Function

The TCL installs a single "object accessor" function to aid in resolving object specifier references. This accessor is installed during the initialization process of the CSwitchboard class and the code is as follows:

```
void CSwitchboard::InstallObjectAccessors()
{
   InstallObjectAccessor(typeWildCard,
                         typeWildCard,
                         MyAccessObjectUPP);
}
```

The foregoing code specifies that an object accessor function pointer called MyAccessObjectUPP is to be used for resolving objects of *any* class and *any* container type (that is, the type codes are both of `typeWildCard`, indicating that they will match any type). The MyAccessObjectUPP symbol is defined to be a pointer to the MyAccessObject function of the CAppleEventObject class.

Accessing the Event's Direct Object

When an Apple event arrives in the event queue and is dispatched to the appropriate handler, that handler will usually call the Get-

DirectObject function in the CAppleEvent class to access the direct object parameter in the event. If the event is sent to the application object, then the TCL needs only to call the DoAppleEvent function in the CApplication class.

If the direct object is an object specifier (that is, it refers to an object class, such as `cDocument`), the GetDirectObject function calls the Resolve function of the CAppleEventObject class to resolve the class-type to a pointer to the appropriate object of that class.

The TCL knows only how to find the application object (`gApplication`) when an event is first being handled, so the event must specify the "null" container for the object being sought. The "null" container is the outermost container of the Apple event Object Model (OM). Because of this, the Resolve function (which calls the AEResolve toolbox function in the Apple Event Manager) will call the MyAccessObject function that was installed as the object accessor function for all types of objects. Recall that the MyAccessObject accessor specified "wild card" values for both the class and container type codes.

When the MyAccessObject function gains control from the AEResolve function of the Apple Event Manager, it is handed the type code of the desired class (for example, `cDocument`), a descriptor token for the container, a type code for the container class (for example, `typeNULL`), a type code for the "key form," a pointer to the "key data," a pointer to the target token's descriptor, and the value of the reference constant. The function creates a data structure that contains the desired class, container class, key form, key data, and reference constant values and then calls the MapDesc function with the container token's descriptor pointer, a pointer to the sAccessObject function, a pointer to the data structure, and a pointer to the result token descriptor.

The MapDesc function calls the specified function for a descriptor or for each element of a list of descriptors in the container token's descriptor structure. Because a container hierarchy is specified as a nested list of descriptors, each container object being represented by another descriptor, the MapDesc function traverses the hierarchy, calls the sAccessObject function, which, in turn, calls the AccessObject function for each container in the list until the specific target object has been located. So in the course

of locating the title of a specified document, the AccessObject function for the application object (`gApplication`) is called to locate the specified document object, and then the AccessObject function for the document object is called to locate its title property to satisfy the request.

Handling Events in the Application Class

The AccessObject function in the CApplication class is equipped only to handle requests for `cProperty` objects, which include the clipboard (`pClipboard`), insertion location (`pInsertion`), user selection (`pUserSelection`), and properties of the application itself, such as whether it is the frontmost application (`pIsFrontProcess`), the application's name (`pName`), and the application's version (`pVersion`). If the desired class is any other than the foregoing, then AccessObject calls the base class function in the CAppleEventObject class to handle the request.

When an event requests that an action be performed for some other object, it specifies which object of that type is being referenced. For example, it might specify the *first document* or the *document named Notes*. In either case, the AccessObject function in the CAppleEventObject class handles the request as follows:

- The property of an Apple event object is defined as an Apple event direct object whose container is the object to which the property belongs. The object specifier record for a property of the CApplication class specifies `cProperty` as the container and a constant such as `pClass` as the requested property (requesting the name of the class to which the property belongs). If the container (desired class) is `cProperty` and the property is `pBestType`, `pClass`, or `pDefaultType`, the AccessObject function creates a CProperty object containing the requested property and the current object (the application) as its container object, and a type code of `cType`. It uses the pointer to this object to call the MakeToken function to create a descriptor whose `descriptorType` field contains the constant `kTokenType` and a `dataHandle` field that contains a pointer to a pointer (that is, a handle) to the current object (that is, the CProperty object). A pointer to the token is stored into the `theToken` argument of the AccessObject function (as its result object pointer) and the AccessObject function returns.

- If the desired class is `cProperty` and the requested property is `pIndex` or `pID`, then the property type is set to `pLongInteger`. If the property is `pName`, then the property type is set to `cChar`. In any of these cases, a CProperty object is constructed to hold the current container object, the requested property, and the type code. Then the pointer to this object is used to call the MakeToken function to create a token that points to the CProperty object with a descriptor type of `kTokenType`. The token pointer is stored into the `theToken` argument of the call and the AccessObject function returns.

- If the desired class of the descriptor is not `cProperty`, then the `keyForm` parameter is tested. The `keyForm` parameter in an object specifier record indicates how the `keyData` parameter is to be interpreted. Several `keyForm` values are handled in the AccessObject function of the CAppleEventObject class, as follows:

  - If the `keyForm` parameter contains the `formAbsolutePosition` value, then the CountObjects function is called with the desired class and a pointer to a long integer variable to hold the resulting count. Because this code is executing for the CApplication-derived object (`gApplication`), the CountObjects function for that class is called.

    If the desired class is `cWindow`, the CountObjects function of the CApplication class calls the GetNumWindows function of the CDesktop object (`gDesktop`) to return the number of windows it is managing. If the desired class is `cDocument`, then the CountObjects function iterates through its list of directors (`itsDirectors`), testing each to determine whether it is a member of the CDocument class and incrementing the count if so.

    After the count of the desired class of objects has been determined, the `descriptorType` field of the `keyData` descriptor is tested to determine whether it is equal to the `typeLongInteger` value. If so, then the index value held in the descriptor is accessed and tested against the count value to ensure that the index is within the range of the count. If not, a failure exception is raised; otherwise, the GetElementByIndex function is called with the desired class, the index value, and a pointer to the `theToken`

argument, into which the result token pointer is to be stored.

The GetElementByIndex function in the CApplication class tests whether the desired class is `cWindow`. If so, then it calls the NthWindow function of the CDesktop object to return the pointer to the window whose index is specified in the call. The CWindow object pointer that is returned by that call is used to call MakeToken to encapsulate the returned object with a `descriptorType` of `kTokenType`. After doing so, the function returns and the AccessObject function that called it also returns, having resolved that portion of the request. If the GetElementByIndex function determines that the desired class is `cDocument`, then it iterates through its list of directors (`itsDirectors`), searching for the one that is a member of the CDocument class, whose index matches the argument to the function. After finding the specified object, it calls MakeToken to encapsulate the object and then returns. The AccessObject function then also returns, having satisfied the current portion of the Apple event request.

If, instead of `typeLongInteger`, the `descriptorType` of the `keyData` descriptor is `typeAbsoluteOrdinal`, then there are several possibilities to handle. An ordinal is a constant value such as `kAEFirst`, `kAEMiddle`, `kAELast`, `kAEAny`, or `kAEAll`. The meanings of these are fairly self-explanatory. The AccessObject function handles each of these by calling the GetElementByIndex function for the container object's class (for example, CApplication). The value of the index is 1 for the `kAEFirst` ordinal value, is the value of half of the count (plus one) for `kAEMiddle`, is the value of the count for `kAELast`, and is a randomly selected value in the range of 1 through the value of the count in the case of `kAEAny`. In the case of `kAEAll`, a list is created for the `theToken` descriptor and then GetElementByIndex is called to access each entry for the desired class, in the range of 1 through the value of the count, appending the returned token to the list. The AccessObject function returns after the requested object token (or list of tokens) is stored.

- If the `keyForm` parameter contains the `formName` value, then the name string is accessed from the `keyData` field, and the GetElementByName function is called with the desired class, the name string, and a pointer to where the result token should be stored.

  The GetElementByName function is implemented in the CAppleEventObject class (from which the CApplication class is derived). It performs the task of locating the object of the desired class by calling the CountObjects function with the desired class and a pointer to a long integer variable in which to place the result. It then iterates through the set of objects of the desired class by calling the GetElementByIndex function with index values that vary from 1 to the value of the count. In each iteration it calls the GetElementName function for each returned token's object pointer and compares the returned name with the specified name to determine whether it matches. If any returned token's name matches the specified name, then that token is stored into the `theToken` argument and the operation is complete. If none of the objects has a matching name, then the request has failed and an `errAENotAnElement` error value is returned in the reply to the event.

  The GetElementName function of the CDocument class returns the name of the file if a file exists, the title of its window if a file does not exist but a window does exist, or `NULL` if neither exists. The GetElementName function of the CWindow class returns the window's title.

- If the `keyForm` parameter contains the `formUniqueID` value, then the AccessObject function accesses the long integer value held in the `keyData` descriptor and calls the GetElementByID function with the desired class, the ID value, and a pointer to where the result token is to be stored.

  The GetElementByID function is implemented in the CAppleEventObject class (from which the CApplication class is derived). It performs the task of locating the object of the desired class by calling the CountObjects function with the desired class and a pointer to a long integer variable in which to place the result. It then iterates through the set of objects of the desired class by calling the GetElementID

function and then matching the returned ID value with the specified ID value, looking for equality in the two identifiers. (*Note:* A unique element ID is assigned to each CAppleEventObject when it is constructed.) If an object with a matching element ID is found, its token is stored in the `theToken` result argument to the AccessObject function and the function returns.

Handling Events in Other Classes

You have read in the foregoing section how a CDocument or CWindow object can be accessed, either by name or position, using the AccessObject function. This section is concerned with the ability to access items contained by those objects by using the same mechanism, with the exception that the AccessObject function for the CDocument or CWindow class is called to perform the function of locating the desired class.

Object specifier records consist of a set of nested Apple event descriptors that specify the eventual target object in terms of a "container hierarchy" that begins with the application object and proceeds to locate the object by naming each of its enclosing objects, in turn, from the outermost to innermost enclosure.

An Apple event requesting a particular property of a specified window will specify the default container (`typeNULL`) as its initial container and then call the AccessObject function of the CApplication object to locate the specified `cWindow` object. The token returned for the `cWindow` object is used by the Apple Event Manager to call the MyAccessObject (object accessor) function for the CWindow object's desired class of `cProperty`, to access one of several window properties.

In order to implement object model support in your applications, you should read the *Apple Event Registry* to determine the appropriate way to organize your classes to make use of the implied framework of the Object Model. Then you should derive the classes for which you intend to provide Apple event support, from the CAppleEventObject class. If certain events are not implemented for your application, the TCL will send the appropriate `errAEEventNotHandled` response. You should be careful to create your `AETE` resource such that it doesn't imply support for events that are truly not supported in the application.

Classes derived from CAppleEventObject will need to override the GetClassID, GetDefaultType, GetContainer, and GetElementName functions. Any events that a base class handles will have to be handled by an override to that function in your new class. If your class handles any events that are not handled by any of the base classes, then you will have to write a new function to handle that event and override the DoAppleEvent function to call the appropriate function for each of those events.

If your new class has properties, then you will need to override the DoPropertyGetDataEvent and DoPropertySetDataEvent functions. If the class supports any properties that are not supported by any of its base classes, then you will need to override AccessObject and call MakeToken to create appropriate tokens for the new properties.

If your class has elements that are not present in any of its base classes, then you will need to override the GetElementByIndex and CountObjects functions. For example, if your application intends to support the access to paragraphs, sentences, and words in a text object, you must first derive that object from CAppleEventObject and then, perhaps, CEditText or CAbstractText. To handle access to any of the text class's elements, you should implement *all* of the foregoing mentioned functions.

Handling Object Information Accesses

As mentioned earlier, your new class should override the GetClassID, GetDefaultType, GetContainer, and GetElementName functions. The data returned by these functions is as follows:

- The GetClassID function should return a defined OM class ID, but only if the base class does not already do so. In the case of the text object mentioned earlier, you should return `cText` as the OM class.

- The GetDefaultType function should return the default type of the data returned by the object. In the case of the text object that we have been using as an example, a default type of `typeChar`, indicating an unterminated string of text, is returned. If your application can return various types of data, then you may have to override the GetBestType function. CAppleEventObject implements GetBestType by calling the GetDefaultType function. If this is not appropriate (for example, the "best" type

of data handled by your text object is styled text), then you should override GetBestType to return a type code that pertains to the data you intend to return if the "best type" is requested.

- The GetContainer function should return a pointer to the C++ object that is the container for the specified object. Usually this is the enclosure (`itsEnclosure`) for the object, but may be the supervisor (`itsSupervisor`) for the object, as is the case for the CDocument object, whose "container" is the application object (`gApplication`).

- The GetElementName function should return a string that is the name associated with the element. A text object may not have a name and so it should return a `MULL` string as the result in that case.

Comparing Objects

The TCL implements two comparison functions that provide the ability to compare two objects for equality (EqualObject) or perform more complex comparisons such as less than, greater than, begins with, ends with, contains, and others. The logic for performing these comparisons is triggered by the presence of comparison operators in the events themselves. If the supplied CompareObjects function does not provide the functionality you require, then your CAppleEventObject-derived object should override the CompareObjects function.

Events containing clauses such as "every window whose name begins with 'untitled'" are handled automatically by the Apple Event Manager by calling the counting function to ascertain the number of windows and then calling the compare function to determine each window's name. The "whose" and "where" clauses are all handled in this way. You may have to supply a counting and/or compare function for your new class, but most of the work of handling complex events will be automated.

Support for Sending Apple Events in the TCL

Apple events can be sent from your application in several circumstances. If another application (or even your own application) makes a request, then the reply is in the form of an Apple event. You can also send Apple events to other applications using the

support functions in the TCL. And, finally, as indicated earlier, you can send Apple events to your own application. When an application is "factored," it separates the request from the action that implements the request. So by factoring your application, as the TCL itself is factored, you can write applications where every action can be commanded by your own or another application.

Replying to Apple Event Requests

When an Apple event is received, the TCL creates an "empty" response descriptor. Your application (or the TCL) can add descriptors to the response record, as required, to send the appropriate result back to the requestor. No overt action is required on your part to reply to an Apple event request.

As an example, consider the simple case where another application has sent an event requesting your application's name. The event is processed as follows:

1. The DoHighLevelEvent function of the CSwitchboard class recognizes the high-level event and calls the AEProcessAppleEvent function of the Apple Event Manager.

2. The Apple Event Manager calls the GenericResultHandler function in the CAppleEventObject class that we installed to handle Get Data events (`kAEGetData`).

3. The GenericResultHandler function calls the GenericHandler function with the event and result descriptors and also a pointer to the DispatchResult function.

4. The GenericHandler function calls the PackageAppleEvent function to create a CAppleEvent object with the event, its reply, and the reference constant, and then calls the DispatchResult function to continue processing.

5. DispatchResult calls the GetDirectObject function in the CAppleEvent class. This accesses the object specifier record in the incoming event; the desired class will be `cProperty` (indicating that the request is for a property), the container will be a null descriptor record (the application has no container), the key form descriptor will have a descriptor type of `typeEnumerated` and its data will be `'prop'` (`formPropertyID`), and the key data will have a property type of `pnam`

(pName). After accessing the direct object, DispatchResult calls the Resolve function of the CAppleEventObject class with the direct object descriptor to be resolved and a pointer to where the resolved object descriptor is to be stored. The Resolve function calls the AEResolve function of the Apple Event Manager to perform the resolution. In this case, the AccessObject function in the CApplication class will be called first.

6. The AccessObject function in the CApplication class tests whether the desired class is `cProperty`. If so, it tests whether the property is one of several that it handles, including the pName property. In the case of pName, it constructs a CProperty object with `this` (the application pointer) as the object, pName as the property ID, and a descriptor type of `cIntlText` (international text). The pointer to the CProperty object is used to call the MakeToken function to construct a token whose descriptor type is `kTokenType` and whose data handle points to the CProperty object. At this point, the direct object has been resolved, so execution continues in the DispatchResult function of the CAppleEventObject class, which calls the MapDesc function to map the DispatchResult1 function against the direct object token that was just constructed.

7. The DispatchResult1 function accesses the token with which it is called and calls that token's DoAppleEvent function with the event and the result descriptors.

8. The CProperty class, which is the object class associated with the token, does not implement the DoAppleEvent function; however, the function inherited from the CAppleEventObject base class is called. That function handles the Get Data event by calling the object's DoGetDataEvent function, which *is* implemented in the CProperty class.

9. The DoGetDataEvent function in the CProperty class accesses the original event descriptor to acquire the optional result type parameter and then verifies that there are no more required parameters in the event. That being accomplished, the DoGetDataEvent function accesses the "object" associated with the CProperty object, which in this case is the application object (indicated as `this` in the foregoing step 6). The

application object pointer is used to call the DoPropertyGetDataEvent function in the CApplication class with the property, the requested type, and a pointer to the result.

10. The CApplication class does not implement the DoPropertyGetDataEvent function, so that function inherited from the CAppleEventObject class is called. The function determines which property has been requested and calls the GetElementNameDesc function.

11. The GetElementNameDesc function calls the GetElementName function in the CApplication class to access the application's name string and then packages the name in a descriptor that it returns as the result to the GetDataEvent function in the CProperty class.

12. The GetDataEvent function in the CProperty class calls the CoerceDescList function to coerce the result (using either its built-in or user-supplied coercion handlers to do so), storing the result into the final Apple event reply descriptor.

At this point, processing of the event is complete, and the foregoing routines return back through the calling chain to return control to the AEProcessAppleEvent toolbox function, which sends the reply to the requesting application.

Sending Apple Event Requests

The TCL contains support classes and member functions that aid in the creation and sending of Apple events, either to your own application (in the event that your application has been factored—more on that later), or to another application.

Sending an Event to Yourself

If your application is factored—I will talk more about the topic of factoring shortly—and you wish to send an event to yourself, then the TCL provides the MakeEventToThis and SendEventNoReply functions in the CAppleEventObject class.

An example of sending an event to your own application would be the decision to quit the application when in the midst of doing something else. You might also want the user to determine whether or not to quit before saving any results. When factoring is enabled for an application, the **Quit** command is handled in just

this way. The code in the DoCommand function of the CApplication class for accomplishing this is as follows:

```
        case cmdQuit:
           if (Factoring())
              SendAEQuit(kAEAsk);
           else
              Quit();
           break;
```

Note that the SendAEQuit function is executed only if the call to the Factoring function returns a nonzero result. The code for the SendAEQuit function is as follows:

```
void CApplication::SendAEQuit(DescType saveOption)
{
   AppleEvent event;

   MakeEventToThis(kCoreEventClass, kAEQuitApplication,
      FALSE, &event);
   CWatchDesc watch(event);
   FailOSErr( AEPutParamPtr(&event, keyAESaveOptions,
      typeEnumerated, &saveOption, sizeof(saveOption)) );
   SendEventNoReply(&event, kAEAlwaysInteract);
}
```

Note in the foregoing that the code defines an AppleEvent variable called `event` and then builds an event by calling the MakeEventToThis function with the event class (`kCoreEventClass`) and the event type (`kAEQuitApplication`), as well as a Boolean value indicating whether a direct object is to be included in the event, and then a pointer to the AppleEvent variable into which the event is constructed.

After the event has been constructed, a CWatchDesc object is constructed that includes the `event` variable in its construction. When the `watch` object is constructed, the `event` variable reference is stored into the `watched` member variable of the CWatchDesc object. If the CWatchDesc object is disposed (as it will be when the SendAEQuit function exits), the destructor for the CWatchDesc class will recognize that the `watched` variable contains a nonzero pointer and the destructor will dispose the descriptor (AppleEvent) variable. The AEPutParamPtr function call adds a parameter with the specified key (`keyAESaveOptions`) to the event, the specified data type (`typeEnumerated`), a pointer to the data (`saveOption`), and the size of the data item.

After the event has been constructed fully, SendEventNoReply is called to send the Apple event, with the user interaction specification of `kAlwaysInteract`.

Sending an Event to Another Process

If you want to send events to another process, then the TCL contains a great deal of help for doing so. You will have to create a CAppleEventSender object, create a descriptor for the event you intend to send, add any needed parameters to the event, and then call one of the Send functions for the object to send the event.

Apple events sent to another process require the Process Serial Number (PSN) of the process to which the event is to be sent. The CAppleEventSender class has a member function called FindProcess that loops through the existing processes, looking for one whose Creator and Type codes match the ones you provide.

An example of a function that sends a quit event to an application whose signature is 'Nsbl' is as follows:

```
void CApp::TellNsblToQuit()
{
   ProcessSerialNumber psn;
   OSErr err;

   err = CAppleEventSender::FindProcess ('APPL', 'Nsbl', &psn);
   if (err == noErr)
   {
      CAppleEventSender event (kCoreEventClass,
         kAEQuitApplication, &psn);
      event.SendNoWait (kAENoReply);
   }
}
```

Although the foregoing code is rather simple, it should give you an idea of how to use the CAppleEventSender class both to find an existing process by using its creator and type codes, and also to send an event to that process. Obviously, if you need to construct complex events that make use of object specifiers and the like, you can use the PutParamDesc or PutParamPtr functions to add descriptor or buffer pointer parameters to the basic event. The CAppleEventSender class also supports the ability to send an event and wait for a reply, with a specified time-out value.

Adding Factoring and Recording Support

A factored application is one in which the code that recognizes the request to perform an action and the code to perform the action are separate. The TCL is highly factored and handles almost every user-specified action by sending an Apple event to itself to perform the action. If you factor your own application in this same way, then all of the application's actions will be recordable by any scripting component. Recording by a scripting component is accomplished by watching the Apple events your application receives and recording these. Because the TCL is already factored, it is also set up to support recording.

When your application is initialized, the `factoring` variable is set to `kFactorWhenRecording` and the `recording` variable is set to `FALSE`. So, by default, factoring will not be enabled. There are three settings for the `factoring` variable. These are `kNeverFactor`, `kAlwaysFactor`, and `kFactorWhenRecording`. You can specify the factoring setting by calling the SetFactoringLevel function with one of the foregoing choices. You can access the current factoring setting by calling the GetFactoringLevel function. You can specify the recording setting by calling the SetRecording function with a `TRUE` or `FALSE` value, and you can access the current recording setting by calling the GetRecording function, which will return a `TRUE` or `FALSE` result.

Apple Events Feature Summary

The TCL contains a great deal of support for handling Apple events automatically. Providing comprehensive handling of a large variety of Apple events (that is, making the application scriptable) is a nontrivial task; however, the TCL will help greatly in easing the burden of this task.

The major elements of the task are the creation and registration of handlers for each of the unique Apple events that the application intends to handle. It is also very important to create an `'AETE'` resource which describes the extent of your application's Apple event support. You can use Apple's ResEdit program to create or modify an `'AETE'` resource, but there are other products that make this process easier.

Creating a factored and scriptable (recordable) application requires a lot of work. You need to go into each of the DoCommand functions and separate the recognition of each command from the actions that perform it, write code to send an Apple event specifying the command and any of its parameters in the recognition section (usually in the DoCommand function or a function that it calls), and then provide code in a DoAppleEvent function to handle the event and perform the specified action. There are many examples of how this is done in the TCL source code.

The next chapter describes how to use chores, tasks, and the Undo/Redo features of the TCL.

Chapter 11

Understanding Chores, Tasks, and Undo/Redo

This chapter describes the various types of chores and tasks that you can schedule for execution using the services of the TCL. Each of the defined classes is either used by the TCL in the course of handling a specific task, or is available to be used as a base or concrete class. The sections that follow discuss the various classes and how they are used. The class inheritance diagram for both tasks and chores is shown in Figure 11-1.

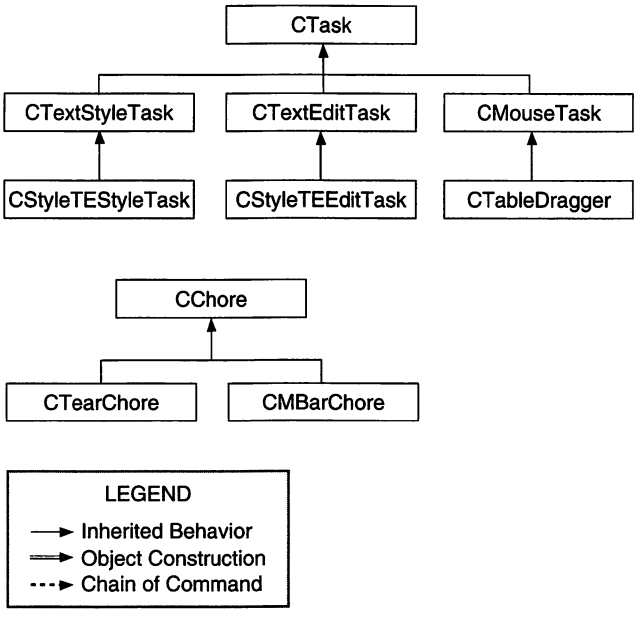

Figure 11-1
Class inheritance diagram for tasks and chores

Using Chores

There are two types of chores, but both are associated with the application as a whole, rather than a document or other object of your program. The CApplication class provides two queues in which chores can be installed. If you need to perform some peri-

odic operation at idle time, then you can create a subclass of the CChore class and override the Perform function to execute the chore. After creating an object of your derived class, you can install the idle-time chore by calling the AssignIdleChore function of the application object (`gApplication`). The chore will then be executed periodically, until it is removed by calling the application object's CancelIdleChore function.

The other type of chore is an "urgent chore." Urgent chores are tasks that must be performed at the next possible opportunity. They are executed only once and then are disposed automatically. After processing the current event, the event loop checks whether any urgent chores are present, executes all of them, and then disposes their pointers before processing the next event.

Creating a Periodic Chore

A perfect example of an idle-time chore would be one that checks the current time when it is executed, comparing that value against the time in the first member of a time-ordered list of reminder object entries. If the current time equals or exceeds the time value associated with the first entry in the list, then the chore might post a notification to the user. The header file of a simple class, derived from CChore, to accomplish this function is as follows:

```
//
// CTimeCheckChore.h
//    Header file for CTimeCheckChore class
//    Copyright © 1995, Richard O. Parker. All rights reserved.
//

#pragma once
#include <CChore.h>
#include "CReminderList.h"

class CTimeCheckChore : public CChore
{
   TCL_DECLARE_CLASS

public:
   CTimeCheckChore (CReminderList *aList);
   virtual void Perform (long *maxSleep);

private:
   CReminderList *theList;
};
```

The foregoing class definition includes the declaration of a constructor function and an override for the Perform function inher-

ited from the CChore base class. The class also declares a private variable that holds a pointer to the list of reminder objects (`theList`). For purposes of this example, we assume that the objects in the list are of type CReminder and that the CReminder class provides access functions to obtain the time that the reminder should be posted, and the text of the reminder message. We will not attempt to get too carried away with the form of the notification, and so we will not display the application's icon or play a sound when the notification is posted. You can certainly choose to do so in your own version of such a chore. The source file contents for the CTimeCheckChore class are as follows:

```
//
// CTimeCheckChore.cp
//    Source file for CTimeCheckChore class
//    Copyright © 1995, Richard O. Parker. All rights reserved
//
#include "CTimeCheckChore.h"
#include "CReminderList.h"
#include "CReminder.h"
extern Boolean gNotePosted;
extern struct NMRec gNMRec;

TCL_DEFINE_CLASS_M1 (CTimeCheckChore, CChore);

/*************************************************************
   CTimeCheckChore

      Constructor

*************************************************************/

CTimeCheckChore::CTimeCheckChore (CReminderList *aList)
{
   theList = aList;
   gNotePosted = FALSE;
}

/*************************************************************
   Perform

      Check the current time, compare it with the time
      associated with the first entry in the aList list
      and install a notification if the time has come
      to do so.

*************************************************************/

CTimeCheckChore::Perform (long *maxsleep)
{
   unsigned long curTime, noteTime;
   CReminder *theFirstEntry;

   // check to see whether a note is already posted
   // if so, then just return; otherwise, start checking.
   if (gNotePosted || (theList->GetNumItems <= 0))
   {
      *maxsleep = 300;    // 5 seconds
      return;
```

```
    }
    GetDateTime (&curTime);
    theList->GetArrayItem (&theFirstEntry, 1);
    if (curTime < theFirstEntry->GetNoteTime())
    {
       // it's not yet time to post the notification,
       // so just return, but after setting a sleep time
       // of about 30 seconds.

       *maxsleep = 1800;  // 30 seconds
       return;
    }

    // now it's time to issue a notification for the first
    // entry in the list, so we use the access function to
    // obtain the information for the message to display to
    // the user. The notification record structure is
    // declared as a global structure called gNMRec.

    gNMRec.qType = nmType;   // queue type == notification
    gNMRec.nmMark = 1;       // mark in the app menu
    gNMRec.nmIcon = NULL;    // no flashing icon
    gNMRec.nmSound = NULL;   // no sound
    gNMRec.str = theFirstEntry->GetNoteStringPtr();
    gNMRec.nmResp = NULL;    // no response procedure
    gNMRec.nmRefCon = 0;     // no reference constant
    NMInstall (&gNMRec);     // install the notification
    gNotePosted = TRUE;      // set note posted
}
```

The foregoing code is rather short and sweet. We are under the assumption that there is only one CTimeCheckChore object and that it will be executed approximately every 30 seconds (1800 ticks). When it is executed, it first checks a global variable called `gNotePosted` to determine whether a note has already been posted (the user can handle only one at a time) or if the list of reminders is empty. If either expression is `TRUE`, then we set the `maxsleep` variable to request that the function be called again, in about five seconds (300 ticks).

If the value of the `gNotePosted` variable is `FALSE`, then we can proceed to check whether the current time (using the millisecond clock value returned by the GetDateTime toolbox function) is later than the time in the first list entry (accessed by using the GetArrayItem function to access the object pointer and then using that to call its GetNoteTime access function). If the current time is less than the time in the first entry, then we set the `maxsleep` variable to 1800 ticks (30 seconds) and return.

If the current time is greater or equal to the time in the first entry, then we must create a notification record and call the NMInstall toolbox function to install it in the operating system queue.

Our notification specifies that no icon is to rotate in the application menu (System 7), no sound is to play, and no response procedure is to execute when the notification is complete. We do provide the text for the Notification Manager to display in a dialog to the user, by accessing a pointer to the text using the GetNoteStringPtr access function.

You may wonder how we are going to remove the entry from the list after the notification has been posted (it has to remain there until the user dismisses the notification because of the pointer to its text string in the notification). One way to do this is to require that the user pull down a **Reminder** menu and choose a **Remove Notification** command. Another way would be for our Perform function to install a pointer to a response procedure that the Notification Manager could call when the user dismissed the note. The response procedure can remove the first entry from the list (assuming it has access to it). However, if you choose to remove the entry from the list, you must also set the value of the `gNotePosted` variable to `FALSE`, allowing the Perform function to begin checking the new first entry in the list, if any.

How the TCL Uses Urgent Chores

Urgent chores are tasks that must be executed at the earliest possible time. As described earlier, these are executed immediately after the current event has been processed. Urgent chores are executed only once and then are disposed automatically.

The TCL includes two classes derived from the CChore base class. These are shown in Figure 11-1 as CTearChore and CMBarChore.

Understanding the CTearChore Class

The CTearChore class implements an urgent chore that causes a torn-off menu's window to be displayed at the point where the user has released the mouse button. The process by which this occurs is somewhat complex, but it is instructive to review how tear-off menus work, so we will cover that, with reference to the Tools menu that we created in Chapter 4 (see page 163), at this point. The various facts regarding a tear-off menu are as follows:

- The standard toolbox menu definition procedure handles only normal menus. So tear-off menus require a custom menu definition procedure to work properly.

- When the CTools class is constructed (see page 168), the constructor code calls the IViewPICTGridDirector function of its CPictGridDirector base class. The arguments to the function are the name of the Tools floating palette view (Tools) and the associated menu ID (129). The IViewPICTGridDirector function creates the floating palette and stores its pointer into the `itsWindow` variable of the director. Then that function creates a CGridMDEF object and passes its constructor the Menu ID, a pointer to the CPICTGrid pane that contains the tools to be displayed in the menu, and a pointer to the CTools object (`this`).

- The constructor of the CGridMDEF class causes the constructor initializer for its CSelectorMDEF base class to be called, which, in turn, causes the constructor initializer for its CPaneMDEF base class to be called. The constructor for the CPaneMDEF class causes the constructor initializer for the CMenuDefProc base class to be called with the menu ID as its sole argument. That constructor function loads the `MDEF` resource whose ID matches the menu ID passed to the function. The `defProc` field of the resource is set to the address of the GenericMDEF function (`sGenericMDEF`), and the `itsMenuDefProc` field is set to point to the CGridMDEF object (`this`). After the constructors of the base classes have completed execution, the body of the CPaneMDEF constructor executes and stores the CPICTGrid object pointer into the `itsPane` member variable, and the pointer to the CTools object into the `itsTearOffMenu` member variable.

- At this point, the menu definition procedure has been installed, and the MenuSelect toolbox function can jump to the first location of the `MDEF 129` resource (which itself contains a `JMP` instruction whose jump target has been patched to call the GenericMDEF function [`sGenericMDEF`]). That function handles the messages to draw the menu (`mDrawMsg`), handles selections from the menu (`mChooseMsg`), determines the menu's dimensions (`mSizeMsg`), and specifies placement for a pop-up menu (`mPopUpMsg`). When the user clicks on the

Tools menu, the sGenericMDEF function will receive the messages to calculate the menu's size, draw the menu, and then report the user's choice. It is in this latter message handler that the tear-off functionality occurs.

- After the menu's dimensions are known and its contents are drawn, then the `mChooseMsg` handler is called. That handler calls the ChooseItem function of the CGridMDEF class (the specific C++ menu definition procedure) to respond to the user's menu choice. If the current mouse position is outside the original menu rectangle, then the choice is set to `NOTHING` and the TearOffMenu function of the CPaneMDEF base class is called with arguments of the menu's original rectangle and the current "hit point."

- The TearOffMenu function is responsible for tracking the mouse, displaying the gray outline of the menu being torn off, and then causing the torn off version of the menu to be displayed in a floating palette. The final step of the foregoing procedure is handled by calling the TornOff function of the menu's CTearOffMenu class (`itsTearOffMenu`), a base class of the CTools class, with the current mouse location.

- The TornOff function serves but one purpose: to create a CTearChore object, initialize the object with a pointer to the CTools object, and then call the AssignUrgentChore function of the application object. That chore object is installed into the application's urgent chores queue and will be executed immediately after processing of the current event is complete.

- The Perform function of the CTearChore object, when called by the event loop when urgent chores are processed, uses the pointer to the tear-off menu's window director object (CTools) to call the MoveToCorner function inherited from its CFloatDirector base class.

- Finally the MoveToCorner function moves the top-left corner of the director's window to the location at which the mouse cursor was released, makes the window visible, and selects it to make it frontmost and active.

The final step in the foregoing sequence completes the process of tearing off a menu and then displaying it as a floating palette. The purpose of using the urgent chore is to cause the window to be

shown only after processing of the tear-off action is complete, and also to separate the code related to handling the menu from the code that handles the window. Any selection of a tool from the window, once it has been displayed, will be treated as a window item selection, rather than a menu selection, even though the distinction between the two is fairly blurred.

I know that the foregoing is rather complex; however, it is interesting to review the process by which tear-off menus are handled, even though this is somewhat incidental to how an urgent chore is created and processed.

The CMBarChore object is created by the CBartender object (`gBartender`) and is scheduled as an urgent chore for all operations that require the menu bar to be redrawn during the processing of the current event. By executing the chore just once, the TCL prevents any unnecessary flashing of the menu bar.

You can create urgent chores for whatever purpose you desire. Just keep in mind that such chores are intended to be associated with the application—and its look and feel—as a whole. If it is necessary to create a chore that is associated with a specific document, then, perhaps, a task would be a better choice.

Using Tasks and Undo/Redo

Tasks are objects that are usually created to handle undoable actions. The TCL supports a number of these, and you can add tasks to your application to support other undoable actions.

You can create and use CTask-derived objects in two different ways, as follows:

1. You can perform an action, then create a CTask-derived object, storing enough information in that object to undo the action, and then call the Notify function of the current object's supervisor (for example, the CDirector-derived object).

2. You can create a CTask-derived object and then call its Do function (which you must override) to perform the action. After that, you must call the Notify function for the current supervisor (for example, the CDirector-derived object).

The Notify function of the CBureaucrat class that is inherited by the majority of the objects that might participate in an undoable action merely calls the Notify function of the object's supervisor (`itsSupervisor`).

When the Notify function is called, a pointer to the caller's window (if any) is saved in the `cTaskWindow` variable, and then the NotifyClean function of the application object (`gApplication`) is called. That function disposes of any task object that is currently stored in the application's `lastTask` variable, and then the pointer to the current task is stored into that variable. In addition, the undone variable is set to a value of `FALSE`.

Creating a Text Style Undoable Action

An example of an undoable task that uses the second method, described earlier, is the case of a style command being applied to a text subview that is derived from CEditText or its CAbstractText base class. In either of these cases, the DoCommand function determines whether a style change (for example, `cmdBold`, `cmdItalic`, `cmdAlignCenter`, and so on) has been commanded, and the MakeStyleTask function inherited from the CAbstractText class is called to create a CTextStyleTask object, initialize it with the style command to be executed, and return the CTextStyleTask object to the DoCommand function. DoCommand continues by saving the pointer to the CTextStyleTask object, calling the Notify function of the CAbstractText-derived object's supervisor, and then calling the task's Do function to perform the task.

The task's Do function saves the current style, spacing, and alignment settings for the object to which it applies, performs the specified style change command, and returns to the DoCommand function that called the task's Do function.

If the user decides to undo the previous action and pulls down the **Edit** menu to choose the appropriate command, the UpdateMenus function will be called for each of the CDirector-derived objects. The Undo/Redo operations are managed by the UpdateMenus function in the CApplication class. That function performs its normal operations and then tests whether the `lastTask` variable contains other than a `NULL` pointer. If so, then the CBartender object (`gBartender`) is called to enable the

cmdUndo command in the Edit menu, and then the UpdateUndo function is called.

The UpdateUndo function accesses the 'STR ' resource that corresponds to either the Undo or Redo text, depending upon whether the undone variable has a value of TRUE or FALSE. Initially, the value in this variable will be FALSE, and the string "Undo" will be accessed. The function then uses the pointer stored in the lastTask variable to access the task's name index value. The index addresses the n^{th} string in another 'STR ' resource to append the text representation of the task to the "Undo" string. The result is written into the command text for the **Undo** command in the **Edit** menu. Thus, the command text would read "**Undo Formatting**" in the case of a text style change.

When the foregoing command is chosen, the DoCommand function in the CApplication class determines whether the undone variable is TRUE or FALSE. In the current example, the value is still FALSE, so the DoCommand function will call the Undo function for the task whose pointer is stored in the lastTask variable. Had the undone variable been TRUE, then DoCommand would have called the Redo function of the task. The Undo function of the CTextStyleTask object saves the current style, spacing, and justification attributes, restores the corresponding previous values of those attributes, performs the operations to restore the style of the text object, calls the CDirector object's ToggleChanged function to set the document's status as unchanged or changed, depending on the previous setting, and, finally negates the value of the undone variable, making it TRUE if it was FALSE or FALSE if it was TRUE. The Redo function inherited from the CTask base class merely calls the Undo function that is overridden by the CTextStyleTask object. Even though the command text changes to "**Redo Formatting**," the operations necessary to perform the function are handled entirely by the Undo function.

Creating a Task for Mouse Tracking

The CTable class in the TCL implements a one- or two-dimension table, in which one or more cells can be selected by clicking the mouse pointer on a cell and dragging the pointer to encompass additional cells, if this behavior is allowed.

When the user clicks on a table cell, the DoClick function of the CTable class is called to handle the click. If the DoClick function finds that the click is inside the table's bounds, then it checks first to see whether it is a double-click. If so, then the command (if any) associated with the CTable object is dispatched by calling the DoCommand function with the cell number in which the click occurred. If the click is not a double-click, then the DoClick function calls the MakeMouseTask function to construct a CTableDragger object, which is returned to the DoClick function. The CTableDragger object is a task that inherits functionality from the CMouseTask and CTask classes. The DoClick function sets the mouse tracking task in motion by calling the TrackMouse function in the CPane class, passing it a pointer to the task object, the point where the mouse click occurred, and the table's bounds.

The TrackMouse function performs the mouse-tracking function as follows:

1. The task's BeginTracking function is called to initialize the task with the beginning mouse location. The BeginTracking function also determines the "rules" by which cells can be selected for the current table, by checking whether only one or multiple cells can be selected and whether selected cells can be non-contiguous. The cell in which the mouse is clicked is selected, and the selection is added to or replaces the existing selections, depending upon whether multiple and/or non-contiguous selections are allowed.

2. The TrackMouse function then enters into a loop that runs while the mouse button is still down. Each time through the loop, the task's KeepTracking function is called with the current mouse location, the previous mouse location, and the initial mouse location. The KeepTracking function determines the cell in which the current mouse location lies and scrolls the table to bring that cell into view, if necessary. After doing so, depending upon the selection criteria, the new cell is selected and, perhaps, added to the previous selections.

3. When the mouse button is released, the TrackMouse function calls the task's EndTracking function with the current point, the previous point, and the initial point. The EndTracking function performs but one function. It deletes the current task by calling TCLForgetObject with `this` as its argument.

The foregoing steps show how a task can be used to handle operations that are unique to a specific user interface object. In this case, the task is not an undoable task—after all, you can't undo the user's dragging of the mouse—but the code in the TrackMouse function of the CPane class is generic enough that it can be used by a variety of task objects to handle tracking the mouse for numerous applications. For example, it is easy to picture the creation of a mouse task that allows the user of a drawing program to drag objects around on the screen. A similar mouse task could be used to provide the drawing function itself. If you wish to create a mouse task for your application, then derive the task object from the CMouseTask class, as was done for the CTableDragger class.

Chores and Tasks Summary

This chapter has described how to you can use chores to perform periodic actions or urgent actions. The TCL keeps two lists of chores for these purposes. The first list of chores contains pointers to procedures that are to be performed when no other events are pending. These are idle chores and are often used for tasks that must be executed periodically. Once an idle chore is entered into the list, it remains in the list until it is removed explicitly.

The second list contains pointers to procedures that must be executed as soon as is practically possible. These chores are executed only once after being entered, but are executed within the event loop, immediately after the current event has completed execution. Urgent chores are often used to handle memory shortages and other urgent situations.

Tasks are used primarily to implement the Undo/Redo of the user's most recent action. A task will save the necessary context so that when an undoable action is taken, the effects of the action can be undone or redone. Tasks are also used to implement generic actions such as mouse tracking and the like.

The next chapter describes some of the nuances of drawing and printing, with emphasis on printing multiple windows and offset panes.

Chapter 12

Drawing and Printing

This chapter wraps up the discussion of the THINK Class Library by discussing the methodology used in the TCL for drawing and printing. In many cases, you will not have to do anything to have the contents of a window drawn or printed; however, in case that you do, the sections that follow describe this process.

Implementing the Draw Function

If you recall, in Chapter 7 we discussed how the Update event caused the DoUpdate function of the CSwitchboard class to call the Update function for the window whose pointer is held in the `message` field of the update event (see page 301).

The Update function in the CWindow class saves the current GrafPort pointer, sets the current window's GrafPort pointer, calls the BeginUpdate toolbox function with the window's pointer, then calls the DeviceLoop toolbox function with the window's visible region (`visRgn`), a pointer to the DoUpdateDraw function as the drawing function to perform, the window's object pointer (`this`) as user data to be passed to the drawing function, and a value of 0 for the `flags` argument.

The DoUpdateDraw function is defined in the CWindow class as a universal procedure pointer to the static drawing function named sDoUpdateDraw. That function is called automatically by the DeviceLoop toolbox function, for each device on which any portion of the `visRgn` appears. The sDoUpdateDraw function uses the pointer to the window object passed in the `userData` argument to call the window's UpdateDraw function.

The UpdateDraw function draws the contents of the window as follows:

1. The clipping rectangle for the port is set to the rectangle of the current port (`portRect`).

2. If the user's machine has Color QuickDraw installed, then the function calls GetForeColor to access the current port's foreground color and calls RGBForeColor with that color to set the color to the closest match for the current device. The GetBackColor and RGBBackColor functions are called to set the closest background color for the device.

3. Then the update rectangle is calculated to be the bounding box of the update region. This is the rectangle that encompasses all of the elements whose contents need to be updated.

4. The UpdateErase function is called to erase the entire contents of the update region by calling the EraseRect toolbox function with a pointer to the update rectangle, causing it to be filled with the background color.

5. The UpdateDraw function completes its job by testing whether the window has any subviews, and if so, calling the Pane_Draw function of the CPane class for each of these.

6. Upon completion of drawing all of the subviews of the window, the window's Update function regains control and calls the EndUpdate toolbox function to cause the update region to be disposed.

The Pane_Draw function is a global function in the CPane class that is called with a pointer to the CPane-derived object and a pointer to the update rectangle value. The function determines whether any portion of the pane or its border (if any) needs to be drawn and does so as described in Chapter 7, beginning in step 2 on page 302. The important thing to bear in mind is that in the course of implementing the drawing of each pane, the Draw function is called with a pointer to the rectangle to be redrawn.

Drawing a Custom View

Most of the logic that prepares you to draw a subview of your application is handled behind the scenes by the foregoing described functions. Your main job is to override the Draw function in the classes in which you need to perform the drawing operation.

We discussed a custom view in Chapter 4, beginning on page 101, in the section called "Creating a Business Account View." The text in that chapter also describes the custom code needed to draw the cells in the CArrayPane-derived CAcctList class. The Draw function is implemented in the CTable base class, and it determines which of the table's cells need to be redrawn, based upon the update rectangle passed to the Draw function. After determining the cells to be drawn, either its DrawRow or DrawCol function is called (depending upon whether cells are to be drawn a row at a time or a column at a time), and both of those functions call the DrawCell function that we have overridden in our CAcctList class. The custom code for our DrawCell function is shown in Chapter 4, beginning on page 145.

Many of the features of a generic Draw function are shown in the DrawCell code. Some suggestions for implementing your own custom Draw function are as follows:

- Save the current pen state before you make any changes to it. This is accomplished in the DrawCell code by calling the GetPenState toolbox function and passing it a pointer to PenState structure called `savePen` that we defined in the DrawCell function.

- Perform any drawing functions that are needed to update the drawing rectangle passed to the Draw function. In the case of the DrawCell function, we are passed only the cell number and its rectangle. If it is more convenient for you to draw the entire pane for which your Draw function is called (rather than just the portion that needs to be updated), then do so. The clipping region will ensure that none of your drawing will overwrite anything else that covers up an area on which you might draw.

- All standard controls are drawn automatically by the Draw function in the CControl class. That function calls the Draw1Control toolbox function to draw a single control in the active window. It handles drawing both active and inactive controls. If you wish to implement a custom control, such as the CShapeButton control implemented in the VA library, then you can write your own Draw function to do so. Take a look at the Draw function in that class to get ideas of how to draw custom controls.

- Restore the saved pen state before returning from your Draw function. This is accomplished in our DrawCell function by calling the SetPenState toolbox function with the pointer to our `savePen` structure.

Other than the foregoing general suggestions, there is nothing special that you need to observe. The TCL will have taken care of ensuring that the current GrafPort is set properly and that the BeginUpdate function is called before your Draw function is called. The EndUpdate function is called after your drawing is complete. Bear in mind that your Draw function might be called multiple times to draw the same pane if the pane covers more than one display device. So don't save any data in your drawing function, because the data will be saved again, with possibly different values, if the function is called multiple times.

In the case of text panes that are based upon the CEditText class, you need not override the Draw function. The TextEdit Manager will handle all of the drawing that is necessary for those panes.

Printing a Window's Contents

The TCL provides the functionality for printing the contents of the window that is owned by the CDocument-based class. Each CDocument-based object has a member variable called `itsMainPane` that contains a pointer to the primary, or most important, CPane object in the document's window (`itsWindow`). Each CDocument-based object also has a member variable called `itsPrinter` that contains a pointer to a CPrinter object. The CPrinter object is constructed and assigned to the `itsPrinter` variable when the CDocument class constructor executes. The `itsMainPane` variable is assigned in the MakeNewWindow function after the document's window has been constructed.

When the user chooses the **Page Setup** or **Print** commands from the **File** menu, the pointer in the `itsPrinter` variable is used to call the DoPageSetup and DoPrint functions in the CPrinter class. The pointer in the `itsMainPane` variable is taken to be the pane whose contents are to be printed.

Although the TCL assumes that only the window that belongs directly to the CDocument-based class will be printed, it also pro-

vides a method by which other windows' contents can be printed, using their own values for the `itsPrinter` and `itsMainPane` variables. We will discuss this shortly, but for the time being, to cover the basic features of printing support, we will talk only in terms of the two variables in the CDocument-based object.

Printing the Notebook Pane

When we speak of printing, we are usually talking about drawing the contents of a window that appears on the screen onto a printer device. In fact, there is very little difference, in most cases, between drawing and printing. If, for example, you have a simple text window—as is the case in the Notebook application example that we showed in Chapters 5 and 9—printing the contents of the window is accomplished entirely by the TCL, with no additional custom code needed. The TCL handles this case as follows:

1. When the user pulls down the **File** menu and chooses the **Print** command, the command is sent to the DoCommand function of the current gopher (in this case, it's the CEditText pane in the main window), and the command rattles up the chain of command until it reaches the DoCommand function of the CDocument class, where it is handled.

2. The `cmdPrint` case of the DoCommand function in the CDocument class calls the DoPrint function for the object whose pointer is stored in the `itsPrinter` variable (a CPrinter object).

3. The DoPrint function first calls the GetPrintRecord function to access and validate the print record for the current printer. It then calls the OpenPrintMgr function to open the Print Manager and calls the RequestInteraction function of the application object (`gApplication`) to ascertain whether the DoPrint function was called in response to an Apple event. If not, then the DoPrint function calls the PrJobDialog toolbox function with the print record.

4. The PrJobDialog function displays the standard print dialog box to ascertain the starting and ending pages, the number of copies, and other information related to printing the current document. If the user clicks the Cancel button, the PrJobDia-

log function returns FALSE; otherwise, it returns TRUE. This value is stored into the wantsToPrint variable.

5. The DoPrint function continues by calling the ClosePrintMgr function and then testing the value stored in the wantsToPrint variable. If the value is TRUE, then the PrintPageRange function is called with the iFstPage and iLstPage (first and last page) values from the print record.

6. The PrintPageRange function in the CPrinter class is responsible for causing the specified range of pages in the document to be printed. It performs this function as follows:

   a. The GetPrintRecord function is called to get the print record.

   b. The AboutToPrint function in the document object (itsDocument) is called. The AboutToPrint function in the CDocument base class accesses the print record and calculates the page width and page height values. It verifies that the user-specified last page number is larger than the first page number and swaps these two numbers if that relation is not TRUE. It then sets the last page number to the minimum of the specified last page value and the value returned by the PageCount function.

   c. The PageCount function tests whether pagination information has been calculated for this document. If not, then the Paginate function in the CDocument class is called. That function determines the page width and page height once again and then calls the Paginate function in the class pertaining to the document's itsMainPane variable. In the case of our text pane, this is the CEditText pane in the main window. The Paginate function is called with a pointer to the current CPrinter object (aPrinter) and the pageWidth and pageHeight arguments.

   d. The Paginate function for the text pane is inherited from the CAbstractText class. That function tests whether the fixedLineHeights variable is TRUE (as is the case for our text pane). If so, then the lineHeight, linesPerPage, and pageHeight values are calculated. Then the Paginate function of the CPanorama class is called

with the pointer to the printer object (`aPrinter`) and the `pageWidth` and `pageHeight` values.

e. The Paginate function in the CPanorama class first accesses the width and height of the panorama, in pixels, and then computes the number of horizontal strips that will be printed by taking the total height in pixels and dividing that by the page height. In a similar fashion, the number of vertical strips is computed by taking the total width of the panorama in pixels and dividing that by the page width. Then the SetStrips function for the CPrinter object is called to record the number of horizontal and vertical strips that make up the entire panorama. In addition, the SetAllStripWidths and SetAllStripHeights functions of the CPrinter object are called with the page height and page width values, so that all of the printed strips will be of the same height and width.

f. When execution resumes in the PrintPageRange function of the CPrinter class, the pagination of the text pane is complete and the OpenPrintMgr function is called prior to commencing the print operation. Immediately after that, a printer port (`macPrintPort`) is assigned by calling the PrOpenDoc toolbox function with the print record. The `printDocOpen` variable is also set to `TRUE`.

g. The main body of the print operation is then handled in a loop in the PrintPageRange function. The function loops through the set of pages, from the first page to last page, sets the `printPageOpen` variable to `TRUE`, and calls the PrOpenPage toolbox function with the printer port. If no error has yet been reported, it calls the PrintPageOfDoc function in the document object (`itsDocument`), sets the `printPageOpen` variable to `FALSE`, and calls the PrClosePage toolbox function. The foregoing actions are executed repeatedly until either the PrError function returns a nonzero result or the specified number of pages have been printed.

h. After printing is complete, the PrintPageRange function sets the `printDocOpen` variable to `FALSE`, calls the PrCloseDoc toolbox function, sets the `macPrintPort` printer port variable to `NULL`, determines whether the

printout was spooled to an external file by testing the print record's `prJob.bJDocLoop` field, calls the PrPicFile toolbox function to initiate hard copy printout of the spooled output, calls the ClosePrintMgr function to terminate the print operation, and, finally, calls the DonePrinting function of the document object (`itsDocument`), which, in turn, calls the DonePrinting function of the text pane (`itsMainPane`).

7. When the DoCommand function of the CDocument class regains control, the printout will have been accomplished. At this point, the function tests whether the application has been factored. If so, it calls the SendPrint function, which sends an Apple event to its own application, specifying that a Print operation has been requested, but also telling the application not to execute the operation. This is for the benefit of any other application that is recording the user's events, so that the print operation can be "played back" at a later time.

What we haven't yet covered in the foregoing steps is how each page of the document is printed. In the inner loop, we indicated that the PrintPageOfDoc function is called to print each page of the document. That function calls the PrintPage function of the main pane object (`itsMainPane`), which, in this case, is the CEditText object.

The PrintPage function of the CEditText class substitutes the printer port for the port in the Text Edit Record (TERec) and then tests whether the output is to be clipped to the page boundary (`clipPAGE`). If so, then the view rectangle is resized to the width and height of the printer's page and the PrintPage function of the CAbstractText base class is called.

The PrintPage function is inherited by the CAbstractText class from the CPanorama class. That function is called with the page number, page height, page width, and a pointer to the current CPrinter object. The function prints each page as follows:

1. The GetPageArea function of the CPrinter object is called with the page number and a pointer to where the rectangular coordinates corresponding to that page are to be stored.

2. After the coordinates for the page within the panorama have been calculated, the ScrollTo function is called to position the

top-left corner of the pane to be printed at the top-left corner of the panorama.

3. The page area's coordinates are then converted to window coordinates and are stored into both the `cPageArea` CPane-class variable and the local `qdArea` variable. Then the DrawAll function is called to draw the specified area into the current port (which is now a printer port). The DrawAll function was described in Chapter 7, in the section titled "Handling Update Events," beginning on page 301. The core operation in the DrawAll function is to call the Draw function for the pane—just as the Draw function is called to draw the pane when it is being displayed on the screen.

Other than the functions that execute before and after the drawing operation, there is little difference between drawing to the screen and drawing to the printer, at least as far as your program is concerned.

Printing an Offset Pane

In some cases, you will want to design a window in which the pane to be printed is offset from the top and/or left edge of the window. This is illustrated in Figure 12-1.

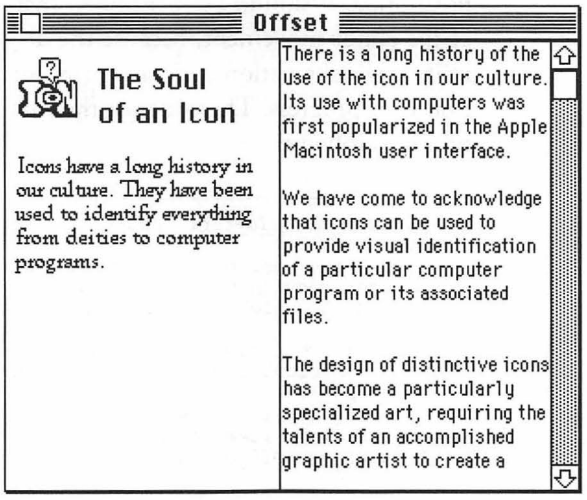

Figure 12-1
An offset scrolling text pane view

The foregoing figure shows a view called Offset that has a picture of a particular item in its upper-left corner, an item title, a brief

description of the icon, and then a scrolling text field that is intended to contain a more comprehensive description of the item. The programmer's intention here is to allow the user to print the contents of the scrolling text field, which is assigned to the `itsMainPane` variable in the program.

If we do nothing, the text will be printed offset from the left edge of the printed page by the amount that the scrolling text field is offset from the left edge of the window. This is usually not what is wanted.

If you recall from our discussion of the PrintPage function on page 457, I mentioned in step 2 that the ScrollTo function was called to scroll the pane to be printed to the top-left corner of the panorama. In this case, the CEditText (scrolling text) field is the panorama, and so the ScrollTo function merely positions the text with its first line at the top-left corner of the scroll pane.

In order to print the text, beginning at the top-left corner of the printed page, we need to move the entire pane so that it is physically positioned at the top-left corner of the window. To do this, we need to override the AboutToPrint function in our CMain (CDocument-derived) class, move the CEditText panorama to the window's corner, print the pane, and then override the DonePrinting function to move the pane back to where it was originally positioned. It should be mentioned that the user will be unaware of the pane's movement, because the display port will still show it in its original position while the printout (or print spooling) function is in progress. The code for the AboutToPrint override function is as follows:

```
void CMain::AboutToPrint (short *firstPage, short *lastPage)
{
   // call the base class function first,
   // before moving the current pane.
   x_CMain::AboutToPrint (firstPage, lastPage);

   // then save the pane's current horizontal
   // and vertical origin values and move
   // the pane.
   saveHOrigin = itsMainPane->hOrigin;
   saveVOrigin = itsMainPane->vOrigin;
   itsMainPane->Offset (hOrigin, vOrigin, FALSE);
}
```

The `saveHOrigin` and `saveVOrigin` variables are defined as member variables in our **CMain.h** header file. They will hold the

original value of the horizontal and vertical components of the scrolling text pane (`itsMainPane`). The Offset function moves the pane by subtracting the specified offset values from the original `hOrigin` and `vOrigin` values. It also adjusts the origin of the pane's enclosure (the scroll pane) by the same amount in each direction.

At this point, the pane is positioned correctly to be drawn at the top-left edge of the printed output. When the printout is complete, we override the DonePrinting function to reverse the offset procedure, as follows:

```
void CMain::DonePrinting ()
{
   // call the base class function and then
   // move the pane back where it belongs.
   x_CMain::DonePrinting();
   Offset (-saveHOrigin, -saveVOrigin, TRUE);
}
```

Note in the foregoing that we have negated the values of the saved origin values, when the Offset function negates these, it will move the pane (and its enclosure) in a positive direction, back to its original position.

Printing a Secondary Window's Contents

We have been talking about printing the contents of a particular pane in the main window up to this point. Our description of the methodology of printing a window's main pane should make it clear that the document's CPrinter object (`itsPrinter`) and the main pane (`itsMainPane`) are securely bolted into the printing code. Fortunately, it is fairly simple to use the same code to print the contents of other windows by overriding the DoCommand function and handling the `cmdPageSetup` and `cmdPrint` commands in the window's director. The code to perform this task, assuming that the Offset window shown in Figure 12-1 is a secondary window for the document, is as follows:

```
void COffset::DoCommand (long theCommand)
{
   switch (theCommand)
   {
      case cmdPageSetup:
      case cmdPrint:
      {
```

```
            try_
            {
               ((CMain *)itsSupervisor)->AboutToPrintSubDirector(
                     itsMainPane, itsPrinter);
               x_COffset::DoCommand (theCommand);
               ((CMain *)itsSupervisor)->DonePrintingSubDirector();
            }
            catch_all_()
            {
               ((CMain *)itsSupervisor)->DonePrintingSubDirector();
               throw_same_();
            }
            end_try_
            break;
         }
         default:
         {
            x_COffset::DoCommand (theCommand);
         }
      }
   }
```

The foregoing code uses the AboutToPrintSubDirector and DonePrintingSubDirector functions inherited from the CDocument class to save and restore the `itsPrinter` and `itsMainPane` variables. The document's copies of those variables are saved into the `savePrinter` and `savePrintPane` variables when the AboutToPrintSubDirector is called, replacing the document's variables with the specified values. When the DonePrintingSubDirector function is called, the original values of the `itsPrinter` and `itsMainPane` are restored. The code is executed in a `try` block because it is possible for a failure condition to occur, and we want to make sure that the original values of the document's variables are restored in that case.

Drawing and Printing Summary

This chapter described the methodology of the TCL in implementing drawing and printing operations. In general, printing is handled by the same Draw function that handles drawing to the screen. There are a few more complexities to the printing process, however, and these are described in terms of the additional actions taken by the TCL to implement the appropriate behavior.

Printing a pane whose left and/or top edges don't coincide with the corresponding edges of the enclosing window is handled by shifting the pane prior to printing, and then shifting it back into its original position after printing is complete.

Printing of the contents of multiple windows requires that the AboutToPrintSubDirector class to be used to save the main document's `itsMainPane` and `itsPrinter` pointers, substitute pointers to the corresponding objects for the window to be printed, and then restore the original main document's pointers.

Index

Note: Page numbers in *italics* refer to illustrations.

C

E

T

U

V

W

X

Z

MACINTOSH PROGRAMMING

The THINK Class Library (TCL) is a comprehensive application framework, common to Symantec C++™ compiler products for both the 68000 and PowerPC. ***Mastering the THINK Class Library*** provides a thorough examination of the TCL and the Visual Architect,™ a graphic user interface development tool that allows you to produce commercial-quality applications with a minimum of effort.

The author describes fully the structure and operation of the TCL, including explanations of all code generated by the Visual Architect, any necessary custom code, and the operation of this code. Visual Architect tutorials provide you with a step-by-step approach for simplifying the development of complex Macintosh applications using the TCL. You'll also learn:

- how to use the Visual Architect to construct a user interface and custom application features
- tips for constructing and implementing many different types of views
- inside information on modal, modeless, and dynamic dialogs
- how event processing works and how the TCL performs most of the work for you
- how to read from and write to external data files
- the inside scoop on how Apple events work inside the TCL
- how Object I/O works and how to use it in your own applications
- printing secrets, including how to print multiple windows and offset panes.

Mastering the THINK Class Library is an essential resource for every Macintosh developer using TCL and Visual Architect.

RICHARD O. PARKER is an established expert in Macintosh programming and Symantec's software tools, and is the author of several Mac programming books. His career spans over thirty years in the computer industry, and his experience includes everything from the development of operating systems and compilers to the design of microprocessor CPU devices and software.

Cover design by David High
Cover photographs by Steffany Rubin

ISBN 0-201-48356-4

$29.95 US
$41.00 CANADA

Addison-Wesley Publishing Company