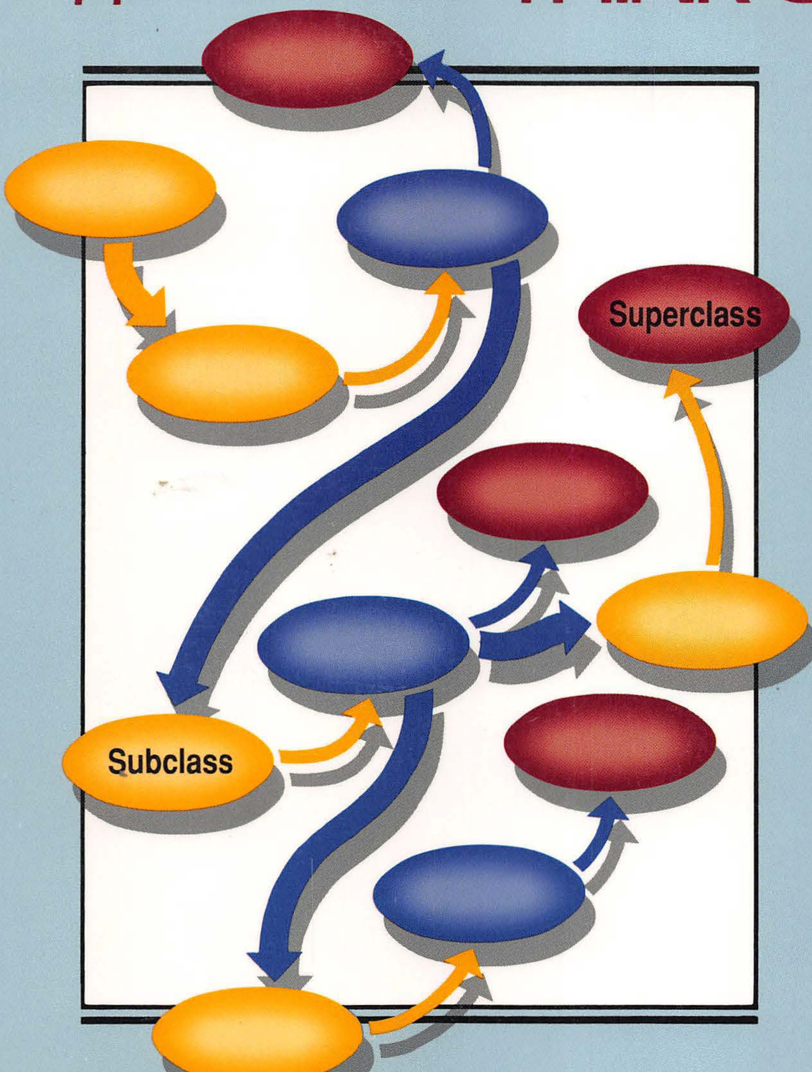
DISK INCLUDED

Easy Object Programming

for the Macintosh Using AppMaker™ and THINK C™



Richard O. Parker

Easy Object Programming For the Macintosh Using AppMaker™ and THINK C™

Richard O. Parker

PRENTICE HALL, Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

Parker, Richard O.
Easy object programming for the Macintosh using AppMaker and THINK C / Richard O. Parker.
p. cm.
Includes index.
ISBN 0-13-092966-2
1. Macintosh (Computer)--Programming. 2. Object-oriented programming (Computer science) 3. AppMaker (Computer file) 4. THINK C (Computer file) I. Title.
QA76.8.M3P35 1993
005.265--dc20 92-38625
CIP

Publisher: Alan Apt
Production Editor: Bayani Mendoza de Leon
Copy Editor: Brian Baker
Cover Designer: Bruce Kenselaar
Prepress Buyer: Linda Behrens
Manufacturing Buyer: Dave Dickey
Editorial Assistant: Shirley McGuire

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-092966-2

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada, Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd, *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

This book is dedicated to my mother. Throughout its creation she constantly encouraged me to keep on writing, even when I was fighting bugs in the code or suffering for lack of the right words to describe the development process.

She is a remarkable woman, a gifted artist in her own right and one who has lived from before the dawn of the 20th century to see and experience all the new technology leading to what this book describes. May God grant her the gift of seeing the dawn of the 21st century as well.

Contents

Chapter 3

Chapter 4

Chapter 11

Preface

This book is about object-oriented programming in C. But, more than that, it stresses the ease with which object-oriented programs can be developed with the aid of an excellent development environment, an extremely robust class library, and a powerful user interface design and code-generation tool.

The book describes the evolution of a complete, multipurpose application, starting from a skeleton application, automatically generated by AppMaker. The user interface of the skeleton application is enhanced within AppMaker to create a single Edit-Text window, in which text can be written in any font, style, or justification. The generated code is enhanced to provide the capability of changing the selected text style, size, and justification, using a custom-designed dialog box for making the selections. Custom code is also provided to write the text to a file and have the ability to open the file at a later date, revise the text, and save the file with the same or a different name. The book describes all of the custom additions to the code, in a manner that shows how the application can gradually evolve from a mere skeleton to a full-fledged Macintosh application.

In subsequent chapters, a spreadsheet window is designed, the generated code for this new addition to the application is described, and the custom code to make it fully functional is covered in full. This is the third stage of evolution for the application. A dialog for changing the characteristics of the spreadsheet window is then designed, implemented, and described.

The penultimate addition to the application is a drawing window, in which graphs depicting patterns in the spreadsheet data are prepared. The user interface design, the generated code, and the customizing needed to fully implement the graphing addition's functionality are discussed.

Finally, chapters that implement and describe the printing of the various windows' contents and a tutorial for creating a stand-alone application are presented. As a whole, the application is called Ensemble, to indicate that it embodies a combination of complementary modules that work together to provide a notebook, worksheet, and graphing facility which would make a good addition to any user's repertoire.

More than anything else, the book strives to show that complex Macintosh applications can be developed quite easily, in an evolutionary manner, by using the right tools and by applying them in a step-by-step fashion. Because of the object-oriented approach of the book, a great number of features in Symantec's THINK Class Library are presented. These illustrate the power of a comprehensive class library that works behind the scenes to minimize the amount of complex code that the programmer is required to develop.

Few, if any, Macintosh programming books cover the evolution of an entire application; most merely focus on the use of individual programming techniques. This book attempts to show how a *real application* can be simply and easily developed, step by step. The presentation is punctuated with data flow diagrams that illustrate the dynamic structure of the application at various stages of its development. There are tutorials on how to use AppMaker to produce the various user interface elements for the windows, dialog boxes, and menus employed in the application. The book contains a detailed examination of the code generated by AppMaker for each new user interface feature, as well as the manually added custom code to make each new feature fully functional. The application is complete and fully operational at each stage of its development.

Not only is the application whole and complete, but it is nontrivial. It makes use of features of the Macintosh toolbox, as well as the THINK Class Library, that would be difficult to present outside the context of a complete application. The Ensemble application incorporates quite a few programming principles that reinforce a useful structure for object-oriented applications in general. These principles can be applied over and over again, especially if the programmer is using AppMaker and the THINK C programming tools.

The enclosed disk contains folders which include the source code, THINK C project files, and AppMaker resource files for six versions of the Ensemble application. These versions represent six distinct phases in the application's evolution and correspond directly to the chapters in the book associated with each folder's name. An executable version of the final Ensemble application, along with its corresponding data file is also included on the disk.

It should be possible to open this book at any one of its chapters and refer to the interface design or customizing descriptions without having to reread the entire book. Each major user interface feature is described by a triad of chapters. The first chapter describes how the feature is designed within the AppMaker environment. The second chapter discusses the code that is generated to implement the default behavior of the feature, and the third chapter describes the custom code that was added to make the new feature fully functional.

I used two mainstream development tools to create the application described in this book: AppMaker version 1.5, created by Bowers Development Corporation, and THINK C version 5.0, created by Symantec Corporation.

AppMaker is a resource editor and code-generation application that allows the programmer to create complex user interface elements with a visual paradigm. Its WYSIWYG (what you see is what you get) tools allow windows, dialog boxes, menus, and alerts to be designed. It also includes a balloon help editor and comprehensive text styling for all of the user interface elements. Through the use of AppMaker, your windows and dialogs can contain all of the standard Macintosh user interface elements, including checkboxes, radio buttons, lists, buttons, drawing panes, borders, gray lines, PICT images, ICONs, and other elements. Once a user interface element is designed, AppMaker will generate code in any one of a variety of popular languages and dialects, including THINK C, THINK Pascal, MPW C, MPW Pascal, or C++. For each of these languages, the generated code can be procedural or object oriented, as desired.

THINK C is an ANSI-compliant C language compiler, with object programming extensions that are a compatible subset of those found in the C++ language. The object features of the language are supplemented by a comprehensive class library called the THINK Class Library (TCL). All of the code in this book is written with the underlying functionality provided by AppMaker and TCL classes. In addition, THINK C is a marvelously efficient development environment, where editing, compilation, and debugging are accomplished with relative ease and speed. These days, when object-oriented programming is de rigueur for most new applications, there are very few books that show how entire applications are structured. This book attempts to fill that gap and show how a complex application can be easily created, in a step-by-step manner, by using the proper tools.

It is my fervent hope that programmers reading the book will be left with an increased understanding of how to approach the design of a complex application by using the suggested tools. I also hope that they will have a greater appreciation of the structure of Macintosh applications and will be better prepared to begin programming in the object-oriented way.

Acknowledgments

This book is the result of the efforts of many people. I am very grateful to each of them for helping to make the publication of this book a reality. I would especially like to thank Carole McClendon, my agent, for helping me understand the complexities of technical book publishing. I would also like to thank Alan Apt, my publisher, for putting up with my barrage of EMAIL messages and for being truly supportive throughout this effort. Thanks also go to Bayani de Leon, my production editor, for his help in creating the camera ready copy for the book. Finally, I would like to add my special thanks to the reviewers of the book. Kurt Schmucker, Apple Computer, offered a great number of suggestions for improving the technical quality of the text and figures, and Spec Bower, Bowers Development, performed a comprehensive review of the technical content of the tutorials and all of the program code. I am very grateful to both of these people for their unselfish contributions.

Richard O. Parker

Chapter 1

Introducing the Tools

This chapter describes the tools that were used to construct the application that is developed in this book. In addition, it contains a tutorial that will allow you to get started using the tools to develop the framework for the sample application that makes up the body of this book.

The fundamental software tools are AppMaker version 1.5 and THINK C version 5.0, although we will also be using Apple's ResEdit program in some instances. In addition, the book will sometimes refer to Apple's six-volume set of *Inside Macintosh* manuals, which contains full documentation of the toolbox routines that provide the Macintosh operating system with its amazing capabilities. You may wish to refer to other important books concerning Macintosh programming. The four volume set titled *Macintosh Revealed*, by Stephen Chernicoff (Hayden Books), and the two-volume set titled *Macintosh Programming Primer* are particularly good. The first volume of the *Macintosh Programming Primer* was written by Dave Mark and Cartwright Reed. The second volume was written solely by Dave Mark. (Both are published by Addison-Wesley.)

The Chernicoff books are written for use by Pascal programmers, but because it is quite easy to translate between Pascal and C, this should not be a deterrent to C programmers wanting to know some of the inside secrets of programming the Macintosh. The Mark and Reed book and Dave Mark's second volume of that series are devoted to programming in C, especially THINK C.

This book departs from those others by illustrating object-oriented programming techniques at the outset. Object-oriented programming is becoming such an essential part of all software development—on a variety of platforms—that I feel that it is important to begin to demystify the whole topic and teach new and experienced programmers alike about the principles of object-oriented programming for the Macintosh.

As with any other endeavor, having the right tools for the job not only makes the job easier to accomplish, but can even turn what seems an impossible chore into something that is entirely feasible, as well as enjoyable, to accomplish.

When the Macintosh was first introduced, it provided a feature within its file structure that was entirely revolutionary. Macintosh files had resource forks, which contained descriptions of the user interface elements used within a given application. To modify the position of a window, the wording of a menu item, or the name of a push button, all you had to do was edit the appropriate resource, and the change was accomplished, with no need to recompile the program. In fact, many early users of the Macintosh became quite adept at customizing their favorite programs, and even the operating system itself, with no access whatsoever to the source code.

The tool of choice in the early days was ResEdit, which is still a viable resource-editing tool that has been kept up to date by Apple with the addition of editor modules for all the latest resource types. ResEdit requires quite a bit of technical knowledge about the various resources it creates and edits, so it is often shunned by beginning Macintosh programmers, who are intimidated by its potential to wreak havoc in their systems. In fact, almost every tutorial on the use of ResEdit hastens to caution the user about its potential dangers, and always includes the admonition to work on a *copy* of the file to be edited. Nonetheless, ResEdit continues to be a handy utility, especially when custom resources need to be created and the user is careful in its use. ResEdit will be used both to modify and to create new resources in this book.

For the applications described in the book, AppMaker will be used almost exclusively. While AppMaker is able to edit the resources in existing applications, its greatest asset is its ability to create the needed user interface resources for new applications by using its onscreen WYSIWYG tools and then generating the code to operate the interface.

In fact, AppMaker generates a complete application program skeleton that includes all the elements which allow the application's user interface, once compiled, to be exercised. Commands can be selected from menus, buttons can be clicked, and dialog boxes can even be opened by making an appropriate menu selection. Visual proof that the interface is operable is provided by the standard highlighting of selected menu commands, check marks appearing and disappearing in checkboxes, and single-selection radio buttons that operate within the group in which they have been defined. The interface is truly operational.

Compilation of an AppMaker-generated application is easily accomplished by using the Starter project for the THINK Class Library (supplied in the AppMaker product), adding the generated source files to the project, and then telling the compiler to bring the project up to date. This process consists of compiling not only the generated files, but also all the files that form the THINK C Class Library (TCL). Because almost all of the TCL is added to each project, and because THINK C keeps the object code inside the project file, it is not unusual for a THINK C project file to be several megabytes. Do not fear that your compiled program will be that large, because the THINK C linker will only include the files your program actually needs to execute properly.

Once the entire project has been compiled for the first time (which might take quite a while, depending on the speed of your particular Macintosh model), future compilations will be limited to only the files that have changed (and others that depend on these files) since the last compilation. In this respect, THINK C is a very efficient environment for developing new applications.

Creating a New Resource File with AppMaker

AppMaker is both a resource editor and a code generator. The outputs from AppMaker are a resource file and (optionally) a set of source files for the selected language. In this book, AppMaker will be used to create and enhance the resource file for our application and also to create source files for compilation by THINK C.

Complete instructions for using AppMaker are contained in the product's manual; however, it is useful to repeat the instructions for creating a new resource file at this point. I'm assuming that you are sitting in front of your Macintosh and are getting ready to launch the AppMaker application at this time. It will be convenient for you to do so as you continue to read this tutorial. Following are the steps for creating a new resource file for your object-oriented programming project:

1. First, create a new folder on the disk where you want your THINK C project and its source files to be stored. Name the folder **Ensemble**.

2. Navigate back to the folder in which the AppMaker application resides and launch AppMaker version 1.5.

3. You will see an open file dialog box for the folder in which AppMaker is located, as shown in Figure 1-1.

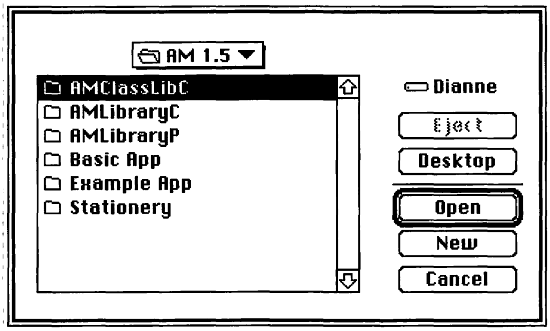


Figure 1-1
AppMaker's open file dialog box

4. Navigate to the **Ensemble** folder, and click on the New button, as shown in Figure 1-2.

5. When the **New** button is clicked, AppMaker will display the dialog box shown in Figure 1-3. You should name the new resource file **Ensemble.π.rsrc**, as shown. This is because THINK C project files are typically named with a file extension of '.π', and our THINK C project file will be named **Ensemble.π** when we get to that point. THINK C will always look for a resource file whose name exactly matches that of the project, with the further file extension of '.rsrc'. Click the **Save** button.

Figure 1-2
Creating a new AppMaker resource

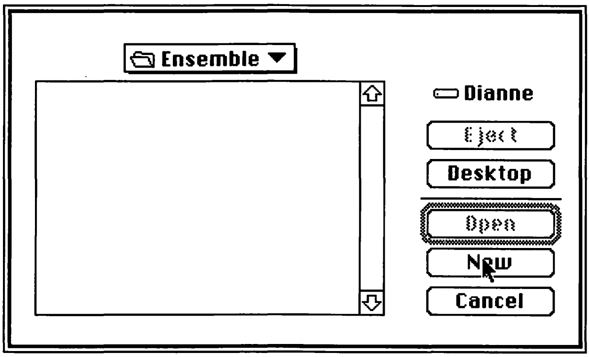
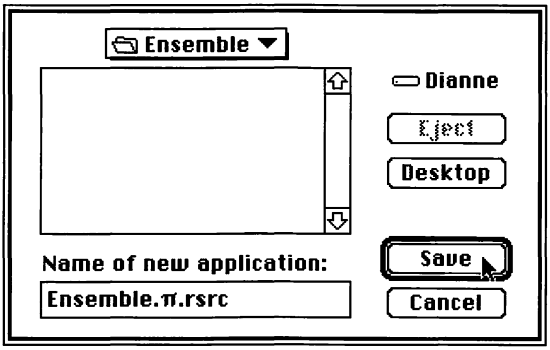


Figure 1-3
Naming the new resource file



6. After the **Ensemble.π.rsrc** file has been created, AppMaker will add a number of resources that THINK C and the TCL require. These are shown in Figure 1-4, which is a screen dump of the contents of the file's resource fork, as shown by the ResEdit utility. As you can see, quite a number of resources are automatically written into the **Ensemble.π.rsrc** file's resource fork. These constitute the minimum set needed to support our AppMaker-generated application, which automatically includes a menu bar, **Apple**, **File**, and **Edit** menus, and a default Window definition. The other resources are needed by the TCL and AppMaker's default generated code.

7. Look at AppMaker's working area screen, which is depicted in Figure 1-5. Notice that in addition to the standard **Apple**, **File**, and **Edit** menus, Appmaker adds an active **Select** menu and inactive **View**, **Tools**, and **Options** menus. In addition, there is a window on the right portion of the screen that contains a list of items in

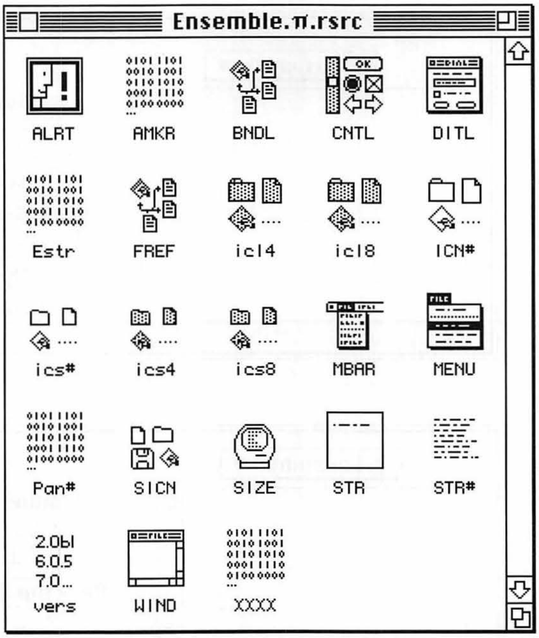


Figure 1-4
Initial resources written into **Ensemble.π.rsrc** by AppMaker

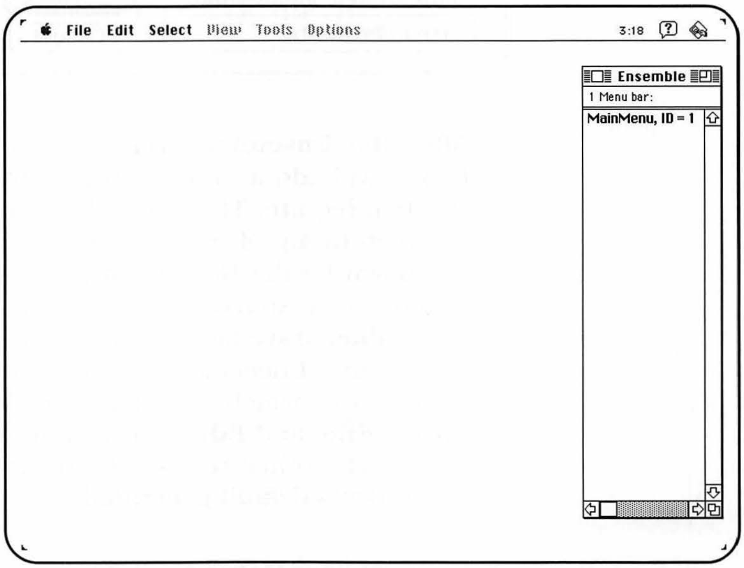


Figure 1-5
AppMaker's working area screen

the current selection category. By default, AppMaker selects the available menus and lists the **MainMenu** in this window. Other resources can be selected by pulling down the **Select** menu and choosing one of the other items.

8. Keep AppMaker's default selection of **Menus**, and double-click on the **MainMenu** item in the current selection window. AppMaker will display the default menu bar that will automatically be included in your application, as shown in Figure 1-6.

Figure 1-6
AppMaker's default application menu bar

9. Click on the **Apple** symbol in the menu bar, and AppMaker will drop down the Apple menu, whose commands will be included in your application. The menu is shown in Figure 1-7. Notice that it only includes an *About* item and a gray line. The names of the current desk accessories (or Apple Menu Items in System 7) will be filled in at run time, when the application is started. The '#256'

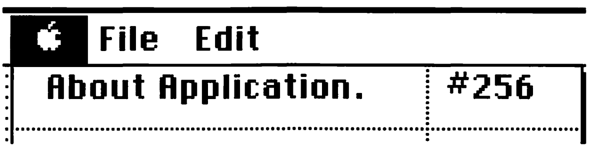


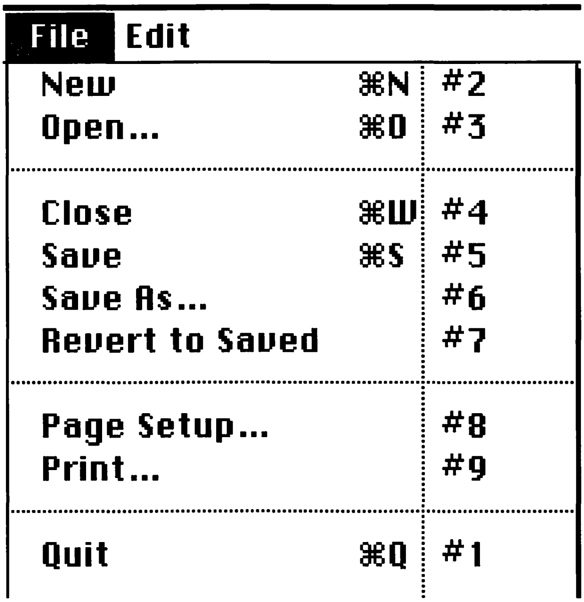
Figure 1-7
AppMaker's default **Apple** menu

appearing in the menu entry is the *command number* required by the TCL for dispatching the selection of that menu command.

10. Now, click on the word **File**, and the default **File** menu will drop down. AppMaker inserts the appropriate **File** menu commands, as required by the TCL, into the default menu, pictured in Figure 1-8. Notice that each of the **File** menu's commands has a *command number*, as is required by the TCL for command dispatching.

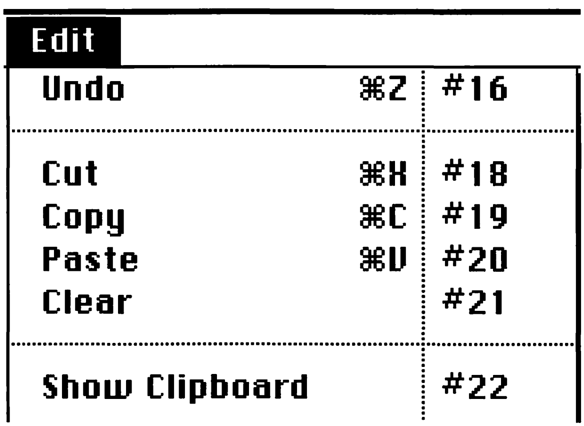
11. Finally, click on the word **Edit** to see the default **Edit** menu provided by AppMaker. This menu contains all the standard commands for cutting and pasting and also the command to show the contents of the automatically gener-

Figure 1-8
AppMaker's default **File** menu commands



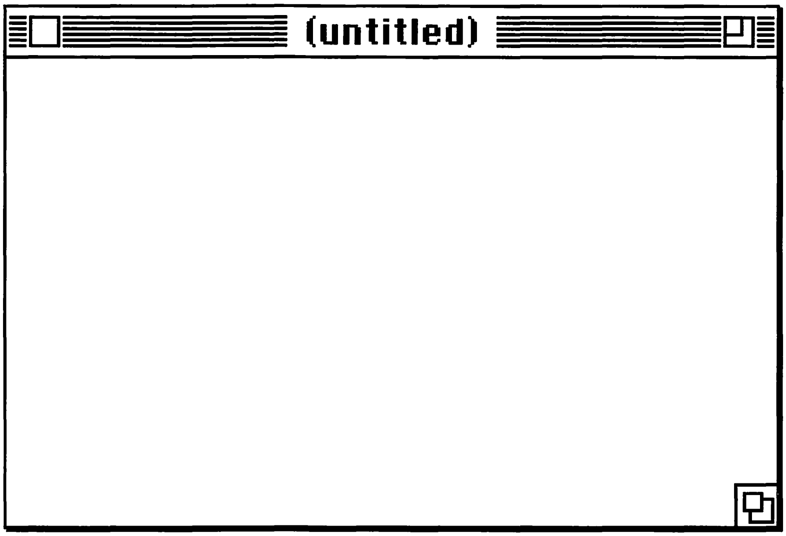
ated **Clipboard**. The default **Edit** menu is shown in Figure 1-9. Once again, each of the **Edit** menu's commands has been assigned a standard *command number*, which corresponds to the definitions in the TCL for these commands.

Figure 1-9
AppMaker's default **Edit** menu commands



12. Now that you have seen the default menus that are automatically generated by AppMaker for your application, click on the **Select** menu and choose the **Windows** item. You will see two standard window items in the current selection window. One is the **Clipboard**, and the other is a standard window called **MainWindow**. Double-click on the **MainWindow** item to see its appearance on your screen, as shown in Figure 1-10. The default **MainWindow** shown in the Figure has a close box, a zoom box, and also a size box. It is initially created as an **(untitled)** window; however, these characteristics can be changed, as with any of AppMaker's generated resources. For now, leave the window definition alone.

Figure 1-10
AppMaker's default **MainWindow** definition



13. If you decide to do so, you can choose the **Dialogs** command from the **Select** menu and see that no dialogs are listed. Selecting the **Alerts** command results in the display of quite a few standard Alerts in the current selection window. Figure 1-11 shows a list of the default ALRT resources defined by AppMaker. You can look at any of these **Alerts** by double-clicking on any of the entries in the current selection window.

14. After you have examined the **Alerts**, you should request that AppMaker generate the code that implements your application's skeleton, using the default set of window, menu, and alert resources that it has generated. To do that, choose **Generate** from AppMaker's **File** menu, as shown in Figure 1-12. When this command is selected, the dialog box shown in Figure 1-13 is displayed. This

Figure 1-11
AppMaker's default ALRT resources

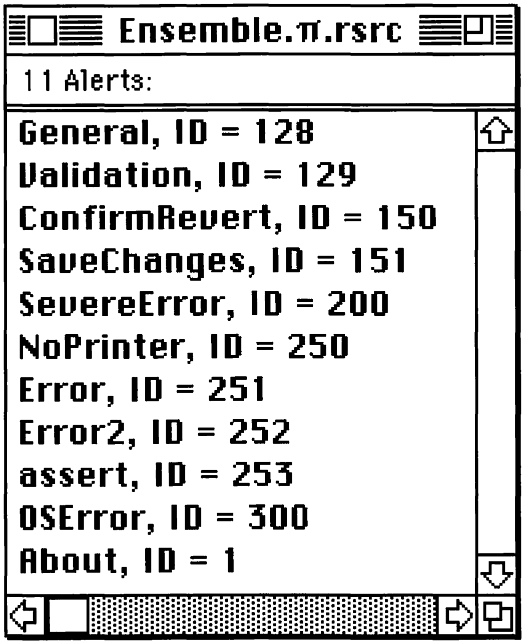
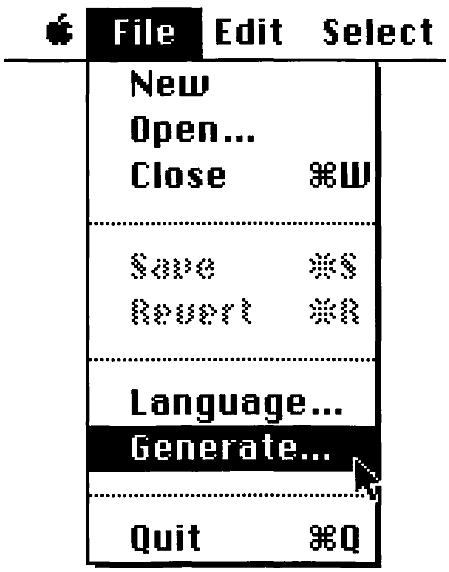


Figure 1-12
Choosing the **Generate** command from the **File** menu



dialog lists all the modules that AppMaker has determined are needed to implement the current user interface design, which in this case includes only the default resources described in the preceeding steps.

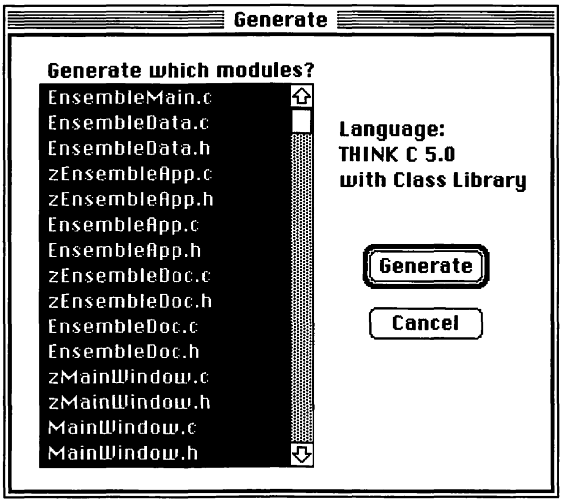


Figure 1-13
AppMaker's **Generate** dialog

15. Once all the modules have been generated, you can choose the **Quit** command in AppMaker's **File** menu. Be sure to click **Yes** when AppMaker displays the dialog box shown in Figure 1-14.

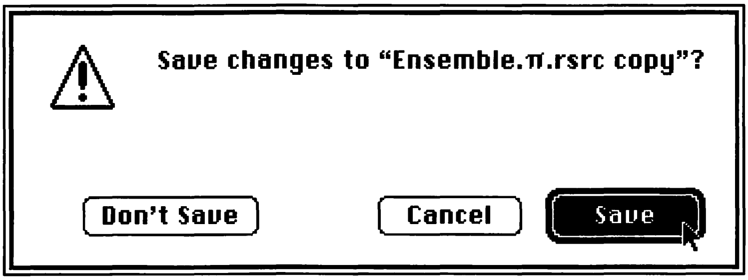


Figure 1-14
AppMaker's save changes dialog

At this point, you have created a complete set of source files that, when compiled along with the THINK Class Library routines, will implement a working application. The next section will discuss how to set up a THINK C project file for this default application and how to add your source files to it. We will also discuss compiling the resulting set of files and running the application.

Perhaps you noticed in the previous set of steps that we allowed AppMaker to use its default set of resources for our initial application. This is an important point. In most cases, the default resources can be used as a starting point for applications you will develop. The next chapter will begin a discussion of the structure of the Ensemble application and the relationship between the classes from which it is composed.

This might be a good time to take a break and review the operations involved in creating a set of default resources and the source files that implement their functionality. Almost every THINK C application that uses the TCL will be built in the same way.

Creating the Think C Project

This section describes how to set up a THINK C version 5.0 project file that will contain all the necessary TCL source files, as well as those generated by AppMaker for our **Ensemble** example.

1. The first step is to make sure that AppMaker's **AMClassLibC** folder is inside the **THINK C 5.0 Folder** on your development disk. This will ensure that the additional classes provided with AppMaker will be available to your projects.

2. Inside the **AMClassLibC** folder is a file called **Starter.π**, which should be duplicated and moved into the folder called **Ensemble** that you created in the previous section. The **Ensemble** folder holds your new AppMaker resource file and the generated source code files.

3. Rename the **Starter.π** file **Ensemble.π** (the 'π' symbol is created by holding down the Option key and pressing the 'p' key). Your set of files should contain those shown in Figure 1-15, which is a small icon view in the Finder.

4. Double-click on the **Ensemble.π** file to launch THINK C version 5.0. You will notice that a great number of files have already been added to the project window, as shown in Figure 1-16. Notice that only a small fraction of the number of files is shown. You can scroll through the files using the scroll bar in the **Ensemble.π** project window.

Figure 1-15
List of **Ensemble** project files in Finder

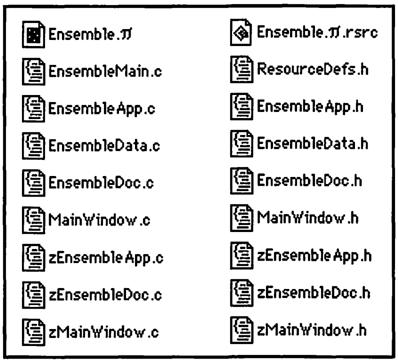
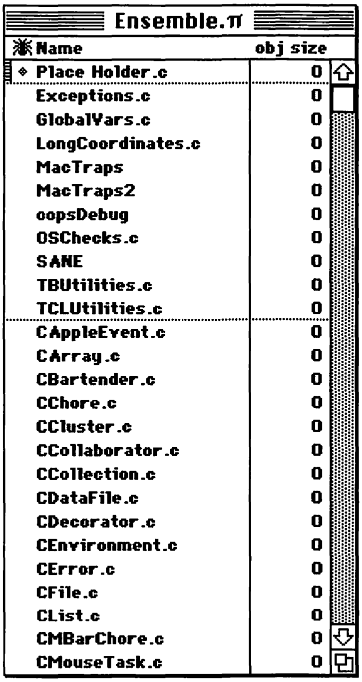


Figure 1-16
THINK C's Ensemble project files



You will also see that the files have already been grouped into segments (separated by gray lines) that are of an appropriate size. If you scroll back to the beginning of the list, you will see a file in the first segment called **Place Holder.c**. This file is in the first segment merely to act as a placeholder for your project's files. It serves no other purpose.

5. Click on the **Place Holder.c** file to highlight it. This selects the first segment for the next operation. Once you have done that, pull down the **Source** menu and select the **Add** command, as shown in Figure 1-17.

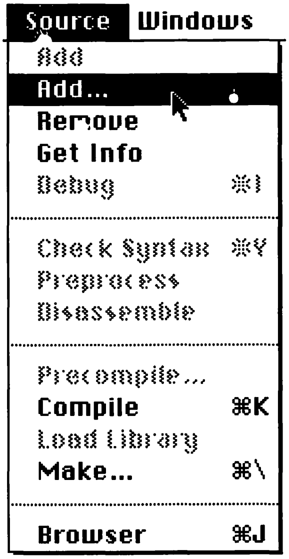


Figure 1-17
Selecting THINK C's Add command

6. When the **Add** command is selected, you will see the dialog box shown in Figure 1-18. This dialog box has two sections. Make sure that you navigate to the **Ensemble** folder using the pop-up menu at the top of the dialog box if it doesn't already indicate that folder's name. Click the **Add All** button, as shown in the figure.

7. After you have added all the files shown in the upper portion of the dialog box, you'll notice that the box is empty, and all the file names have moved to the lower portion, as shown in Figure 1-19. Click **Done**, as shown.

8. If you look in the **Ensemble.π** project window, you will see that all the C language source files have been added to the project, in alphabetical order, as shown in Figure 1-20. In the next step, the **Place Holder.c** file will be removed.

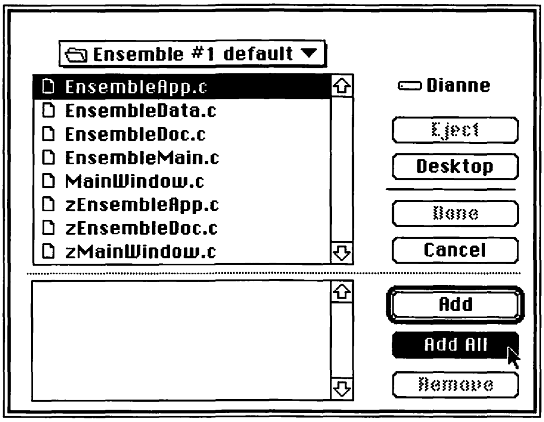


Figure 1-18
Adding all the Ensemble files

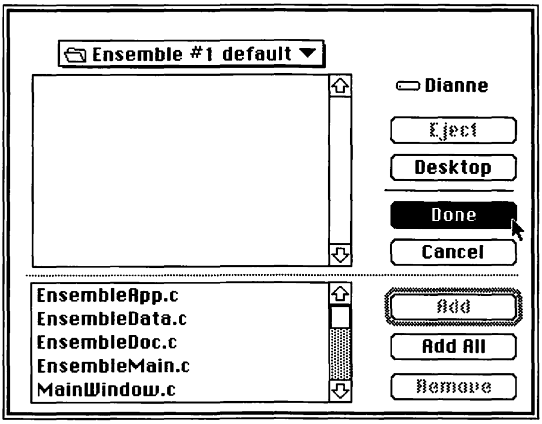


Figure 1-19
Clicking **Done** in the **Add** dialog box

9. Click on the **Place Holder.c** file to highlight it, and then pull down the **Source** menu and select the **Remove** command, as shown in Figure 1-21.

10. At this point, you are ready to compile all the files in the project. Select the **Bring Up To Date** command from THINK C's **Project** menu, as shown in Figure 1-22. This will cause THINK C to begin compiling all the source files.

11. When the compilation is complete, select **Run** from the **Project** menu, and THINK C will display its debugger windows, with the execution cursor positioned at the beginning of your application's **main** function, as shown in Figure 1-23. The figure also shows that the **Go** button in

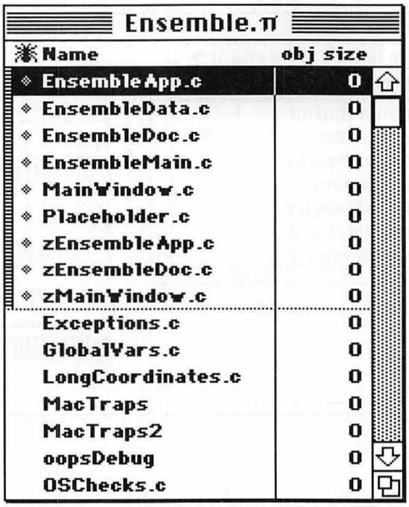


Figure 1-20
All Ensemble files have been entered into the project

Figure 1-21
Removing the **Place Holder.c** file

the debugging window is about to be pressed. Doing so will cause the application to begin execution.

12. When the application begins execution, it will display the menus and default window that were created by AppMaker, as described in the previous section. The display will be similar to that shown in Figure 1-24.

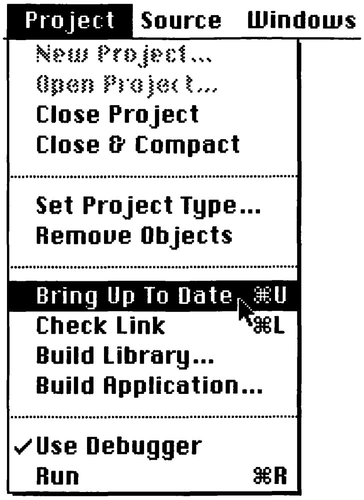


Figure 1-22
Bringing the project up to date by compiling all its files

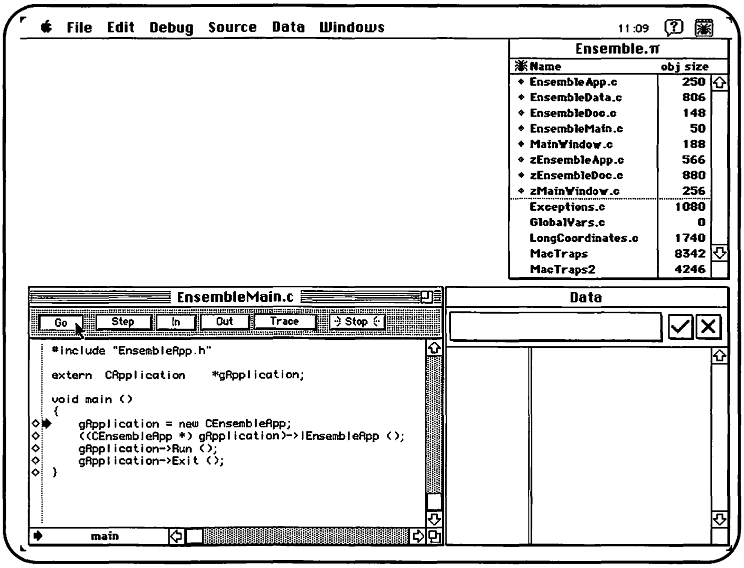


Figure 1-23
Ensemble application ready to run

13. When the application is running (Figure 1-24), the default window and menus are active. Make some selections from the **File** menu. If you choose **New**, another new untitled window will be created. If you choose **Close**, the current window will be closed. If you choose **Open**, then you will be given the opportunity to open a file. This will result in the file's name being displayed in the window title; how-

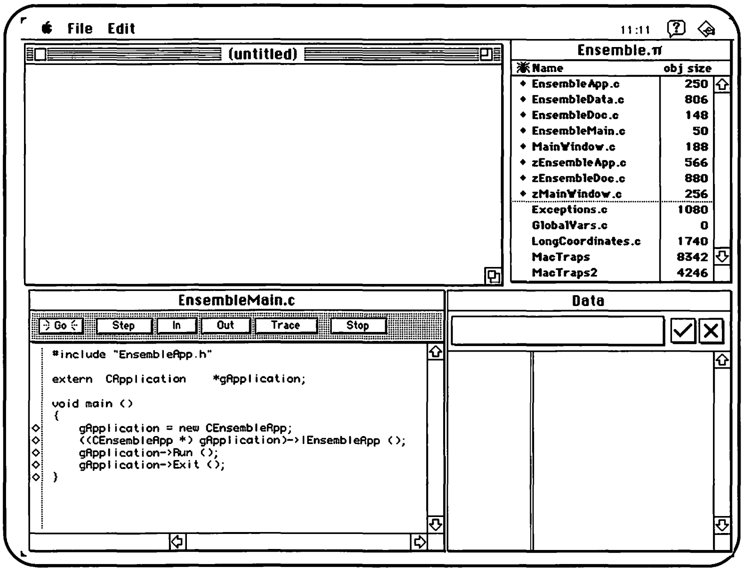


Figure 1-24
Ensemble application running

ever, because there is no code in the default program to display the contents of the file—because AppMaker is in no position to infer the format of the data in the selected file—nothing will be displayed at this time. The code to display the contents of various types of files used in the Ensemble application will be covered in later chapters.

14. When you have finished trying out the various commands, you should choose **Quit** from the **File** menu to stop execution of the application.

At this point, you have created, compiled, and executed a complete Macintosh application. It's true that it doesn't accomplish much, but we've all heard that creating the user interface for an application is the most difficult and time-consuming chore there is in program design. This is certainly true if it must be accomplished with inadequate tools; however, you have accomplished the feat with ease.

The combination of AppMaker's resource editor and code generation, and the TCL's extraordinary facilities for performing much of the work in animating the user interface, is something that you will come to appreciate more and more as you continue to develop the Ensemble application and grow more proficient at using the AppMaker and THINK C development tools described in this text.

Exercises

This book contains a number of exercises at the end of each chapter. Some of the exercises will be simple to complete, while others could be classified as relatively major projects. Even if you are not a student, it will be worthwhile for you to work the simple problems and think about the more complex tasks that are suggested as "extra-credit" projects. All "extra-credit" projects are noted as such in the text of the exercise or in a footnote.

1. Describe the features and functions of the AppMaker application, and contrast them with the features of Apple's ResEdit application.

2. Modify the contents of the **About** alert for the default application. This alert will be shown when the **About Application** command is chosen in the **Apple** menu for the application that is running.

3. Experiment with some of the tools in AppMaker, and create a few simple user interface elements. Generate code for a variety of languages and examine the results.

4. Contrast the code generated in exercise 2 for a procedural language, such as the procedural version of THINK C, with the object-oriented code generated for that same language.

5. Explain the purpose of the **Placeholder.c** file in the starting THINK C project file.

Chapter 2

Examining Ensemble's Structure

This chapter discusses the structure of the initial version of the **Ensemble** application's files, classes, and methods. It also describes how the generated classes and methods relate both to each other and to the THINK Class Library (TCL) routines.

The discussion begins with a description of the THINK C source program files generated by AppMaker. The files included in the **Ensemble** project can be grouped into two categories:

1. Those in the first category have names beginning with the letter 'z' and contain classes and methods that you should never need to modify. Each of the classes in these modules is referred to as a *superclass*. Each time you modify the **Ensemble.π.rsrc** resource file and then generate code, AppMaker will generate new contents for all the *superclass* files. There is nothing special about the letter 'z'; it is merely a standard adopted by AppMaker to aid in differentiating between the two categories of files.

2. In most cases, the second category of files contains direct descendants of the classes and methods in the *superclass* files. The file names in the second category do not begin with the letter 'z' and will never be automatically regenerated by AppMaker if the **Ensemble.π.rsrc** resource file is modified. These files usually contain *subclasses* of the corresponding *superclasses* and will eventually contain all the code that implements the application's unique functionality.

In general, each THINK C source file also has a corresponding header file, whose name ends in the extension '.h'. The exceptions to this rule, in the default set of files, are the file **En-**

sembleMain.c (which has no header file) and **ResourceDefs.h** (which has no corresponding source file).

As you follow along with the tutorials in the succeeding chapters of this book, you will be making modifications to the files in the second category. The automatically regenerated files should be treated as though they are "read only."

The header files (whose names end in '.h') contain the class and method declarations. The source files (whose names end in '.c') contain the method definitions (the code that implements each method's functionality). The default source files are as follows (their contents will be described in greater detail later in the chapter):

EnsembleMain.c
: This file contains the **main** function, in which execution initially commences.

zEnsembleApp.c
: This file contains the initialization method for **Ensemble's** application class. It also contains the methods to create new documents, open existing documents, set up the initial menus, update menu items, and handle menu commands. It will be regenerated each time the **Ensemble** resource file is modified.

EnsembleApp.c
: This file contains a method that specifies the type and creator for files read and/or written by the application. In addition, it contains methods that inherit and extend the behavior of the methods in the **zEnsembleApp.c** file for updating menus and handling menu commands.

zEnsembleDoc.c
: This file contains methods associated with the **Ensemble** application's document class. The methods are invoked to create a new file, open an existing file, save an existing file, revert to a previously saved version of an existing file, create the initial windows for the application, update menu items, and handle menu commands.

EnsembleDoc.c
: This file contains methods that override the behavior of its ancestor's methods in the **zEnsembleDoc.c** file. In particular, the file contains application-specific initialization and methods to update menu items and handle menu commands.

EnsembleData.c
: This file contains the methods that actually read, write, open, close, save, and dispose of data contained in Macintosh files. The methods in **zEnsembleDoc.c** call corresponding meth-

ods in the **EnsembleData.c** file to handle the specifics of the physical file formats.

zMainWindow.c — This file contains the initialization method to establish the appearance of the **MainWindow** resource (See page 9), as well as methods to update menu items and handle commands.

MainWindow.c — This file contains methods that override those in its ancestor class in **zMainWindow.c** for performing application-specific initialization, updating menu items, handling commands, and handling other events associated with the window's user interface items.

ResourceDefs.h — This file contains mnemonic definitions for each of the resources defined in the generated code. Instead of referencing a menu command by its number, you can use the corresponding mnemonic. When new resources are added to the **Ensemble.π.rsrc** file, the contents of the **ResourceDefs.h** file are rewritten.

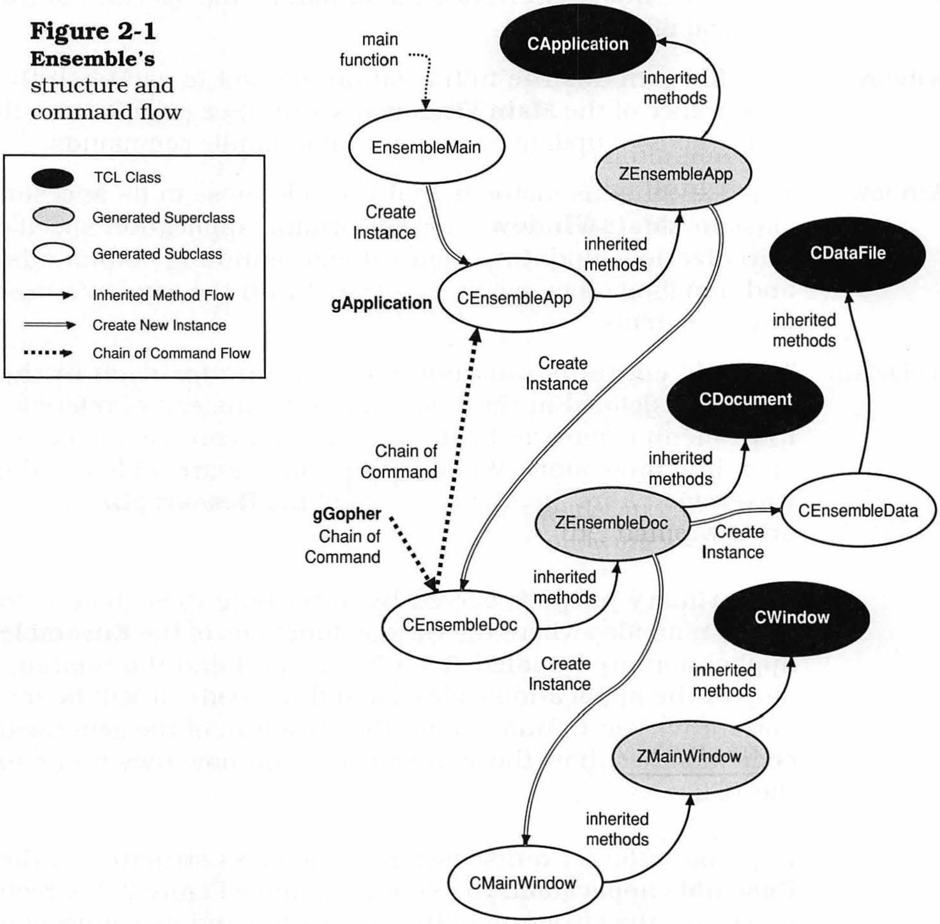
The primary purpose served by describing these files is to give you an idea where the various functions of the **Ensemble** application are handled. To fully comprehend the relationship of the application's classes and methods, it will be important for you to understand the structure of the generated code modules, how these interrelate, and how they relate to the TCL.

In preparation for a discussion of the class structure of the **Ensemble** application, you should examine Figure 2-1, which illustrates the structure of the application and its connection with the TCL. Notice in the figure that there are three categories of classes, indicated by the different background appearances of the ovals. The two sets of generated classes (shaded and unshaded) are respectively contained in the *superclass* and *subclass* files described earlier. The ovals with a black background refer to classes in the TCL. Not all of the TCL classes that interact with the **Ensemble** application are shown.

Ensemble's Classes and Methods

Figure 2-1 shows the relationship between the various classes in the application and the TCL. The **main** function, where execution begins, is located in the module whose name

Figure 2-1
Ensemble's structure and command flow



is **EnsembleMain**, in the figure. The **main** function is not represented as a method of any class. It is not object oriented. When it begins executing, none of the other *superclass* or *subclass* instances exist. The complete code for the **main** function is as follows:

*The **main** function is in the **EnsembleMain.c** file*

```
void main ()
{
    gApplication = new CEnsembleApp;
    ((CEnsembleApp *) gApplication)->IEnsembleApp ();
    gApplication->Run ();
    gApplication->Exit ();
}
```

It should be evident that the purpose of the **main** function is to create an instance of the **CEnsembleApp** class, initialize that instance, and then send it the **Run** and **Exit** messages. The presumption is that when the **Run** method is sent to the application, it will continue running and not return to the **main** function until the user has selected the **Quit** command. When this occurs, sending the application an **Exit** message gives it the opportunity to clean up and exit in an orderly fashion.

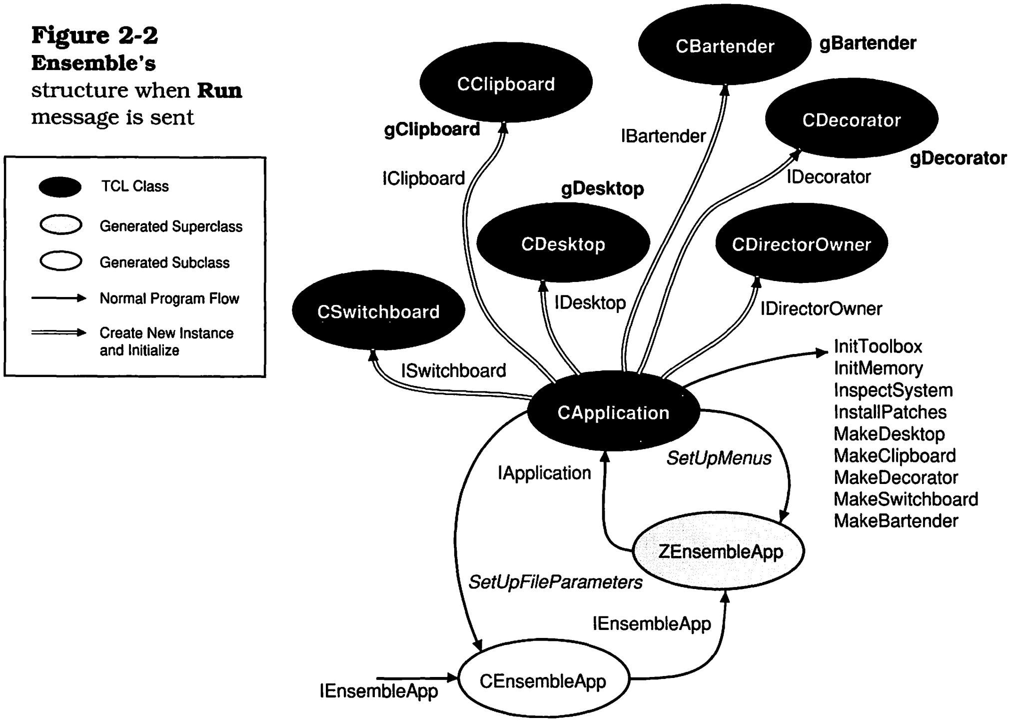
Messages can be sent to the application at any time, by referring to the global variable called **gApplication**, in which a handle to the application's instance is stored. Because the TCL's definition of the **gApplication** variable requires that it contain an instance of class **CApplication**, the foregoing code must "recast" **gApplication** as an instance of **CEnsembleApp** to call the initialization method.

The following section describes the actions that result from sending an **IEnsembleApp** message to the **CEnsembleApp** instance and how this simple message results in a set of actions that perform a host of initialization functions which prepare the application to begin execution.

CApplication's Initialization Method

Figure 2-2 shows the structure of the **Ensemble** application at the time the **IEnsembleApp** message is sent to the **CEnsembleApp** instance. Note that this message is inherited from its *superclass* instance, **ZEnsembleApp**. The *superclass* is responsible for performing all of the default initialization for the application and does so (in this case) by sending the **IApplication** message, which is processed by the corresponding method in the **CApplication** class in the TCL. The implications of executing the **IApplication** method are shown in the figure, which also illustrates the new object instances that are created during execution of the **IApplication** method. To reiterate, the **IApplication** message is sent to the **gApplication** instance in response to its **IEnsembleApp**'s method being called. When the **IApplication** method—inherited from **CApplication**—gains control, it creates instances of a number of additional classes, including the following:

Figure 2-2
Ensemble's structure when **Run** message is sent



CBartender This instance manages the menu bar and initially receives control when the mouse is clicked in the menu bar. **CBartender** can be addressed by use of the global variable **gBartender**.

CClipboard This instance controls the operation of the application clipboard and is responsible for exchanging its contents with the contents of the system clipboard when the application is activated or deactivated. It can be addressed by use of the global variable **gClipboard**.

CDecorator This instance handles the positioning of windows on the desktop. It is responsible for initially sizing and placing windows. It can be addressed by use of the global variable **gDecorator**.

CDesktop This instance manages a view that encompasses the entire screen. It manages a list of windows and can be addressed by use of the global variable **gDesktop**.

CDirectorOwner This class is part of the TCL. The **CApplication** class inherits functionality from the **CDirectorOwner** class. The **CDirec-**

torOwner class is shown in the figure because the superclass is initialized when the **IApplication** message is handled.

CSwitchboard This instance manages the main event loop and is responsible for dispatching events to other methods in the application. All events, including key presses, mouse clicks, update, activate, suspend, resume, and high-level (Apple) events are processed by **CSwitchboard**.

In addition to creating and initializing the preceeding instances, the **IApplication** method allocates the memory resources that are anticipated by the application for allocating handles and pointers in the application heap. In the course of this action, **IApplication** sets up a memory reserve, called the "rainy day fund," and allocates space for a number of master pointers. It also creates instances of two other entities: An instance of class **CList** is created to handle "Idle Chores," followed by creation of an instance of **CCluster** to hold "Urgent Chores."

Before continuing with the discussion of the **CEnsembleApp** class and the actions of the **IApplication** method, it is important to stress that instances of **ZEnsembleApp** and **CApplication**, as depicted in Figure 2-2, don't really exist. Instead, only the **CEnsembleApp** instance exists. When it is created, by virtue of the TCL object hierarchy, it inherits all the instance variables and methods of its ancestors. The **CEnsembleApp** instance *is* a **CApplication** object, in every sense of the word. Therefore, at this stage of our application's execution, only **CEnsembleApp** and the instances created by its **IApplication** inherited method actually exist. These include the **CDesktop**, **CClipboard**, **CDecorator**, **CBartender**, and **CSwitchboard** class instances. In this book, we will continue to show both the real (subclass) instance, its direct superclass, and that class's ancestor in the TCL, to aid in clarifying the relationship between the classes and the location of their corresponding methods. Hopefully, this will not mislead you into thinking that a multiplicity of instances exist for newly created objects that are deeply buried in the TCL's class hierarchy.

After the **CDesktop**, **CClipboard**, and **CDecorator** instances have been created and initialized, the **IApplication** method sends the **SetUpFileParameters** message, for which it has a default method. However, the **SetUpFileParameters** method is overridden by a method of the same name in the **CEn-**

sembleApp *subclass*. (Note that overridden methods are shown in oblique type in Figure 2-2.) Although the **SetUpFileParameters** method in the **CEnsembleApp** *subclass* first calls the corresponding inherited method in **CApplication**, this is a suitable place to customize the file types and creator you wish to use for your application. By default, AppMaker generates code to set a single file type of 'TEXT' and a signature (creator code) of 'XXXX'. These can easily be changed (as shown in a later chapter).

After sending the **SetUpFileParameters** message, the **IApplication** method sends the **SetUpMenus** message, for which it also has a default method. Once again, however, this method has been overridden by a method with the same name in the **ZEnsembleApp** *superclass*. The purpose of the override in this case is to load and initialize any special menus not handled by the normal operation of the **IBartender** method (such as pop-up menus and the like). Because our application does not currently have any special menus to initialize, the generated code merely calls the inherited **SetUpMenus** method in the **CApplication** class.

The last act of the **IApplication** method is to set the value of the **gGopher** global variable to point to the **CEnsembleApp** instance. (The **gGopher** is a global variable that points to the currently active member of the *Chain of Command*.) Contrary to the stable state shown in Figure 2-1, where the **gGopher** global variable points to the **CEnsembleDoc** instance, an instance of this class doesn't exist at the time **IApplication** is called. Therefore, **gGopher** is set to point to the **CEnsembleApp** instance.

CApplication's Run Method

After **CEnsembleApp** receives the **IEnsembleApp** message, and the preceding sequence of events is complete, the **main** module sends a **Run** message to the **CEnsembleApp** object. The **Run** method is not overridden by methods in the **CEnsembleApp** or **ZEnsembleApp** classes. Instead, the message is directed to the **Run** method inherited from the **CApplication** class.

It is important to note that at the time the **Run** method is executed, the **Ensemble** application may have been initially invoked in one of two different ways:

1. The **Ensemble** application's icon can be double-clicked, or it can be selected and then the **Open** command in the Finder's **File** menu can be chosen.

2. One or more of **Ensemble's** files (types that carry the **Ensemble** application's creator code) is selected, and then either the Finder's **Print** command or **Open** command is chosen from its **File** menu.

In the first case, nothing special needs to be done inside the **CApplication** class's **Run** method. In the second case, however, the selected files must be opened or printed, as required. One of the first actions of the **Run** method is to determine in which way the application was invoked and then handle that situation in an appropriate manner. This is accomplished by invoking the **Preload** method, which performs the following actions:

1. If the icon was double-clicked or the icon was selected and then opened, the **Preload** method does nothing, and the **Run** method can begin processing events.

2. If one or more files were selected, and either the **Open** or **Print** Finder command was chosen, the **Preload** method is obligated to open the chosen files, one by one, and process them in an appropriate manner. In the case of the **Open** command, the application is sent an **OpenDocument** message, which happens to be overridden by our *superclass*, **ZEnsembleApp**. In the case of the **Print** command, a **DoCommand** message with a parameter of **cmdPrint** is sent to the application, which in our case is ignored (for the moment).

The modularity of this approach is important when it is necessary to override the ***GetAnEvent*** *method to "peek" into the event queue.*

After the **Open** or **Print** command has been handled, the **Run** method resumes control and begins processing events. It sends a **Process1Event** message, which is handled by a method inside the **CApplication** class. This method sends a **ProcessEvent** message to the **CSwitchboard** instance, which, in turn, sends a **GetAnEvent** message that is normally processed by its method of the same name in the **CSwitchboard** instance.

If the **GetAnEvent** method returns with a valid event, then a **DispatchEvent** message is sent. This is usually handled by the method of that name in the **CSwitchboard** instance. If no event is currently in the queue, then the **GetAnEvent** method sends a **DoIdle** message, which is handled by its method of that name, which sends an **Idle** message to the application by referring to the **gApplication** global variable.

After the event has been processed (event processing is covered later in the chapter), the **CApplication** class's **Process1Event** method regains control. It then determines whether any urgent chores need to be processed and if so, performs them one by one. Finally, it handles switching to and from a desk accessory, if necessary, cleans up, and returns to the event-processing loop inside the **Run** method. Events are continually processed inside this method until something resets the **CApplication** instance's **running** variable to FALSE. When that occurs, the application returns to the **main** function (inside the **CEnsemble** module), at which time the **Exit** message is sent to the **CEnsembleApp** instance. In the case of our application, this message is handled by the **Exit** method inherited from **CApplication**, which is an empty (do-nothing) method. The **main** function then returns to the operating system, where the Finder regains control.

Processing Events

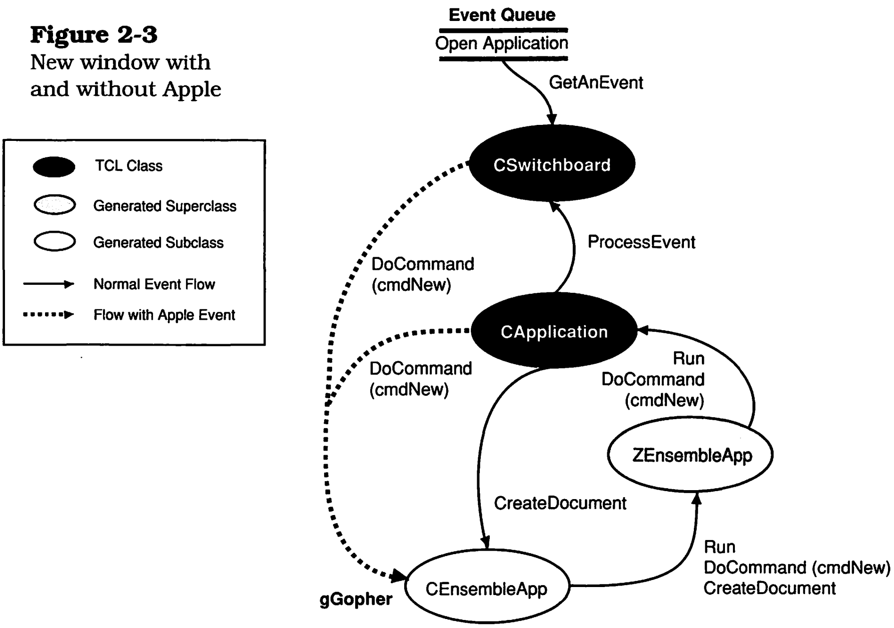
When the **Preload** method is ready to return to the **Run** method, it sends the application a **StartUpAction** message, which is handled by a method of that name in the **CApplication** class. The **StartUpAction** method tests whether any files were preloaded by either the **Open** or **Print** commands and also whether the application environment supports high-level Apple Events. If neither of these conditions is true, then the method sends a **DoCommand** message, with a **cmdNew** parameter, to the instance referenced by the current value of the global **gGopher** variable. This results in the execution of a **New** command, as though the user had chosen the command from the application's **File** menu.

If no files were preloaded and the application is capable of receiving high-level Apple Events, then when the application begins processing events, it will discover an **Open Application** event (placed in the queue by the Finder). It will handle this event by sending a **DoAppleEvent** message, containing the **Open Application** event code, to the instance referenced by the current value of the global **gGopher** variable (which points to the **CEnsembleApp** instance at this point in our application).

The **DoAppleEvent** message will be handled by a method of the same name in the **CApplication** class, which will send a **DoCommand** message with a **cmdNew** parameter to the instance referenced by the **gGopher** variable. This will result in the creation of a new document, just as if the user had chosen the **New** command from the application's **File** menu.

Although the preceeding process seems rather circuitous, it is necessary in an environment in which an application can be started by any other application simply by sending the Finder a request to start it. It should also be evident that the TCL automatically handles a variety of situations. The entire set of linkages is illustrated in Figure 2-3.

Figure 2-3
New window with and without Apple



Sending the application (**gGopher**) the **DoCommand** message with **cmdNew** as a parameter starts another sequence of events, which is described in the next section. Bear in mind that event processing is the primary job of any Macintosh application; every application action is triggered by an event of some kind.

Handling the DoCommand (cmdNew) Message

When the **DoCommand** message is sent to the **gGopher** instance with a parameter of **cmdNew**, the **CEnsembleApp** doesn't recognize that command, so it passes it on to its *superclass*, **ZEnsembleApp**, which then passes it on to the **CApplication** class method of the same name. The **DoCommand** method in the **CApplication** class handles the **cmdNew** parameter by sending a **CreateDocument** message to the current application instance (**CEnsembleApp** in our case). The method that implements this message is inherited from the **ZEnsembleApp** *superclass*.

The **CreateDocument** method in the **ZEnsembleApp** class is responsible for creating a new subclass of the **CDocument** class, which is the supervisor of the data file and default window associated with the **Ensemble** application. The code for the **ZEnsembleApp** class's **CreateDocument** method is as follows:

The TRY, CATCH, *and* ENDTRY *statements are part of the TCL's error recovery features.*

```
void ZEnsembleApp::CreateDocument(void)
{
   CEnsembleDoc*theDocument;

   TRY
   {
      theDocument = new CEnsembleDoc;
      theDocument->IEnsembleDoc (this, TRUE);
      theDocument->NewFile ();
   }
    CATCH
   {
      ForgetObject (theDocument);
   }
    ENDTRY;
}
```

The foregoing code was entirely generated by AppMaker. It uses the new error-handling features of the TCL, which include the ability to place statements that might fail inside a block headed by the TRY keyword and the ability to put the error-handling code inside a block headed by the CATCH keyword. The CATCH block is ended by an ENDTRY keyword.

The function of the **CreateDocument** code is to create a new instance of class **CEnsembleDoc**, which is a *subclass* of **ZEnsembleDoc**, which is itself a subclass of **CDocument**, as shown in Figure 2-1.

After the instance is created, it is initialized by sending the **IEnsembleDoc** message, which is inherited from the **ZEnsembleDoc** *superclass*. The initialization consists of setting the **itsMainWindow** instance variable to NULL and then sending an **IDocument** message, which is handled by the inherited method of that name in the **CDocument** class. This message serves to initialize a number of the instance variables inherited from the **CDocument** class.

After the **CEnsembleDoc** instance has been initialized, it is sent a **NewFile** message, which is handled by the method of the same name inherited from and contained within the **ZEnsembleDoc** module. The code for the **NewFile** method is as follows:

```
void ZEnsembleDoc::NewFile (void)
{
   CEnsembleData  *theData;

   TRY
   {
      theData = new CEnsembleData;
      theData->IEnsembleData (this);
      itsFile = theData;

      BuildWindows (theData);
      itsWindow->Select ();
   }
    CATCH
   {
      ForgetObject (theData);
   }
    ENDTRY;

}
```

In the **NewFile** method (inherited from the **ZEnsembleDoc** module), a new instance of class **CEnsembleData** is created. Although there is no generated *superclass* for this instance, it inherits its behavior and instance variables from the TCL's **CDatafile** class—which, in turn, inherits instance variables and methods from the **CFile** class. When the **CEnsemble-Data** instance has been created, it is sent the **IEnsemble-Data** message, to initialize the instance. The initialization code is as follows:

```
void CEnsembleData::IEnsembleData   (CDocument   *theDocument)
{
   inherited::IDataFile ();
   hasFile = FALSE;
   itsDocument = theDocument;

   // your application-specific initialization
   itsData = NULL;
}
```

The preceeding code was generated by AppMaker. It first sends an **IDataFile** message, which is inherited from the **CDatafile** class, and then initializes the **hasFile** instance variable to FALSE, indicating that no file is currently open for this document. The **itsDocument** instance variable points back to the **CEnsembleDoc** instance, so that the **CEnsembleData** instance can subsequently refer to the document's methods. The **itsData** instance variable is set to NULL, indicating that no data currently exist. Note that AppMaker has indicated with a comment that this is a good place to insert additional initialization code that is pertinent to the **CEnsembleData** instance's functionality. None is needed at this time.

The **NewFile** method (shown on page 33) follows up the initialization of the **CEnsembleData** instance by setting the **itsFile** instance variable to the value of the **CEnsembleData** instance. It then sends a **BuildWindows** message, which is handled by the method of the same name inherited from the **ZEnsembleDoc** *superclass*. The code for this method is as follows:

```
void ZEnsembleDoc::BuildWindows(void)
{
   itsMainWindow = new CMainWindow;
   itsMainWindow->IMainWindow (this, itsData);
   gDecorator->StaggerWindow (itsMainWindow);
   itsMainPane = itsMainWindow->itsMainPane;
   itsWindow = itsMainWindow;
}
```

The purpose of the **BuildWindows** method is to create the windows that are intended to be open initially in the application. In our case, this is a single window whose default *subclass* name is **CMainWindow**. The method begins by creating an instance of **CMainWindow**, and then initializes the window, passing it arguments of **this (CEnsembleDoc)** and the value of the **itsData (CEnsembleData)** instances.

The **CEnsembleDoc** instance is the *supervisor* of the window, as required by window initialization methods, and passing the **CEnsembleData** instance allows the window to be able to refer to the instance variables and methods in that *subclass*. In particular, if data are entered into the window, it will be possible to mark the window as *dirty* and refer to other data structures associated with the data class instance. The code for **IMainWindow** is as follows:

```
void  CMainWindow::IMainWindow(CDirector*aSupervisor,
                               CEnsembleData*theData)
{
   itsData = theData;
   inherited::IZMainWindow (aSupervisor);
   // any additional initialization for your window
}
```

Once again, this code was wholly generated by AppMaker. The reference to the **CEnsembleData** instance is saved in the window's **itsData** instance variable, and then the **IZMainWindow** message (inherited from the **ZMainWindow** class) is sent. The code for the **IZMainWindow** method is as follows:

```
void  ZMainWindow::IZMainWindow(CDirector      *aSupervisor)
{
   CView          *enclosure;
   CBureaucrat    *supervisor;
   CSizeBox       *aSizeBox;
   IWindow (MainWindowID, FALSE, gDesktop, aSupervisor);
   itsMainPane = NULL;
   enclosure = this;
   supervisor = this;
   aSizeBox = new CSizeBox;
   aSizeBox->ISizeBox (enclosure, supervisor);
}
```

The **IZMainWindow** method (inherited from the **ZMainWindow** class) calls the **IWindow** method inherited from the TCL's **CWindow** class. Then the **IZMainWindow** method sets the **itsMainPane** instance variable to NULL, indicating that no pane currently exists.

The **IZMainWindow** method sets local variables called **enclosure** and **supervisor** to point to **this**, which is the **CMainWindow** instance. It then creates an instance of a **CSizeBox** class and initializes that class, making the window the supervisor and enclosure of the size box that appears at the lower right-hand corner of the default window.

When the **IZMainWindow** and **IMainWindow** methods return to the **BuildWindows** method (see page 34), that method sends the **CDecorator** class instance (via the global **gDecorator** variable) a **StaggerWindow** message, which staggers the window with respect to any other windows on the screen. This ensures that all active windows are at least partially visible. The last act of **BuildWindows** is to set the **itsWindow** instance variable to the value of **itsMainWindow** (**CMainWindow** in our case).

The **BuildWindows** method returns to the **CEnsembleDoc NewFile** method (see page 33), which sends the window a **Select** message, making the window visible. This is the culmination of handling the **cmdNew** command that was created within the **CApplication**'s **Run** method.

Examining the Chain of Command

A command is defined either as an item selected from one of the application's menus or a keyboard shortcut for that item (e.g., typing Command-C, instead of choosing **Copy** from the **Edit** menu). In addition, AppMaker creates "click commands" for buttons, checkboxes and radio buttons. Commands begin as events that are fetched from the event queue and processed according to the following rules:

- ❖ The **CApplication** instance's **Run** method sends the **Process1Event** message, which is handled by the method of that same name in the **CApplication** class.
- ❖ The **Process1Event** method sends a **ProcessEvent** message to the **CSwitchboard** instance.

- The **ProcessEvent** method sends a **GetAnEvent** message, which is handled by a method of that name in the **CSwitchboard** instance.

- The **GetAnEvent** method in **CSwitchboard** calls the Macintosh event manager to fetch an event. Upon returning to the **ProcessEvent** method, the event is examined. If no event or a system event was fetched, then **ProcessEvent** sends a **DoIdle** message. If an event for this application was fetched, then **ProcessEvent** sends a **DispatchEvent** message, which is also handled in **CSwitchboard.**

- The **DispatchEvent** method discriminates between the various types of events (mouse events, key presses, disk events, update, activate, high-level events, etc.) and sends a message to the appropriate handler. In the case of a mouse-down in the menu bar, **DispatchEvent** sends a **DoMouseDown** message.

- The **DoMouseDown** method in **CSwitchboard** sends a **DispatchClick** message to the **CDesktop** instance.

- The **DispatchClick** method discriminates between the various places on the desktop in which a mouse click can occur and sends an **UpdateAllMenus** message to the **CBartender** instance. It also sends a **MenuSelect** message, and if a menu command was selected, it sends a **DoCommand** message to the instance stored in the global **gGopher** variable (in our case, **CEnsembleDoc**).

- If, instead of a mouse click, the **DispatchEvent** method recognizes a key press event, it sends a **DoKeyEvent** message, which is handled by the method of that name in the **CSwitchboard** instance. This method determines whether the Command key is down, and if so, it sends an **UpdateAllMenus** message to the **CBartender** instance. Then, if a valid Command key combination was entered, **DoKeyEvent** sends a **DoCommand** message to the instance stored in the global **gGopher** variable (which, again, is our **CEnsembleDoc** instance).

Looking at the default code generated by AppMaker, you can see that when the **CMainWindow** instance is created, the **itsGopher** instance variable is set to point to the **CEnsemble-**

Doc instance, as shown in Figure 2-1. This means that all commands will first be handled by the **DoCommand** method in the **CEnsembleDoc** instance.

If you examine the code for the **DoCommand** method in the **CEnsembleDoc** class, you will see that it does not handle even a single command, but, instead, calls the inherited **DoCommand** method in the TCL's **CDocument** class. The code for the **CEnsembleDoc** class's **DoCommand** method is as follows:

```
void EnsembleDoc::DoCommand(long theCommand)
{
   switch (theCommand)
   {
      default:
      {
         inherited::DoCommand (theCommand);
         break;
      }
   }
}
```

The commands handled by the **DoCommand** method inherited from the **CDocument** class include **cmdSave**, **cmdSaveAs**, **cmdRevert**, **cmdPageSetup**, **cmdPrint**, and **cmdUndo**. All of these commands correspond to similarly named items in the **File** and **Edit** menus. If the chosen command is not one of these, then the **DoCommand** method inherited from the **CBureaucrat** class (which is an ancestor of the **CDocument** class in the TCL) will send the **DoCommand** message to the supervisor of the current instance (which in this case would be **CEnsembleApp**).

The **DoCommand** method of **CEnsembleApp** doesn't do much to handle any other command, but it is worthwhile to look at its code:

DoCommand *method code (beginning)*

```
void CEnsembleApp::DoCommand (long theCommand)
{
   short theMenu;
   short theItem;
   Str255 theItemText;

   if (theCommand < 0)
```

DoCommand
method code
(concluded)

```
        {
            /* menu generated dynamically */
            theMenu = HiShort (-theCommand);
            if (theMenu == MENUapple)
            {
                /* handle Apple menu in superclass */
                inherited::DoCommand (theCommand);
            }
            else
            {
                theItem = LoShort (-theCommand);
                GetItem (GetMHandle (theMenu), theItem, theItemText);
                /* do the right thing with the text of the item */
            }
        }
        else
        {
            switch (theCommand)
            {
                default:
                    inherited::DoCommand (theCommand);
                    break;
            }
        }
    }
```

There are several important features of AppMaker's generated code in the **CEnsembleApp** instance's **DoCommand** method. First, the code tests whether the command number is negative. This will be the case only for desk accessories or other dynamically generated menu commands.

The handling of Apple menu items is relegated to the superclass (**ZEnsembleApp**). Other menu commands created at run time should be handled by adding code at the place indicated by the comment. If the command number is positive, then the command is automatically passed to the inherited **DoCommand** method in the *superclass.*

The **DoCommand** method in the **ZEnsembleApp** instance also handles a single instance: the **About Application** command from the Apple menu, as shown in the following code:

DoCommand
method code
(beginning)

```
void  ZEnsembleApp::DoCommand(long  theCommand)
{
    short      itemNr;
```

DoCommand
method code
(concluded)

```
  switch (theCommand)
  {
    case cmdAbout:
    {
      itemNr = Alert (1, NULL);
      break;
    }
    default:
    {
      inherited::DoCommand (theCommand);
      break;
    }
  }
}
```

When the **About Application** command is chosen from the Apple menu, the foregoing code will display an Alert. All other commands are passed to the inherited **DoCommand** method. In this case, the **DoCommand** method in **CApplication** is invoked. This method handles the commands **cmdNew**, **cmdOpen**, **cmdClose**, **cmdQuit**, **cmdUndo**, **cmdCut**, **cmdCopy**, **cmdPaste**, **cmdClear**, and **cmdToggleClip**. All of these correspond to commands in the **File** and **Edit** menus.

CApplication's DoCommand method also handles any commands with negative command numbers that are passed up the chain of command for it to handle. If no other method in the chain of command is able to handle it, a command will be ignored if the **DoCommand** method in **CApplication** cannot handle it.

Examining Event Handling

To recap, events are continuously fetched by the loop inside the **Run** method of the **CApplication** class.

The **Run** method sends the **Process1Event** message, which sends a **ProcessEvent** message to the **CSwitchboard** instance. The corresponding **ProcessEvent** method sends **GetAnEvent** and then **DispatchEvent** messages to the **CSwitchboard** instance. The **DispatchEvent** method is the crucial discriminator in how various types of events are subsequently handled.

In the discussion regarding the dispatch of commands (either from menu choices or by combinations keyboard commands), occurrence of the event was followed by sending a **DoCommand** message to the current instance held in the **gGopher** global variable.

Handling of events other than commands takes place in a different fashion. When the **DispatchEvent** message is sent, the corresponding method determines what type of event has occurred and how it should be handled. Some of the possibilities are:

- In all cases, a mouse click is sent to the **CDesktop** instance for resolution by its **DispatchClick** method. Clicks in the menu bar are handled as described in the discussion of the chain of command. Other possibilities for mouse clicks are:
    - If the mouse click occurs on an insignificant part of the desktop, the number of clicks is counted and the **DoClick** message is sent; however, neither **CDesktop** nor its **CView** ancestor performs any function for this message.
    - If the mouse click occurs in a "system window" (i.e., a desk accessory), the toolbox **SystemClick** routine is used to handle the event. If this is the case, the application's event handler has washed its hands of the event, and no further processing takes place.
    - If the mouse click occurs in the content region of a window, then if the window is inactive, it is selected, and if the **actClick** instance variable for the window is TRUE, the window is sent an **Activate** message. If the **wantsClicks** instance variable for the window is TRUE, then a **DispatchClick** message is sent to the window; otherwise, the click is handled in the same way as a click in the desktop (i.e., it is essentially ignored).
    - If the click occurs in the *drag region* of the window's title bar, a **Drag** message is sent to the window. The **CWindow** class's **Drag** method sends a **DragWind** message to the **CDesktop** class to handle dragging the window on the desktop.
    - If the mouse click event occurs in the *grow box* of a window, then a **Resize** message is sent to the window. The

Resize method of the **CWindow** class handles this event by calling the **GrowWindow** toolbox call, and then **Resize** sends a **ChangeSize** message to change the window's physical size within the maximum and minimum size constraints for the window. **Resize** also sends an **Update** message to the window, to force it to redraw its contents.

- If the click occurred in the *go-away box* of a window, then a **Close** message is sent to the window. The **CWindow** class handles the **Close** message by sending the window's supervisor a **CloseWind** message. In the case of the **Ensemble** application, **CEnsembleDoc** is the supervisor of the window, and a **CloseWind** message would be handled by the inherited method of that name from the **CDocument** class.

- If the click occurred in the *zoom box* of a window, then if a mouse-up event also occurs in that box, the window is sent a **Zoom** message. The **CWindow** class handles this case by calling the **ZoomWindow** toolbox method, and then **ZoomWindow** adjusts the size of all the subviews by sending them an **AdjustToEnclosure** message.

❖ Mouse-up events result in sending a **DoMouseUp** message to the last view that was referenced by the initial mouse-down click (held in the **gLastViewHit** global variable).

❖ For key-down, key-up, or repeated key (autoKey) events, the **CSwitchboard's DispatchEvent** method sends a **DoKeyEvent** message, which the **CSwitchboard** class's method of that name handles differently, depending on the type of event that is involved. In most cases, the **gGopher** global variable contains the destination instance to receive the event. This allows keystrokes to be sent directly to an active text field, minimizing the dispatch time.

- If the event is a key-down event and the Command-key is also pressed, the event is treated as a command, as previously described.

- If a key-down event was the **F1** function key, it is handled as an **Undo** command by sending a **DoCommand** message with **cmdUndo** to the current **gGopher** instance.

- If a key-down event was the **F2** function key, it is handled as a **Cut** command by sending a **DoCommand** message with **cmdCut** to the current **gGopher** instance.
- If a key-down event was the **F3** function key, it is handled as a **Copy** command by sending a **DoCommand** message with **cmdCopy** to the current **gGopher** instance.
- If a key-down event was the **F4** function key, it is handled as a **Paste** command by sending a **DoCommand** message with **cmdPaste** to the current **gGopher** instance.
- All other key-down events result in sending a **DoKeyDown** message to the current **gGopher** instance.
- Key-up and repeated key events are sent as **DoKeyUp** and **DoAutoKey** messages, respectively, to the current **gGopher** instance.

❖ The **DispatchEvent** method also handles "disk events" by sending a **DoDiskEvent** message, which the **CSwitchboard** method of that name ignores, unless the event *message* indicates that an error has occurred (i.e., the disk requires formatting). If this is the case, an alert is displayed and the user is given the opportunity to format the disk.

❖ If an **Update** event occurs, **DispatchEvent** sends a **DoUpdate** message, which its method of that name handles by sending an **Update** message to the window associated with the **Update** event. **Activate** and **Deactivate** events are handled in the same way, by sending a **DoActivate** or a **DoDeactivate** message, which, in turn, causes an **Activate** or a **Deactivate** message to be sent to the appropriate window.

❖ **Suspend** and **Resume** events occur when another application is selected while running under Multifinder or when the current application is being resumed after a previously active application was suspended while running under Multifinder. **DispatchEvent** sends a **DoSuspend** or **DoResume** message in this case, which, in turn, causes a **Suspend** or a **Resume** message to be sent to the **gApplication** instance.

- High-level events (Apple Events) are handled by sending a **DoHighLevelEvent** message, which the **CSwitchboard** method handles by checking whether the system is capable of handling Apple Events, and then if so, calling the **AEProcessAppleEvent** toolbox routine to handle the standard Apple Events. If an application is capable of handling other than the standard Apple Events, then it can override the **DoHighLevelEvent** method and process the additional high-level events.

- If any other event occurs, **DispatchEvent** sends a **DoOtherEvent** message, which results in the method of that name in the **CSwitchboard** class being invoked. The **DoOtherEvent** method is empty, but can be overridden to produce any other desired behavior.

Summary of Ensemble's Structure and Capabilities

The **Ensemble** application is quite useless in its present form. While most of the necessary structural members are implemented, the application lacks a purpose or intrinsic functionality.

This chapter has focused on the structure of the application and how commands and events are handled, with the intention of convincing you that almost every application that you create will embody at least the default features described in the foregoing text.

Most applications have at least one window and at least the standard **Apple**, **File**, and **Edit** menus. Commands and events are handled identically, regardless of the structure or complexity of an application built upon the THINK Class Library. The function of the global **gGopher**, **gDesktop**, **gBartender**, **gApplication**, and other global variables does not change as the application increases in complexity.

Subsequent chapters will discuss the procedures for transforming the **Ensemble** application into a worthwhile program. Each major addition to the application will be built upon what has previously been presented. This is a standard technique used when constructing applications.

While you may elect to define a greater percentage of the user interface in a single session, incrementally adding the features described, combined with AppMaker's power to maintain the integrity of your unique code, provides the incentive to approach the application development process in a step-by-step, methodical way.

Exercises

1. Explain how AppMaker's code-generation approach for THINK C will allow subsequent changes to be made to the user interface without requiring the programmer to cut and paste code from one generation to the next.

2. Describe the situations in which AppMaker's code generation approach will inhibit rather than help during the development of a large program. (*Hint*: Suppose you decide to redesign the user interface completely. What effect would this have on the generated code?)

3. When the user clicks the mouse button in various areas of the screen, what TCL method receives these events, and where are they dispatched?

4. What happens when the user enters keystrokes when the application is active? Why is there no visible result of these keystrokes in our default Ensemble application?

5. What is the purpose of the TRY and CATCH blocks in the generated code? Explain what service these perform. (*Hint*: Look in the *Object-Oriented Programming Manual* for THINK C.)

6. Explain the purpose of the **gGopher** variable, what it contains, and how it serves the object-oriented application. In particular, how is the **gGopher** variable related to the handling of events?

7. Assuming that Figure 2-1 illustrates the universe of objects at the time the illustration was drawn, how many actual object instances exist? Explain.

8. Describe the meaning of the terms "encapsulation," "inheritance," and "polymorphism" with respect to object-oriented programs.

Chapter 3

Creating the Ensemble Application

This chapter begins the task of improving and customizing the operation of the **Ensemble** application. Up to this point, we have used AppMaker to generate a default user interface, with standard menus and a single window, but no specific functionality. In this and the succeeding chapters, we will mold the user interface and the code into a useful application.

It is the intent of this book to document the creation of a single, but nontrivial application, discussing many of the features of the TCL and how they are applied to realize a variety of functions inside the application. The application is named **Ensemble** because it embodies a set of cooperative functions inside a cohesive framework. It *is* an ensemble of functionality.

In order to explore most of the TCL's intrinsic capabilities, the **Ensemble** application will incorporate the features of text editing, spreadsheet, and graphing functions. Together, these will operate as a cooperative ensemble. The main intention is to explore the facilities of the TCL that support text editing, rectangular cellular tables, and drawing functions. With some experience in all these disciplines, it should be quite easy to apply these techniques to other, similar applications.

Adding Text-editing Features to Ensemble

The first, and easiest feature to add to the **Ensemble** application is a text-editing window. The intention of the design is not to provide an editor that is of desktop-publishing quality, but to create a window that contains text in a single font, size, style, and justification, with limited cut-and-paste editing abilities. The design is similar to a notebook in which you would write short sections of text. The design will allow you to have multiple open text windows, and each of these can have its own text font, size, style, and justification.

The primary purpose of adding features to the Ensemble application in a piecemeal fashion is to illustrate how easy it is to add capabilities incrementally to an existing object program. We'll be using AppMaker to create the new user interface elements, and then we will add the necessary code to bring the interface to life. Once the coding is complete, we'll compile and run the application.

Using AppMaker to Enhance the MainWindow

Recall that the resource file for the Ensemble application is named **Ensemble.π.rsrc**. If you double-click on this file, you will launch AppMaker, telling it to use the file. In the following tutorials, we will be adding functionality to the previously created resource file, and then we will also generate code to operate the new resources. The steps to enhance the Ensemble application's resource file are as follows:

1. Launch AppMaker by double-clicking on the **Ensemble.π.rsrc** file.

2. You will see a screen that looks like that shown in Figure 1-5 on page 6. Only the selection window is displayed, and it contains the currently defined menu bars. In our case, only the **MainMenu** bar exists at this time.

3. Click the mouse cursor on the **Select** menu and choose **Windows**, as shown in Figure 3-1.

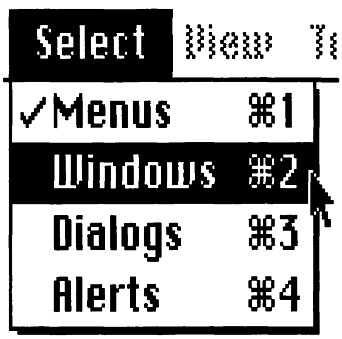


Figure 3-1
Selecting Windows in AppMaker

4. Note that the current selection window now displays the Clipboard and default **MainWindow** entries. Double-click on the **MainWindow** entry, as shown in Figure 3-2.

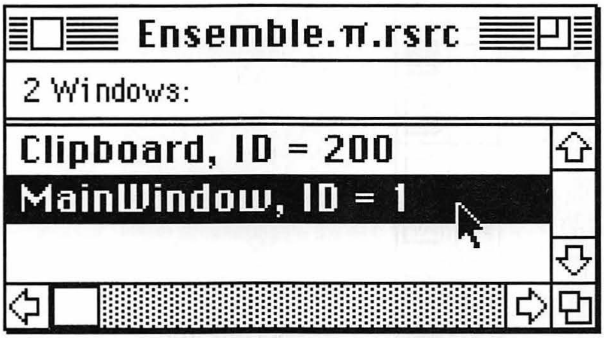


Figure 3-2
Picking the **MainWindow** entry

5. When the **MainWindow** entry is selected, the default window that we created in Chapter 1 will be displayed. This window has the appearance shown in Figure 3-3. Notice that there are no scroll bars or other indications that the window contains an editing pane. In fact, it does not. The purpose of the next few steps is to instruct you how to construct a text-editing pane inside this **MainWindow** definition. You start the process by clicking the **Scroll-Pane** tool, as shown in Figure 3-4.

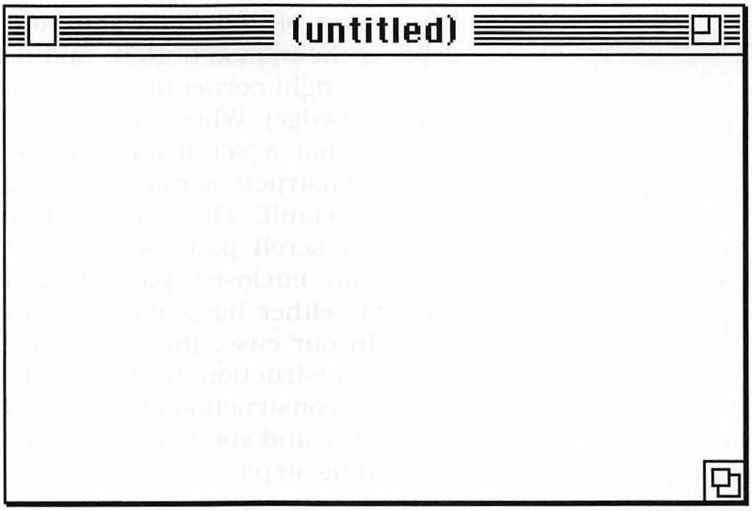
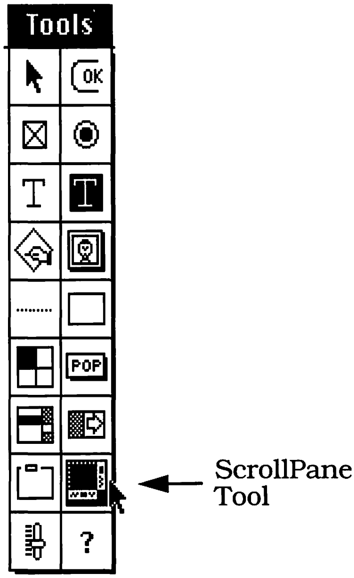


Figure 3-3
Ensemble's default **MainWindow** definition

Figure 3-4
Selecting the ScrollPane tool from the Tools menu



6. When you select the **ScrollPane** tool, the cursor changes into a cross, and you should position the cross at the top left corner of the blank portion of the window pane, depress the mouse button, and drag the cursor down to the lower right corner of the window pane (right to its bottom right edge). When you release the mouse button, you will see that a scroll pane has been constructed. AppMaker constructs scroll panes with only a vertical scroll bar, by default. The horizontal scroll bar can easily be added. A scroll pane is a pane with a scroll bar that allows an enclosed pane (called a panorama) to be scrolled—either horizontally, vertically, or in both directions. In our case, the panorama is the **EditText** pane, whose construction is described in the next step. The complete construction of the text-editing pane is shown in Figure 3-5, and you may wish to refer to this figure for the succeeding steps.

7. The next step is to add the **EditText** pane. Select the **EditText** tool, as shown in Figure 3-6. The cursor will again turn into a cross, and you should move it to the top left corner of the scroll pane, click and hold the mouse button down, and drag the mouse down to the lower right corner of the blank portion of the scroll pane. The **EditText** pane will show lines that correspond to the line

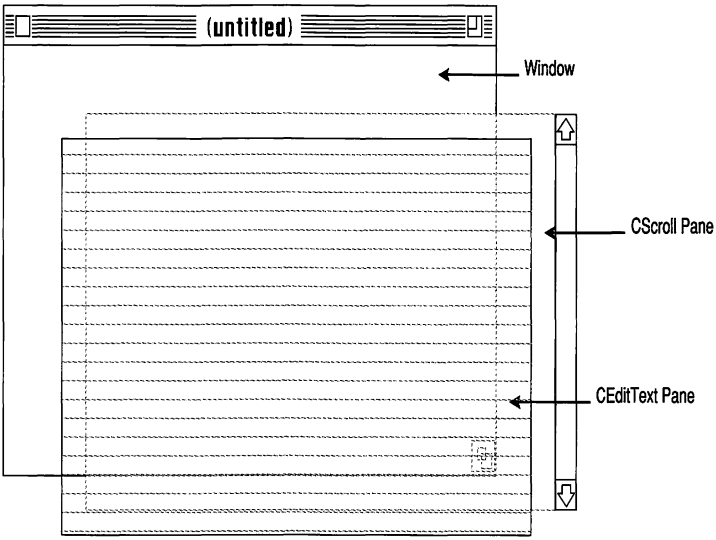


Figure 3-5
Adding a ScrollPane to the **MainWindow**

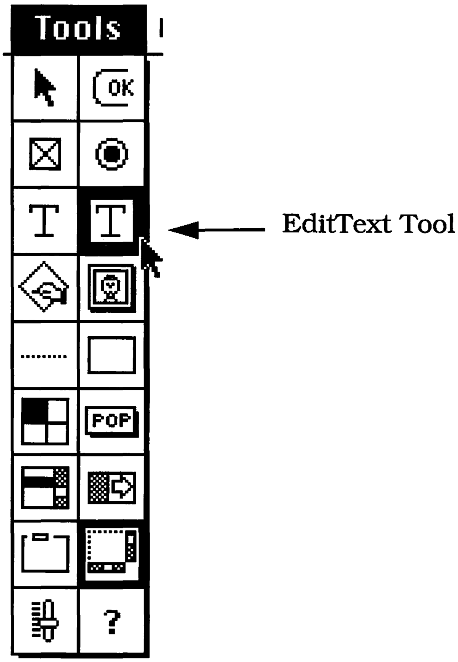


Figure 3-6
Selecting AppMaker's EditText tool

spacing of text in the default font, size, and style. You don't have to bother changing this at the present time: We're going to provide the user with a method of changing the text font, size, and style that will change the appearance of the entire text in a single operation. The complete text-editing window appears as shown in Figure 3-7.

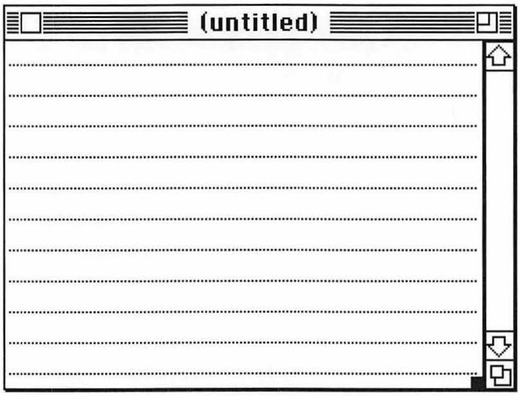


Figure 3-7
Complete text-editing window

This completes the creation of the text-editing window. However, in order to provide the user with the ability to change the text font, size, and style, we are going to create a new menu, a menu command, and a corresponding dialog box for changing the text window's appearance.

Adding a New Menu to Ensemble

The next few steps describe the step-by-step approach for adding a new menu to the **Ensemble** application's menu bar. This menu will contain only a single command at this time; however, new commands will be added as the application's feature set grows.

1. The first step in creating a new menu requires that you click on AppMaker's **Select** menu, pull it down, and select the **Menus** choice, as shown in Figure 3-8.

2. AppMaker's selection window will show the **MainMenu** choice. Pick that choice by double-clicking on it in the window, as shown in Figure 3-9.

3. The default menu bar will appear on your screen. At this point, go to the **Edit** menu and choose the **Create Menu** command, as shown in Figure 3-10. This will create a new blank menu to the right of the **Edit** menu in the

Figure 3-8
Choosing **Menus** from AppMaker's **Select** menu

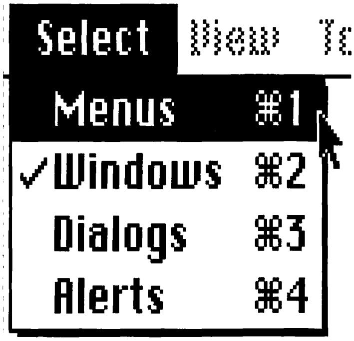


Figure 3-9
Picking the **MainMenu** bar in AppMaker's selection window

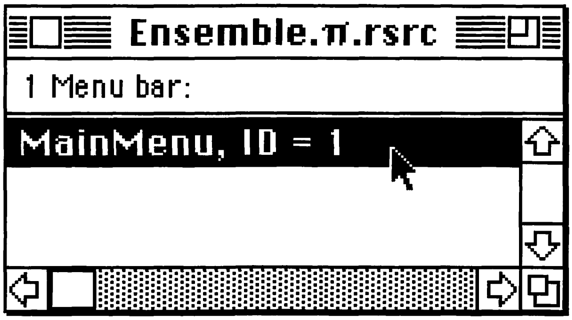
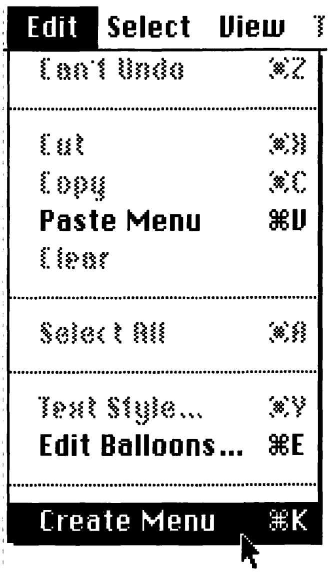


Figure 3-10
Select **Create Menu** from AppMaker's **Edit** menu



Ensemble application's **MainMenu** menu bar, as shown in Figure 3-11.

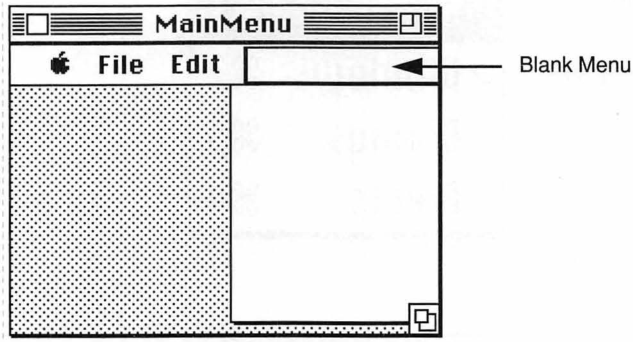


Figure 3-11
New blank menu

4. The next step is to click inside the rectangle at the top of the new blank menu, type the name **Format** and then enter a carriage return. AppMaker will create a rectangle for the first menu command in the **Format** menu. You'll notice that it has three compartments, containing, from left to right, the command name, the command key, and the command number. Enter the name **Notebook...** (note that the ellipsis '...' is formed either by typing three consecutive periods or by using the **Option-;** key combination), tab twice to skip the command key field, and type **2000** for the command number. The entire menu entry is shown in Figure 3-12. Type the **Enter** key to indicate that

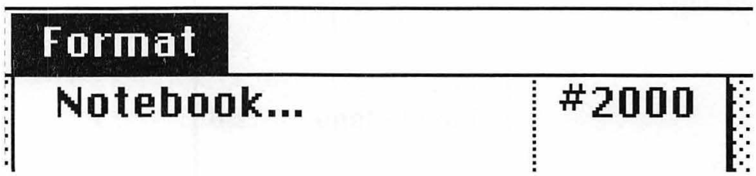


Figure 3-12
New **Format** menu, with **Notebook...** command

the menu is complete, and then click on the close box of the menu bar to dismiss the window.

Adding a New Menu Bar and Font Menu to Ensemble

The next series of steps creates a new menu bar for the Ensemble application. The purpose of this menu bar is simply to contain a **Font** menu that we will be using when adding new code to insert the user's installed fonts into the **Format Notebook** dialog box that will be described shortly.

1. Make sure that the MainMenu window is closed, but that Menus are still checked in the **Select** menu.

2. Pull down the **Edit** menu and select **Create Menu Bar**, as shown in Figure 3-13.

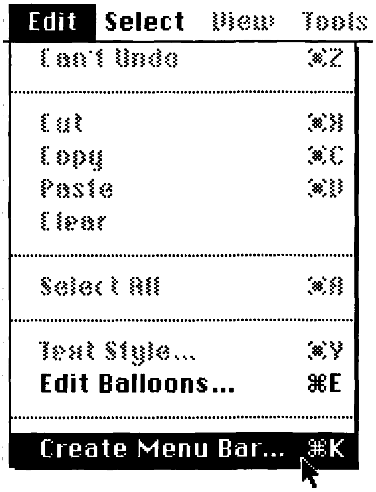


Figure 3-13
Create a new menu bar to hold the **Font** menu

3. After you release the mouse button, AppMaker will display a dialog in which you can name your new menu bar, as shown in Figure 3-14. The suggested new name is **SubMenus**. This is fine, so click OK in the dialog box. AppMaker will display a new blank menu bar, and the name **SubMenus** will be in the selection list.

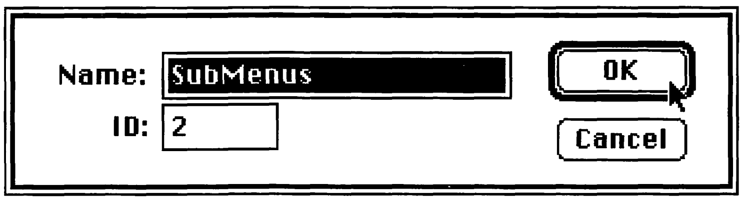


Figure 3-14
New **SubMenus** menu bar

4. Select **Create Menu** from the **Edit** menu at this time, as shown in Figure 3-15.

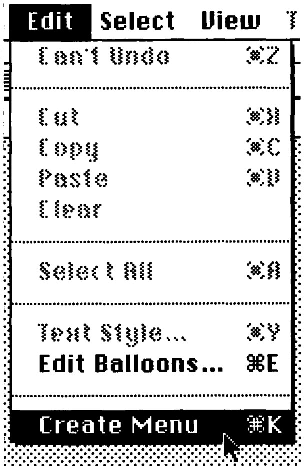


Figure 3-15
Create new menu to hold FONT information

5. Finally, type in the name **Font** at the top of the new menu, and enter the names **System** and **Application** into the menu, as shown in Figure 3-16. This will provide the first two entries for the new menu. AppMaker will automatically generate code to load the user's FONT names into this menu, as you will see in the next chapter.

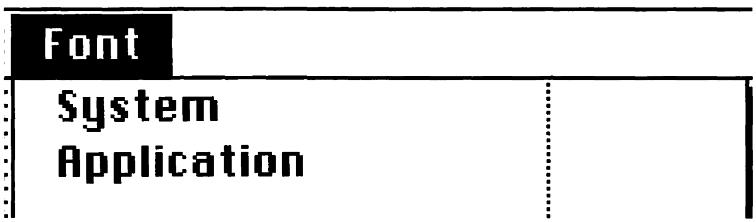


Figure 3-16
New **Font** menu and two initial entries

This concludes the operations necessary to create the **Font** menu in a new menu bar. This menu will never be displayed; however, the two initial entries and the user's font names that are added by AppMaker's generated code will be used to create a scrolling list of font names in the **Format Notebook** dialog box.

Adding a Dialog Box to Ensemble

The next series of steps involves the creation of a dialog box that will automatically open when the **Format Notebook** command is chosen. When additional dialog boxes are added, we will follow this same procedure.

1. The first step is to choose **Dialogs** from AppMaker's **Select** menu, as shown in Figure 3-17.

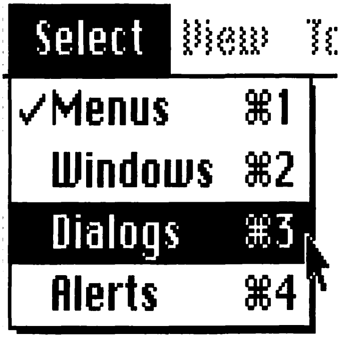


Figure 3-17 Choosing **Dialogs** from AppMaker's **Select** menu

2. You will notice that AppMaker's dialog selection list is empty at this point. This is because no dialogs have yet been defined. To create a new dialog, pull down the **Edit** menu and choose **Create Dialog**, as shown in Figure 3-18.

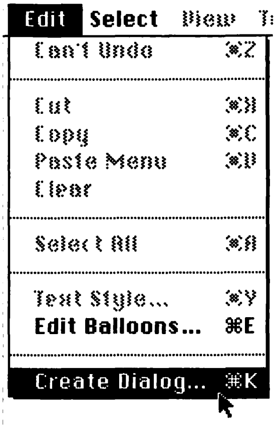


Figure 3-18 Choosing **Create Dialog** from AppMaker's **Edit** menu

3. AppMaker will display a screen that shows the various choices for types of dialog windows. Type in the information and select the standard plain dialog window, as shown in Figure 3-19.

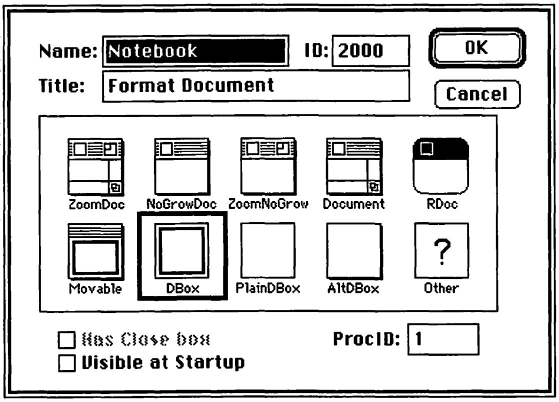


Figure 3-19
AppMaker's dialog information box

4. You should type the name **Notebook** for the name of the dialog and anything you want for the title, which is not displayed for a plain dialog window. The name **Notebook** is used by AppMaker to match against corresponding Menu commands, and because this name matches the command with that name in the **Format** menu (see Figure 3-12), AppMaker will generate code to open our **Notebook** dialog automatically when that menu command is chosen. When you have completed this step, click the **OK** button, and you'll see the **Notebook** entry appear in AppMaker's Dialog Selection List, as shown in Figure 3-20.

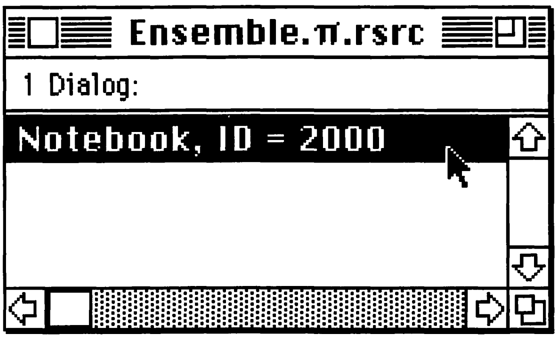


Figure 3-20
AppMaker's dialog selection list

5. Double-click on the **Notebook** entry in the selection list and you will see the empty **Notebook** dialog that is shown in Figure 3-21. In the next few steps, you will be filling in the contents of this dialog.

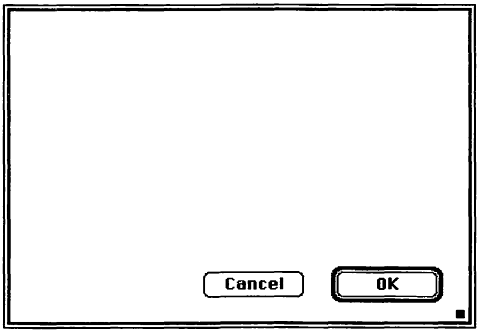


Figure 3-21
Empty **Notebook** dialog box

6. In order to help you visualize what you need to do to duplicate its appearance, the **Notebook** dialog is shown in its completed state in Figure 3-22.

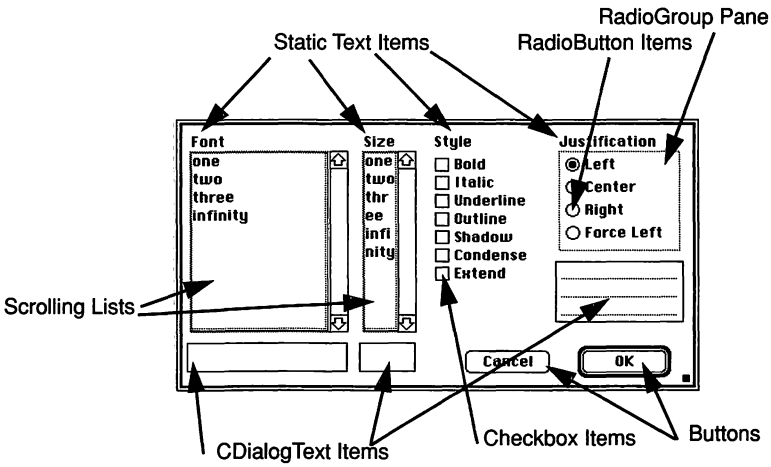


Figure 3-22
Notebook dialog with components annotated

7. Note that the dialog is created with default **OK** and **Cancel** buttons. The first step is to resize the dialog box to make it wide enough to contain all the items shown in the figure. You will also need to move the **OK** and **Cancel** buttons by clicking on them with the arrow tool and dragging them to appropriate positions in the dialog.

8. Creation of the scrolling lists consists of combining several interface components. This is best illustrated by the diagram in Figure 3-23.

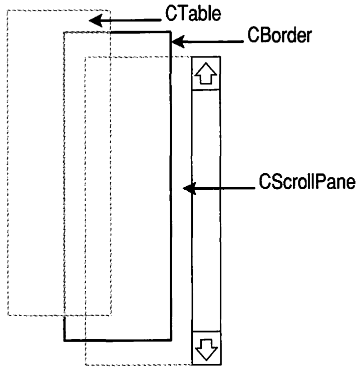


Figure 3-23
Construction of scroll pane

9. Notice that the scrolling list is created from three basic components: a **CScrollPane**, a **CBorder**, and a **CTable** component. If you choose **Show Tools as Text** from AppMaker's **View** menu, you will be able to choose these components by name. Start by choosing the **CScrollPane** tool, position the cursor (cross) at the top left edge of where you want to locate the pane, press the mouse button, and drag down and to the right to create the pane. When you release the mouse button, a pane with a vertical scroll bar will be shown.

10. The next step is to create the border that encloses the blank portion of the scroll pane. This is accomplished by choosing the **CBorder** tool, positioning the cursor (cross) at the upper-left corner of the ScrollPane, pressing the mouse button, and dragging down and to the right just until the border pane covers the blank portion of the scroll pane.

11. The final step is to create a **CTable** object that fits within the border. Select the **CTable** tool, click the cursor (cross) at the top left of the border pane, press the mouse button, and drag down and to the right so that the **CTable** pane fits within the border previously drawn. When you release the mouse button, you will see that AppMaker has written four entries into the pane, to show you the appearance of the completed table. These entries read: **one**, **two**, **three**, **infinity**.

12. Create another scrolling list by following steps 9–11. This will be the Font Size scrolling list. Create the **CScroll-Pane**, **CBorder**, and **CTable** components, as shown.

13. The next step is to create the checkboxes that will be used to set the text style. Because text styles are additive, multiple boxes can be checked simultaneously. The checkbox is the perfect interface element for this application. A checkbox is a user interface element that changes state from **on** to **off** each time it is clicked. The **on** status is shown as an **X** inside the box, whereas in the **off** state, the interior of the box is empty. To create a checkbox, select the **CCheckbox** tool and click the mouse button at the left edge of where you want the checkbox and its label to be positioned. After you release the mouse button, you will be able to type in the label for the checkbox. Although AppMaker allows you to specify the style of each of the text items you define, leave all the text in its default style.

14. The next interface item consists of two components: a **CRadioGroupPane** and multiple **CRadioControl** elements. A radio group pane groups a set of radio button elements, the distinguishing feature of which is that only one button in the group can be active. The active state is shown as a black dot inside the button, whereas when a radio button is **off**, the interior of the button is empty. The construction of this item is shown in Figure 3-24. You have to create the **CRadioGroupPane** first, by selecting that tool. Then, position the cursor at the top left corner of the pane, press the mouse button, and drag down and to the right to the bottom right corner that marks the extent of the group. The **CRadioControl** elements are placed inside the **CRadioGroup** pane.

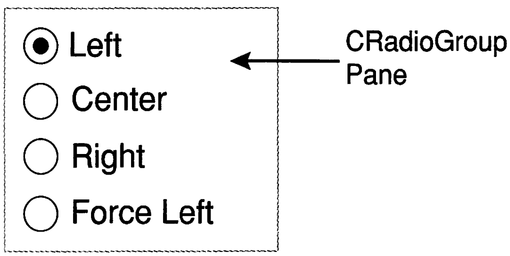


Figure 3-24
Construction of radio group

15. Create the **CRadioControl** elements by selecting that tool and then clicking the mouse button at the position where you wish the left edge of the button to be located. When you release the mouse button, you will be able to type in the text associated with the button. If you need to resize the **CRadioGroupPane** to accommodate the number of buttons, that is easily accomplished by selecting the **Arrow** tool, clicking on the pane's border, and then using the pane's size box to change the size of the pane.

16. Next, create the **CDialogText** elements by choosing that tool and creating a single-line element below each of the scrolling lists, and a third element below the **RadioGroupPane**. These will be used to show the chosen font, the chosen size, and a sample of the actual text, respectively.

17. The final step in creating the dialog box is to enter the **Static Text** items that identify the **Font**, **Size**, **Style**, and **Justification** elements in the dialog box.

18. This completes the modifications to the **Ensemble.π.rsrc** file at this time. Save the file and choose the **Generate** option from the **File** menu. You will see that all of the files whose names begin with the letter '**z**' will be regenerated, and four new files will have been added to the list: **Notebook.c**, **Notebook.h**, **zNotebook.c**, and **zNotebook.h**. These four files comprise the code that implements the **Notebook** dialog box that you have just created.

During the course of creating the **Notebook** dialog box interface elements, you may have to resize the dialog box or one or more of the elements. Feel free to do so, until everything looks correctly proportioned to your eye. You will be able to change any item at any time in the future, so don't worry about getting everything right the first time. You can also delete an element and recreate it at any time. This is one of the powerful features of AppMaker; it allows you to modify the interface appearance at any time. If you make changes by adding or deleting elements, make sure that you choose **Generate** from the **File** menu before you quit. Always save the results of modifying the **Ensemble.π.rsrc** file when an AppMaker session is complete.

Each time you generate files in AppMaker, you will discover that all of the files whose names begin with the letter '**z**' are regenerated. In addition, AppMaker will regenerate the **ResourceDefs.h** file, because this file contains definitions of the resource numbers for important user interface elements.

You'll notice that AppMaker never regenerates the files whose names do not begin with the letter **z**. In this particular case, we have defined a new dialog box, and therefore, two brand new files are generated. The subclass files called **Notebook.c** and **Notebook.h** will be generated this time, in addition to their superclass files **zNotebook.c** and **zNotebook.h**.

You can force AppMaker to regenerate a subclass file by deleting the file from the project folder. When AppMaker notices that it is missing, it will elect to regenerate the file.

Compiling the Generated Code

After the changes have been made to the **Ensemble.π.rsrc** file, and new source code files have been generated, you can launch THINK C to add these new source files to the project. To do this, double-click on the **Ensemble.π** project file and choose the **Add** command from THINK C's **Source** menu. Navigate to the project's folder if necessary, and add the **Notebook.c** and **zNotebook.c** files. (These should be the only entirely new files.) Now you can compile the new code. The best way to do this is to select **Make** from the **Source** menu. Click the **Use Disk** button, wait for THINK C to determine which files need to be recompiled, and then click on the **Make** button.

When the source files have been recompiled, select **Run** from the **Project** menu. This will display the debugger's windows, with the execution cursor positioned at the first instruction in the **main** function. Click the **Go** button in the debugger control window. The **MainWindow** will be displayed. If you click the mouse inside the window, you will be able to type text in a default font. You can also zoom and resize the window. If you type more text than will fit inside the window, the vertical scroll bar will become active and you will be able to scroll through the text. The standard Macintosh **Cut** and **Paste** commands will also be active for this window. Try cutting or copying some text to the clipboard and then pasting it back into the window. The text-editing features, with the exception of font, size, and style selection, are now complete.

You will also see that a **Format** menu has been added to the menu bar, and pulling down that menu will display the **Notebook** command. Choose this command now. The **Notebook** dialog box should be displayed on the screen, just as it was defined in AppMaker. Note that the scrolling lists include entries with the words **one**, **two**, **three**, and **infinity**. This is the result of code that AppMaker has automatically generated and that we will modify for our own purposes in Chapter 5. If you click on one of the scrolling list entries, it should become highlighted. Clicking below all the entries should remove any existing highlight.

If you click in the checkbox elements, the check marks will appear and disappear with multiple clicks. The radio button elements will work as a group, allowing only one member of the group to be selected at a time.

It should be apparent that quite a bit of the code to operate the application has already been automatically generated. This is the real value of the AppMaker and THINK C combination. Between AppMaker's generated code and the features of the THINK Class Library, most of the hard work has already been accomplished.

The next chapter describes portions of the newly generated code, accenting the code that implements the new text-editing features added to the Ensemble application.

Exercises

1. Describe the purpose of the scroll pane, border, and text-editing user interface elements. Indicate how these interact during the course of the application's execution.

2. Use AppMaker's **Text Style** dialog to change the style of the text associated with each of the checkboxes in the **Notebook** dialog. The box titled **Bold** should be displayed in a boldface style; the one titled **Italic** should be displayed in an italic style, and so forth.

3. Explain the difference in functionality between the check-box and radio button interface elements. When is the use of one preferred over the other?

4. Why are the radio buttons placed inside a radio group element in the **Notebook** dialog? What would be the effect if the radio group were not present. How would the radio buttons react to mouse clicks?

5. Consider the consequences of allowing the user to enter a font name or font size into the corresponding dialog text fields. What should be done, if anything, to protect against the entry of a non-existent font name or font size? Explain.

6. The text-editing pane could have been created as a **CStyleText** element instead of a **CEditText** element. Describe the difference in functionality between these two element types. (*Hint:* Refer to the description of the **CEditText** class in the *THINK C Object-Oriented Programming Manual*, and the source code for the **CStyleText** class in the *Text Classes* folder within the *THINK Class Library 1.1* folder in the THINK C version 5.0 product.)

7. Design and implement the text-editing pane as a **CStyleText** element. Make provisions for the pane to include text in multiple fonts, sizes, and styles.[1]

---

1. Creating and supporting the text-editing feature as a **CStyleText** element is a rather large undertaking. Covering this topic as part of an advanced course in object-oriented software development is highly recommended.

Chapter 4

Examining the EditText Code

Let's take a moment to summarize the accomplishments described in the previous chapter. The Ensemble application's default resource file was initially created by AppMaker, as described in Chapter 2. Beginning with that file, according to the descriptions in Chapter 3, the following functions have been added:

❖ ScrollPane and EditText panes to the **MainWindow** window generated by AppMaker.

❖ A new **Format** menu, with a command named **Notebook**.

❖ A new **SubMenus** menu bar with a **Font** menu containing two initial entries; **System** and **Application**, referring to the System and Application fonts, respectively.

❖ A **Notebook** dialog box with static text, scroll pane, text editing, checkbox, radio group, and radio button elements.

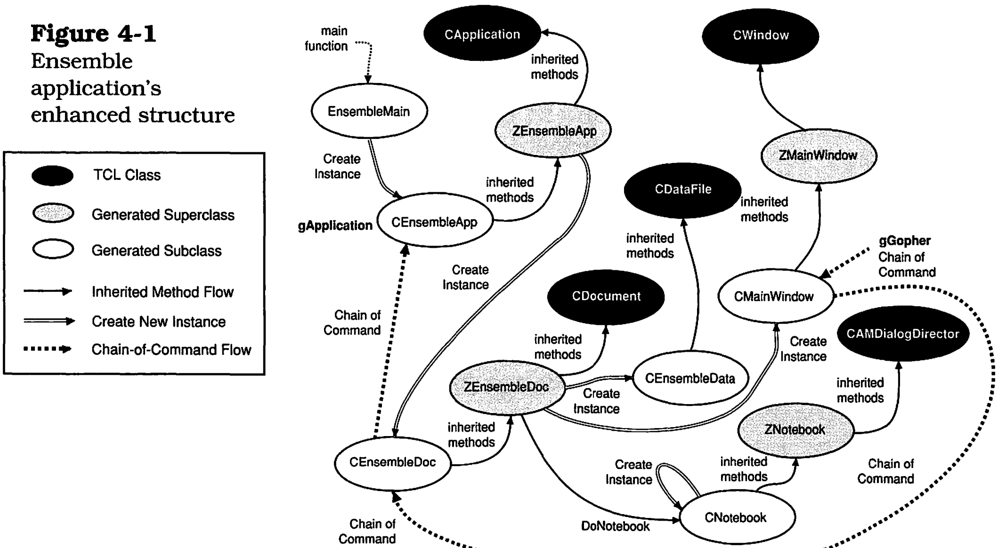
❖ Generated code for the newly added features. This code consists of new versions of the **zEnsembleApp**, **zEnsembleDoc**, and **zMainWindow** files, and the new **zNotebook** and **Notebook** files. The **ResourceDefs.h** file has also been regenerated.

All of the new interface elements are implemented in code within the files mentioned. As previously indicated, a file whose name begins with a character other than '**z**' is never regenerated. The **Notebook.c** and **Notebook.h** files *were* generated this time, because they are completely new files. On subsequent invocations of AppMaker, these files will not be automatically regenerated.

The EditText Code Structure

The best way to see how the new code differs from AppMaker's default-generated code is to compare the diagram in Figure 4-1 with the diagram for the default code shown in Figure 2-1 on page 24.

Figure 4-1
Ensemble application's enhanced structure



Notice that the new **CNotebook** and **ZNotebook** classes are now attached to the **ZEnsembleDoc** class. This is by virtue of the new code generated into the **ZEnsembleDoc**'s **DoCommand** method. In addition, because the **MainWindow** has a pane that will accept events, the **gGopher** and the chain of command extend to the **CMainWindow** instance, as shown in the diagram. The **CNotebook** instance is created by the **DoNotebook** function that is embedded in the generated code for the **CNotebook** module. When the **Notebook** dialog is open, the **gGopher** variable will point to the **CNotebook** instance. In this case, the **DoCommand** method of the **Notebook** dialog will be the first to receive any commands sent to the **gGopher**.

What might not be apparent from looking at Figure 4-1 is that quite a bit of new code has been added to several of the '**z**' file classes. In order to put the newly generated code into perspective, Table 4-1 has been prepared to show the classes in which new code has been generated, the specific methods that have been enhanced, and the nature of the enhancements. Later sections will discuss the details of the generated code.

Table 4-1
Generated code for the new EditText pane and new **Format** menu

Class	Method	Description
ZEnsembleApp	SetUpMenus	Includes code to create the FONT menu and add all the font names
ZEnsembleDoc	DoCommand	Adds code to handle the **Format Notebook** command
ZMainWindow	IZMainWindow	Additional code to create and install the CScrollPane and CEditText panes
ZNotebook	IZNotebook	Code to create all the elements of the **Format Notebook** dialog
CNotebook	DoNotebook	Not really a method of this class, but a global function that operates the **Notebook** dialog

The sections that follow in this chapter describe the new code. Chapter 5 shows the details of the custom code, added to these classes, that fully implements the text-formatting features of the Ensemble application.

Newly Generated Code in ZEnsembleApp

In the new version of the generated code, the **ZEnsembleApp** module has been updated to include additional functionality. In general, only one of the original methods has been updated with new code. The remaining methods in the **ZEnsembleApp** module are unchanged.

SetUpMenus Method Code

The **ZEnsembleApp** module's **SetUpMenus** method has been enhanced to include code to create the Font submenu, and also automatically add the names of all the FONT resources by generating the **AddResMenu** code. The newly generated code is as follows:

```
void ZEnsembleApp::SetUpMenus(void)
{
   MenuHandl macMenu;

   inherited::SetUpMenus ();
   macMenu = GetMenu (5);   // Font menu
   FailNILRes (macMenu);
   AddResMenu (macMenu, 'FONT');
}
```

The default version of this method contained only the call to the inherited **SetUpMenus** method, which is responsible for setting up the standard **MainMenu** menu bar and its menus. The new version adds the code to read in the menu (5) resource.

The names of the user's list of installed fonts are added to the new menu using the Mac Toolbox's **AddResMenu** function, which adds the names of all the open font resources to the menu, in alphabetical order. This is the approved method for obtaining a list of the installed fonts.

The new menu won't be used as such, but its list of fonts will be invaluable when the **Format Notebook** dialog box is created.

Newly Generated Code in ZEnsembleDoc

The next significant change to the generated code, as shown in Table 4-1, is the additional code in the **ZEnsembleDoc** class's **DoCommand** method. The new code for **DoCommand** is as follows:

DoCommand *method code (beginning)*

```
void  ZEnsembleDoc::DoCommand  (long  theCommand)
{
   switch (theCommand)
   {
      case cmdNotebook:
      {
         DoNotebook(this);
         break;
      }
      default:
      {
         inherited::DoCommand (theCommand);
```

DoCommand
method code
(concluded)

```
            break;
        }
    }
}
```

Note that the **DoCommand** method handles only the **Notebook** command from the **Format** menu. The generated code calls a global function **DoNotebook**, which is contained in the **Notebook.c** file. The single parameter passed to the **DoNotebook** function is a handle to the current object (**CEnsembleDoc**). If any other command is chosen, the **DoCommand** method calls its inherited method in the TCL's **CDocument** class.

Newly Generated Code in ZMainWindow

The next new addition to the generated code can be found in the **IZMainWindow** method in the **ZMainWindow.c** file. It's easy to see how the new code, shown on page 71, differs from the code in the default version, shown on page 35. Specifically, new code has been generated to install and initialize the **CScrollPane** and **CAMEditText** elements in the window.

AppMaker can take care of creating these elements, because it has all the information it needs to do so. The resources that define the **CScrollPane** and **CAMEditText** elements are found in the **Ensemble.π.rsrc** file, from which AppMaker can generate the appropriate code.

AppMaker's general approach to adding new user interface elements is to generate a TCL-compatible resource containing the parameters associated with the element, and then use the **IViewRes** method inherited from the **CView** class in the TCL to initialize the newly created element. Since most elements have a few parameters that aren't visible in the current version of AppMaker, it is possible to "tune" the element's appearance or behavior by using **ResEdit** or other resource-editing applications. The code for the new **IZMainWindow** method is as follows:

IZMainWindow
method code
(beginning)

```
void  ZMainWindow::IZMainWindow  (CDirector  *aSupervisor)
{
    CView   *enclosure;
    CBureaucrat  *supervisor;
```

IZMainWindow *method code (concluded)*

```
	CSizeBox	*aSizeBox;
	IWindow (MainWindowID, FALSE, gDesktop, aSupervisor);
	itsMainPane = NULL;

	enclosure = this;
	supervisor = this;

	ScrollPane1 = new CScrollPane;
	ScrollPane1->IViewRes ('ScPn', 131, enclosure, supervisor);

		Field3 = new CAMEditText;
		Field3->IViewRes ('AETx', 133, ScrollPane1, supervisor);

	ScrollPane1->InstallPanorama (Field3);

	aSizeBox = new CSizeBox;
	aSizeBox->ISizeBox (enclosure, supervisor);

}
```

Note that in the first (default) version of this method (shown on page 35), AppMaker only generated code to call the inherited **IWindow** method and create the **CSizeBox** instance.

In the new version of the **IZMainWindow** method, AppMaker has generated code to create the **CScrollPane** (ScrollPane1) and **CAMEditText** elements and install the **CAMEditText** (Field3) element as the panorama for the **CScrollPane**. This code is all that is needed to allow you to type into the **MainWindow** pane, using the default system font.

Newly Generated Code in ZNotebook

The **ZNotebook** module (and its companion **CNotebook** module) implements the content and user interface functions of the **Format Notebook** dialog. The **ZNotebook** superclass contains four methods, each of which will be fully described in this section.

IZNotebook Method Code

The **IZNotebook** method is responsible for creating each of the user interface elements in the **Format Notebook** dialog and for initializing those elements. Each element is installed into the window by creating an instance of its associated class and then calling the appropriate **IViewRes** method to set up its appearance and behavior. The code, which is quite long, is as follows:

*Beginning of **IZNotebook** method to create and initialize the user interface elements in the **Format Notebook** dialog*

```
void  ZNotebook::IZNotebook  (CDirectorOwner  *aSupervisor)
{
  CView  *enclosure;
  CBureaucrat  *supervisor;

  inherited::IAMDialogDirector (NotebookID, aSupervisor);

  enclosure = itsWindow;
  supervisor = itsWindow;

  OKButton = new CAMButton;
  OKButton->IViewRes ('CtlP', 128, enclosure, supervisor);

  CancelButton = new CAMButton;
  CancelButton->IViewRes ('CtlP', 129, enclosure, supervisor);

  FontLabel = new CAMStaticText;
  FontLabel->IViewRes ('AETx', 129, enclosure, supervisor);

  SizeLabel = new CAMStaticText;
  SizeLabel->IViewRes ('AETx', 130, enclosure, supervisor);
  StyleLabel = new CAMStaticText;
  StyleLabel->IViewRes ('AETx', 131, enclosure, supervisor);

  BoldCheck = new CAMCheckBox;
  BoldCheck->IViewRes ('CtlP', 132, enclosure, supervisor);

  ItalicCheck = new CAMCheckBox;
  ItalicCheck->IViewRes ('CtlP', 133, enclosure, supervisor);

  UnderlineCheck = new CAMCheckBox;
  UnderlineCheck->IViewRes ('CtlP', 134, enclosure, supervisor);

  OutlineCheck = new CAMCheckBox;
  OutlineCheck->IViewRes ('CtlP', 135, enclosure, supervisor);

  ShadowCheck = new CAMCheckBox;
  ShadowCheck->IViewRes ('CtlP', 136, enclosure, supervisor);

  CondenseCheck = new CAMCheckBox;
  CondenseCheck->IViewRes ('CtlP', 137, enclosure, supervisor);

  ExtendCheck = new CAMCheckBox;
  ExtendCheck->IViewRes ('CtlP', 138, enclosure, supervisor);
  JustificationLabel = new CAMStaticText;
```

IZNotebook *method (condluded)*

```
    JustificationLabel->IViewRes ('AETx', 132, enclosure, supervisor);

    Field14 = new CAMDialogText;
    Field14->IViewRes ('ADTx', 128, enclosure, supervisor);

    Field15 = new CAMDialogText;
    Field15->IViewRes ('ADTx', 129, enclosure, supervisor);

    Field16 = new CAMDialogText;
    Field16->IViewRes ('ADTx', 130, enclosure, supervisor);

    Group17 = new CRadioGroupPane;
    Group17->IViewRes ('Pane', 128, enclosure, supervisor);
    CenterRadio = new CAMRadioControl;
    CenterRadio->IViewRes ('CtlP', 140, Group17, Group17);
    RightRadio = new CAMRadioControl;
    RightRadio->IViewRes ('CtlP', 141, Group17, Group17);
    ForceLeftRadio = new CAMRadioControl;
    ForceLeftRadio->IViewRes ('CtlP', 143, Group17, Group17);
    LeftRadio = new CAMRadioControl;
    LeftRadio->IViewRes ('CtlP', 139, Group17, Group17);

    ScrollPane22 = new CScrollPane;
    ScrollPane22->IViewRes ('ScPn', 132, enclosure, supervisor);

        Rect24 = new CAMBorder;
        Rect24->IViewRes ('Bord', 130, ScrollPane22, supervisor);

        List25 = NewList25 ();
        List25->IViewRes ('ATbl', 134, Rect24, supervisor);

    ScrollPane22->InstallPanorama (List25);

    ScrollPane26 = new CScrollPane;
    ScrollPane26->IViewRes ('ScPn', 133, enclosure, supervisor);

        Rect28 = new CAMBorder;
        Rect28->IViewRes ('Bord', 131, ScrollPane26, supervisor);

        List29 = NewList29 ();
        List29->IViewRes ('ATbl', 135, Rect28, supervisor);

    ScrollPane26->InstallPanorama (List29);

}
```

As you can see, there is a great deal of generated code to implement the appearance of the **Format Notebook** dialog. The code performs the following actions:

1. The **OKButton** is the first to be created. AppMaker will automatically create standard **OK** and **Cancel** buttons in every new dialog. The **OK** button is the element that will be given the bold outline when the dialog is first shown.

2. The **CancelButton** is the next to be defined and initialized. This button does not have a bold outline.

3. The next three elements are **CAMStaticText** elements called **FontLabel**, **SizeLabel**, and **StyleLabel**. They appear in this order only because they were defined in that order.

4. The next series of elements conprises instances of **CAMCheckbox**, which implements a standard Macintosh checkbox function. The TCL takes care of automatically drawing the '**X**' in the box when it's selected and clearing the '**X**' when it's deselected. The **CAMCheckbox** elements are named according to their labels: **BoldCheck**, **ItalicCheck**, **UnderlineCheck**, **OutlineCheck**, **ShadowCheck**, **CondenseCheck**, and **ExtendCheck**.

5. The **JustificationLabel CAMStaticText** element is the next element to be defined. When the dialog box was created, that label was added after the width of the checkbox elements had been determined.

6. The next three elements are the **CAMDialogText** items, corresponding to the blank boxes in the dialog. The various elements of the dialog are shown in Figure 3-22, on page 59, where they are referred to as **EditText** items. They are named **Field14**, **Field15**, and **Field16** and will eventually hold the selected font name, selected font size, and a sample of the font in the selected size, style, and justification, respectively.

7. The next element, called **Group17**, is a **CRadioGroupPane** instance, which holds and manages the text justification radio buttons.

8. The individual text justification **CAMRadioControl** instances are named **CenterRadio**, **ForceLeftRadio**, **RightRadio**, and **LeftRadio**, to correspond to their respective labels. Each of these is created and initialized by passing the **IViewRes** method its supervisor, the **Group17 CRadioGroupPane** element.

9. The final two elements are the scrolling lists that will hold the font names and font sizes when the dialog is fully initialized. Each of the scrolling lists is built as shown in Figure 3-23 on page 60. An instance of **CScrollPane** is created, an instance of **CAMBorder** is placed inside of the scroll pane, and then a new list is created and placed inside the border. When the elements have all been created, the list is installed as the panorama for the composite pane.

The two scrolling lists are very similar; their only fundamental difference is the use of the **NewList25** and **NewList29** methods to create the list instances. The rationale for providing custom methods for the lists is discussed in the following section.

NewList25 Method Code

The **IZNotebook** method calls the **NewList25** method to create the font name list.

The code to implement this (along with AppMaker's comments) is as follows:

```
// The only purpose of this function is so that you can override it
// to create the list as your subclass of CAMTable

CAMTable   *ZNotebook::NewList25   (void)
{
   CAMTable   *theList;

   theList = new CAMTable;
   return (theList);

}
```

As indicated by the generated comment, AppMaker expects you to use an override method in the **CNotebook** module to fully create and initialize the list instance. In fact, AppMaker even generates an override method in the **CNotebook** class, as will shortly be shown.

NewList29 Method Code

As with the **NewList25** code, AppMaker generates a method to create an instance of the font size list. AppMaker expects you to use an override method to fully create and initialize the list instance. The code generated into the **ZNotebook** module for the **NewList29** method is as follows:

```
CAMTable   *ZNotebook::NewList29   (void)
{
   CAMTable   *theList;

   theList = new CAMTable;
   return (theList);

}
```

UpdateMenus Method Code

The **ZNotebook** class also contains an override of the inherited **CAMDialogDirector** class's **UpdateMenus** method, merely to provide a method for the **CNotebook** class to override. The code that implements this method is as follows:

```
void   ZNotebook::UpdateMenus   (void)
{
   inherited::UpdateMenus ();
}
```

As is apparent, the generated code merely calls the inherited method.

Newly Generated Code in CNotebook

Quite a few methods are provided in the AppMaker-generated **CNotebook** module. In addition, the **DoNotebook** global function is also generated into this module. This is the function that the code in the **DoCommand** method of the **ZEnsembleDoc** class calls to initiate the opening of the **Format Notebook** dialog. The next few sections discuss the code that was generated into the **CNotebook.c** file. In order to provide the dialog with full functionality, additional code, described in Chapter 5, will be provided.

DoNotebook Function Code

The generated code for the global **DoNotebook** function is found in the **CNotebook** module. The function is global so that the dialog can be called from any module in the application, not just the **ZEnsembleDoc** module. The code for the function is as follows:

```
void DoNotebook   (CDirectorOwner *aSupervisor)
{
   CNotebook   *dialog;
   long         dismisser;

   dialog = NULL;
   TRY
   {
      dialog = new CNotebook;
      dialog->INotebook (aSupervisor);

      /* initialize dialog panes */
      dialog->BeginDialog ();
      dismisser = dialog->DoModalDialog (cmdOK);
      if (dismisser == cmdOK)
      {
         /* extract values from dialog panes */
      }
      dialog->Dispose ();
   }
   CATCH
   {
      ForgetObject (dialog);
   }
   ENDTRY;
}
```

The generated code for **DoNotebook** contains an exception handling mechanism that is a new feature in THINK C version 5.0. The TRY and CATCH keywords are used, respectively, to introduce code that might fail when the enclosed code is executing and to specify the exception-handling procedure to use if that situation occurs. The ENDTRY keyword delimits the end of the exception-handling code. If an error occurs (such as the inability to allocate memory for a new instance of the **CNotebook** class), the CATCH code will receive control, dispose of the dialog object, and propagate the failure condition up the exception handler stack, which eventually will display an appropriate alert to the user. It is possible to customize the CATCH handler to display its own alert, with information that is pertinent to the current application context.

AppMaker has generated comments in the code for the **DoNotebook** function to indicate where to place additional custom code to initialize the dialog and also where to extract the results of the user's actions when the dialog is dismissed by clicking the **OK** button.

INotebook Method Code

The initialization code for the **CNotebook** instance created in the **DoNotebook** function is as follows:

```
void   CNotebook::INotebook   (CDirectorOwner   *aSupervisor)
{
   inherited::IZNotebook (aSupervisor);
}
```

Basically, this code calls the inherited **IZNotebook** method, which creates instances of all the dialog interface elements and initializes their default appearances. The purpose of such an override method is to provide a place to perform additional initialization of the dialog's interface elements. **IZNotebook** will later be enhanced with custom code additions.

CList25 IViewTemp Method Code

The code for this method is as follow:

```
void   CList25::IViewTemp(CView          *anEnclosure,
                          CBureaucrat    *aSupervisor,
                          Ptr            viewData)
{
   inherited::IViewTemp (anEnclosure, aSupervisor, viewData);

   // any additional initialization for your subclass
   AddRow (4, 0);    // e.g., add 4 rows at the beginning of the list
}
```

AppMaker generates code in the **CNotebook** subclass to override the **ZNotebook** class's **IViewTemp** method for creating the scrolling lists. In this case, AppMaker also generates a single line of code that adds four rows to the beginning of the list, just to indicate how adding rows is done. This method will be modified to perform the appropriate initialization of **CList25**, using the font names from the **Font** menu constructed within AppMaker.

CList25 GetCellText Method Code

The standard method that AppMaker uses in its generated code for user interface elements of the **CTable** class is to override the **CTable** class's **GetCellText** method. This override provides a very simple method of supplying the text for table cells.

The **GetCellText** method is called with three arguments. The first argument contains the cell (column and row) that **CTable** requires; the second provides the width of the cell, for situations where you want to provide special clipping of the cell's contents; and the third is a pointer to a **Str255** variable, in which the text for that cell is to be stored. The function of the **GetCellText** method is to provide the text for the cell.

Notice that the **CTable** class in the TCL keeps track of which cells have text that needs to be updated. This is a perfect example of the power of the THINK Class Library providing most of the functionality of an interface element. Given the cell whose text is required, you need only supply the code that provides it, as as follows:

***GetCellText** method code (beginning)*

```
void	CList25::GetCellText	(Cell	aCell,
					short	availableWidth, StringPtr	itsText)
{
	// replace with your own code which uses the cell coordinates to access
	// your private data structures, then convert the cell data to a Str255.
	switch (aCell.v) {
		case 0:
				CopyPString ("\pOne", itsText);
			break;
		case 1:
				CopyPString ("\pTwo", itsText);
			break;
		case 2:
				CopyPString ("\pThree", itsText);
```

GetCellText method code (concluded)

```
            break;
        default:
            CopyPString ("\pInfinity", itsText);
            break;
    }
}
```

The default-generated code uses the cell's row (**aCell.v**) value to determine which of the four messages to copy to the **itsText** string. Notice that cell rows (**aCell.v**) and columns (**aCell.h**) are zero based. That is, their values begin with 0, instead of 1. In the section of this chapter that discusses customizing the **GetCellText** code, all of this code will be customized to store the font names into the table's cells.

CNotebook NewList25 Method Code

The **ZNotebook** superclass defined a method called **NewList25**, shown on page 76. This method provides the opportunity to override **NewList25**'s functionality in the **CNotebook** subclass, to provide a different type of list, or to add new functionality to the list. The **NewList25** code will not need to be changed:

```
CAMTable   *CNotebook::NewList25   (void)
{
    CList25   *theList;
    theList = new CList25;
    return (theList);
}
```

CList29 Class Methods

The **CList29** class is generated for the font size table, and the default initialization code generated for it is nearly identical to the corresponding **IViewTemp**, **GetCellText**, and **New-List$_n$** code for the **CList25** class.

In general, AppMaker will generate a similar new class and corresponding methods for each list element defined in the user interface. Each such class will contain an **IViewRes** method for its initialization, a **GetCellText** method to provide the contents of each cell to the default **DrawCell** method inherited from the **CTable** class in the TCL, and finally, a **New-**

List$_n$ method that creates the instance of the list. We will not be showing duplicates of these classes and methods in this book, except when they need to contain unique custom code.

CNotebook UpdateMenus Method Code

AppMaker also generates an **UpdateMenus** method, as an override of the same method generated in the **ZNotebook** superclass. The method is intended for situations where menu commands need to be enabled or disabled, depending on the status of the dialog. The default-generated code is as follows:

```
void  CNotebook::UpdateMenus  (void)
{
   inherited::UpdateMenus ();
}
```

The code for the **UpdateMenus** method will not need to be modified.

CNotebook DoCommand Method Code

The generated code for the **CNotebook** class also contains code for a **DoCommand** method. When the **Format Notebook** dialog is active, the **gGopher** global variable points to the dialog. Whenever an event occurs while the dialog is active, its **DoCommand** method will be called. The code for the **DoCommand** method is as follows:

DoCommand *method code (beginning)*

```
void  CNotebook::DoCommand  (long  theCommand)
{
   switch (theCommand) {
      case cmdBoldCheck:
            /* DoBoldCheck ();*/
         break;
      case cmdItalicCheck:
            /* DoItalicCheck ();*/
         break;
      case cmdUnderlineCheck:
            /* DoUnderlineCheck ();*/
         break;
      case cmdOutlineCheck:
            /* DoOutlineCheck ();*/
         break;
      case cmdShadowCheck:
            /* DoShadowCheck ();*/
```

DoCommand method code (concluded)

```
        break;
    case cmdCondenseCheck:
        /* DoCondenseCheck ();*/
        break;
    case cmdExtendCheck:
        /* DoExtendCheck ();*/
        break;
    case cmdCenterRadio:
        /* DoCenterRadio ();*/
        break;
    case cmdRightRadio:
        /* DoRightRadio ();*/
        break;
    case cmdForceLeftRadio:
        /* DoForceLeftRadio ();*/
        break;
    case cmdLeftRadio:
        /* DoLeftRadio ();*/
        break;

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

The **DoCommand** method is called with the command number for the interface element associated with the command. It is important to note that AppMaker generates *click commands* for all of the checkboxes and radio buttons in the **Format Notebook** dialog.

AppMaker also generates comments in the code which recommend that you supply methods that handle the various types of commands (e.g., **DoBoldCheck( )**). The custom code will be added directly to the **DoCommand** method's individual cases. This is appropriate because each command will only require the addition of a single statement. Creating separate methods or functions for the purpose would be inefficient.

CNotebook ProviderChanged Method Code

The final method generated into the **CNotebook** class is called **ProviderChanged**. This is a very powerful method that is part of the TCL's *provider* and *dependent* notification methodology.

There are many cases in which it is important to indicate to an instance that the user has modified an interface element associated with a different class. For example, if you had an application that computed conversions in inches, picas, points, or ciceros, it would be important to know that the user had typed a value into one of these fields and to be able to automatically update the others to reflect the change.

The TCL implements a class called **CCollaborator** which contains methods that allow you to add *providers* and *dependents*, such that when a provider class senses a change, it can broadcast the nature of the change, and its dependents can determine what to do in each instance.

The TCL's **CBureaucrat** class directly overrides the **CCollaborator's BroadcastChange** method and sends a **ProviderChanged** message to the supervisor of the class in which the change occurred.

Each of the user interface items in the dialog is supervised by the **CNotebook** class. When the user types into any of the dialog text panes, a **BroadcastChange** message is sent by the **DoKeyDown** method inherited by the **CAMDialogText** instance from the **CDialogText** class.

In a similar fashion, the **SetValue** method inherited from the **CControl** class sends a **BroadcastChange** message when the state of a control (checkbox or radio button) is changed, and finally, the **SelectRect** and **DeselectRect** methods of the **CTable** class send the **BroadcastChange** message when a table entry is selected or deselected, respectively. Other classes in the TCL also send the **BroadcastChange** message; however, the ones previously mentioned pertain directly to the classes whose elements appear in the **Format Notebook** dialog.

When the user clicks on one of the selections in the font name list, for example, the **SelectRect** method inherited from the **CTable** class sends a **BroadcastChange** message that includes the argument **"tableSelectionChanged"**, which is intercepted by the **BroadcastChange** method override in the **CBureaucrat** class. This method in turn sends a **ProviderChanged** message to the table's supervisor, which is an instance of our **CNotebook** class. The **CBureaucrat** class also passes on the **BroadcastChange** message to the **CCol-**

laborator class, which sends a **ProviderChanged** message to any of the registered dependents of the provider class. In our case, there are no registered dependents.

The **ProviderChanged** method in the **CNotebook** class will catch the changes made to selections in the scrolling lists, as well as keystrokes entered into any of the text panes. The default-generated code for the **ProviderChanged** method merely identifies which user interface element is affected by the change and leaves it up to us to provide appropriate code to respond to the change. This method will be customized to handle new list element selections and data typed into the text panes. The default-generated code for the **ProviderChanged** method is as follows:

ProviderChanged *method code (beginning)*

```
void  CNotebook::ProviderChanged  (CCollaborator  *aProvider,
                                    long  reason,
                                   void*  info)
{
   if (aProvider == Field14) {
      if (Field14->GetLength () == 0) {
         // text is empty
      } else {
         // there is some text
      }
   }
   if (aProvider == Field15) {
      if (Field15->GetLength () == 0) {
         // text is empty
      } else {
         // there is some text
      }
   }
   if (aProvider == Field16) {
      if (Field16->GetLength () == 0) {
         // text is empty
      } else {
         // there is some text
      }
   }
   if (aProvider == List25) {
      if (List25->HasSelection ()) {
         // perhaps activate some buttons
      } else {
         // perhaps deactivate
      }
   }
```

ProviderChanged *method code (concluded)*

```
    if (aProvider == List29)
    {
        if (List29->HasSelection ()) {
            // perhaps activate some buttons
        } else {
            // perhaps deactivate
        }
    }
}
```

AppMaker generates code for the text fields that determines whether the fields contain data or are empty. In the case of the list panes, AppMaker generates code that determines whether a selection has been made or whether the click that caused the method to be called has deselected all elements in the list. The comments indicate the type of actions your code might perform when the various events occur.

Recap of the Generated Code

As previously mentioned, AppMaker will generate new versions of all the superclass modules (the ones whose names begin with the letter '**z**'), and it will also generate both superclass and subclass modules for new windows and dialogs. The **zNotebook** and **Notebook** modules are examples of this code-generation philosophy.

When a new menu command is added, code will be added to the **DoCommand** method of the document's superclass (e.g., **zEnsembleDoc**) to recognize the new command. If a dialog has the same name as a menu command, then code will be generated to invoke the dialog (e.g., **DoNotebook**) in the **DoCommand** method.

The next chapter discusses how the generated code presented in this chapter can be customized to fully implement the **EditText** features of the **Ensemble** application.

Exercises

1. Assuming that Figure 4-1 shows the universe of objects that exist at the time the illustration was drawn, how many actual object instances are represented in the dia-

gram? Why are some of the classes not "real" object instances?

2. Explain how the "chain of command" operates and what happens to commands and keystroke events that are relevant to the "chain of command."

3. Explain the operation of the **DoCommand** method in the **ZEnsembleDoc** class. When and for what purpose does this method execute?

4. Describe the interaction of the CScrollPane and CAMEditText class methods. In what way are these interrelated in our application? (*Hint*: Look up the description of the **CPanorama** class in the *Object-Oriented Programming Manual* for THINK C.)

5. The **IZNotebook** method illustrates a very flexible feature of AppMaker's approach to code generation. What is significant about the code in **IZNotebook**?

6. Describe the meaning of "collaboration," as this term is used in the THINK Class Library, and how our application benefits from its use.

7. Describe the purpose and operation of the **ProviderChanged** method in the **CNotebook** class.

Chapter 5

Customizing the EditText Code

Table 5-1 shows the classes and methods that will be customized and described in this chapter.

Table 5-1
Customized methods to fully implement **EditText** features

Class	Method	Description
CEnsembleApp	SetUpMenus	Adds **Font** menu
CEnsembleDoc	IEnsembleDoc	Creates CFontData instance
CEnsembleDoc	NewFile	Overrides the NewFile code
CEnsembleDoc	InitTextFormat	Initializes the **EditText** pane
CEnsembleDoc	OpenFile	Sets handle to file data into **EditText** pane
CEnsembleDoc	DoCommand	Invokes **DoNotebook** dialog
CEnsembleData	IEnsembleData	Initializes **itsEditTextData** handle
CEnsembleData	ReadData	Reads **itsTextData** from file
CEnsembleData	WriteData	Gets **EditText** handle and write out its contents
CEnsembleData	DoRevert	Disposes of current data
CFontData	IFontData, Get-FontData, Set-FontData	New class to encapsulate data for **Format Notebook** dialog controls and settings
CMainWindow	GetEditTextHan dle, SetEdit-TextHandle	Access methods for getting and setting the handle to the **EditText** pane
CMainWindow	SetTextFontInfo	Sets the text font information
CNotebook	DoNotebook, and others	Implements the **Format Notebook** dialog box

Customizing Methodology

This is an appropriate place to mention the overall philosophy of customizing AppMaker's generated code. The modules generated for our application are typical, especially at this point in its construction. They include:

CEnsembleMain — The main program function, which creates the initial instance of **CEnsembleApp** and serves to initiate execution of the application.

CEnsembleApp — The subclass module for the application instance. Along with its **ZEnsembleApp** superclass, it defines the applicationwide behavior of the application.

CEnsembleDoc — The document subclass that owns the primary window and interfaces directly to the abstract data class, **CEnsembleData**. Along with its superclass, **ZEnsembleDoc**, this class forms the basis for all document-oriented behavior of the application.

CEnsembleData — The abstract data class for the application. All operations that interface with the file system are routed through this module. Its direct ancestor is the TCL's **CDataFile** class.

CMainWindow — The primary window in the application and the one owned directly by the **CEnsembleDoc** class. Along with its superclass, **ZMainWindow**, it defines the appearance of data within the window.

CNotebook — The single dialog class, that, along with its **ZNotebook** superclass, provides the appearance and functionality of the dialog. It is instantiated via its embedded **DoNotebook** global function.

Following is a list of what, in general, we must always customize:

1. To implement any applicationwide features, such as adding the **Font** menu to the **CBartender** instance's list, we must modify the **CEnsembleApp** instance's methods.

2. To customize aspects of our application that relate to its document, including the creation of new container classes for document-oriented data, modifications to the input/output interface routines, and the handling of doc-

ument-oriented commands, we must modify the appropriate methods in the **CEnsembleDoc** instance.

3. To implement the input/output (I/O) functions that are specific to our document, we must modify the methods in the **CEnsembleData** instance, especially the **ReadData**, **WriteData**, and **Revert** methods. These are the sole interface with the physical data with which we will deal.

4. To implement the appearance of the main window, which handles specific drawing functions, we must modify the methods associated with the **CMainWindow** instance. In the case of our EditText pane, all of the drawing is automatically handled; however, we must still provide the window with access to the data to be transferred to and from the window via the document and its abstract data class.

5. Finally, to implement the functionality of the **Notebook** dialog, we must modify the **DoNotebook** function and the methods in the **CNotebook** instance.

Each new window added to the application will result in the generation of both a subclass and superclass, with methods that are similar to those in the **MainWindow** modules. Each new dialog will also result in the generation of a subclass and superclass for the dialog, with a **Do**<*dialog-name*> global function that must be modified, along with the methods in the dialog subclass instance. If a new addition requires that the file formats be changed, then modifications will have to be made to the **CEnsembleData** instance's methods. The customization methodology is thus predictable. The following section begins a step-by-step examination of our first set of custom changes to AppMaker's generated code.

Customizing the CEnsembleApp Methods

The **CEnsembleAp_** module has been modified to add functionality to the application level. A **SetUpMenus** method has been added to override the superclass's method, making the menu available in the **CBartender** class's list of menus. The code for the enhanced method is as follows:

```
void CEnsembleApp::SetUpMenus ()
{
   inherited::SetUpMenus();
   gBartender->AddMenu (5, TRUE, hierMenu);// Font menu
}
```

The override method first calls the inherited method and then sends a message to the **CBartender** instance (using the global **gBartender** variable) to add the Font menu (menu 5) to its list, treating the menu as a hierarchical menu (which won't cause the menu bar to be redrawn).

This method merely adds the menu to the **gBartender** instance's list of menus. It does not cause the menu to be installed in the menu bar. The primary purpose of adding this customized feature is to permit the menu's contents to be accessed easily by other application methods. Placing the initialization in the application guarantees that the menu will be installed at the earliest possible moment.

Implementing the File Menu Commands

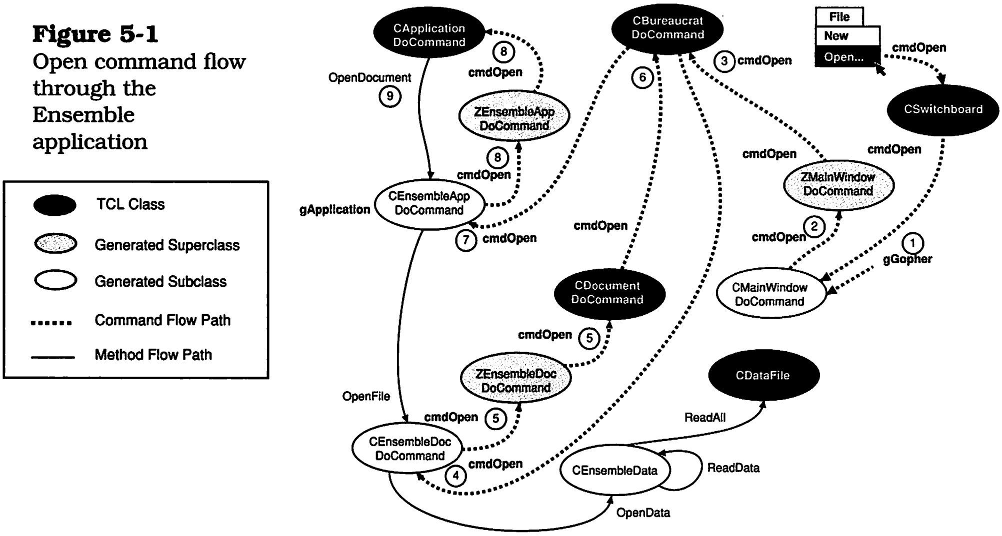
The next set of changes to AppMaker's generated code involves implementing the standard **File** menu commands. These provide for saving text that has been entered into the **MainWindow** pane, opening and reading text previously saved in a file, and reverting to a previously saved version of a file.

The commands named **New**, **Open**, **Save**, **Save As**, and **Revert** that appear in the **File** menu are all sent to the **DoCommand** method of the class whose handle is stored in the **gGopher** global variable.

While the **MainWindow** is active, these commands will be sent to the **MainWindow** subclass instance because the **gGopher** variable will be pointing to that instance. (The **Activate** method for the **CMainWindow** class sets the new value of the **gGopher** variable, when the window is activated.)

The sequence of events that occur when one of these commands is issued is shown in Figure 5-1 and is described in the steps that follow (the numbers in the figure correspond to the step numbers below):

Figure 5-1
Open command flow through the Ensemble application



1. Assuming that the **Open** command has been selected, **CSwitchboard** will send this command to the **DoCommand** method of the current **gGopher**, which is pointing to the **CMainWindow** instance.

2. The **CMainWindow**'s **DoCommand** method calls the **inherited DoCommand** method in the **ZMainWindow** class.

3. The **ZMainWindow**'s **DoCommand** method doesn't recognize the **Open** command, so it calls its **inherited DoCommand** method, in the **CBureaucrat** class.

4. The **CBureaucrat**'s **DoCommand** method sends the **Open** command to the **DoCommand** method in the supervisor for the current instance, which in this case is an instance of **CEnsembleDoc**.

5. Neither **CEnsembleDoc** nor its superclass, **ZEnsembleDoc**, recognizes the **Open** command; instead, they pass it up to the inherited method in the **CDocument** class.

6. Once again, the **CDocument**'s **DoCommand** method is unable to recognize the **Open** command, so it passes the command to its inherited method from the **CDirector** class, which then passes the command up to the **CBureaucrat** class to handle.

7. As in step 4, the **CBureaucrat** class sends the command to the **DoCommand** method in the supervisor of the current instance, which in this case would be an instance of the **CEnsembleApp** class.

8. Neither the **CEnsembleApp** nor its **ZEnsembleApp** superclass handles the **Open** command in its **DoCommand** method, so the command is passed to the **CApplication** class.

9. Fortunately, the command's long trek ends here, because the **DoCommand** method in the **CApplication** class does indeed recognize the **Open** command. In addition, it also recognizes the **New**, **Quit**, and **Show Clipboard** commands. When the **Open** command is recognized, the **CApplication**'s **DoCommand** method calls the **OpenDocument** method, which is an empty method in that class, but is overridden in the **ZEnsembleApp** class.

The preceeding steps describe the (somewhat circuitous) journey of a command to its intended handler. Most of the commands in the **File** menu are recognized in either the **CApplication** or **CDocument** class in the TCL. The cases in the corresponding **DoCommand** methods in those classes call upon other methods that must be overridden in the user's supplied code. The reason for this is that only the user knows what is required to open the selected document, how to read its data, how to save its data, and how to revert to a previously saved version of a file.

Fortunately, AppMaker generates most of the code to provide the functionality we need for these tasks. The following sections describe the additional code that we have added to implement the **File** menu command handlers.

CreateDocument Method Code

The **CApplication** class recognizes the **New** command and sends a **CreateDocument** message, which is handled by the override method in the **ZEnsembleDoc** class, as shown on page 32. The override method in turn sends a **NewFile** message, which is also handled by the generated code in the **ZEnsembleDoc** class, as shown in the sample code on page 33. The generated code for these methods is unchanged in the new version of the **Ensemble** application. However, because we wish our text window to be initialized so that the user can immediately begin typing, we have added an override for the **NewFile** method in our **CEnsembleDoc** class. The code for this method is as follows:

```
void  CEnsembleDoc::NewFile   (void)
{
   inherited::NewFile();
   InitTextFormat();
}
```

After the inherited **NewFile** method is called, we invoke a new method that initializes the text pane format. This method makes use of the new instance variables that we have defined in the **EnsembleDoc** module. The complete class definition for the new module is as follows:

CEnsembleDoc *class declaration (beginning)*

```
class CEnsembleDoc : public ZEnsembleDoc
{
public:
   //
   // manually added instance variables
   //
   CFontData *theTextData;

   //
   // generated public methods
   //
   void  IEnsembleDoc (CApplication  *aSupervisor,  // is override
                       Boolean     printable);
   void  UpdateMenus(void);         // is override
   void  DoCommand(long    theCommand);    // is override

   //
   // manually added methods
   //
   void  NewFile (void);                    // is override
   void  OpenFile(SFReply*macSFReply);  // is override
   void  InitTextFormat(void);
```

CEnsembleDoc *class declaration (concluded)*

```
    CMainWindow *GetTextWindow (void);

protected:
    // your application-specific instance variables:
};
```

The new **InitTextFormat** method is responsible for changing the text format to the font, size, style, and alignment saved in the text file. The code for the **InitTextFormat** method is as follows:

```
void   CEnsembleDoc::InitTextFormat(void)
{
    itsMainWindow->SetTextFontInfo(theTextData);
}
```

The **InitTextFormat** method sends a message that is handled by the **SetTextFontInfo** method in the **CMainWindow** instance to accomplish its purpose. The **theTextData** is an instance of the **CFontData** class, as shown in the **CEnsembleDoc** class declaration. To complete the picture, the code for the **SetTextFontInfo** method is as follows:

```
void   CMainWindow::SetTextFontInfo   (CFontData *theFontData)
{
    fontInfo  itsFontData;

    itsFontData = theFontData->GetFontData();
    Field3->SetFontNumber(itsFontData.fontNumber);
    Field3->SetFontSize(itsFontData.fontSize);
    Field3->SetFontStyle(0); // reset first
    Field3->SetFontStyle(itsFontData.fontStyle);
    Field3->SetAlignment(itsFontData.fontAlign);
}
```

The **CMainWindow** class must handle changing the attributes of its panes, because, for example, the **Field3** variable is specific to the **CMainWindow** class (and refers to the **CAMEditText** pane). The **theFontData** argument to the **SetTextFontInfo** method will be discussed later, in the context of the **Notebook** dialog code descriptions.

OpenDocument Method Code

The generated code for the **OpenDocument** method in the **ZEnsembleApp** class is left as is. The code sends an **OpenFile** message to a newly created instance of **CEnsembleDoc**. The **OpenFile** method is called with the **SFReply** record, identifying the file that the user wishes to open. The code for the **CEnsembleDoc**'s **OpenFile** method is as follows:

```
void   CEnsembleDoc::OpenFile   (SFReply  *macSFReply)
{
   Handle theData;

   inherited::OpenFile(macSFReply);
   theData = ((CEnsembleData *)itsFile)->GetEditTextHandle();
   ((CMainWindow *)itsWindow)->SetEditTextHandle (theData);
   theData = ((CMainWindow *)itsWindow)->GetEditTextHandle();
   ((CEnsembleData *)itsFile)->SetEditTextHandle(theData);
   InitTextFormat();
}
```

The code in the **CEnsembleDoc**'s **OpenFile** method first calls the method inherited from its **ZEnsembleDoc** superclass. It is important to recall that the superclass code is never customized; all customizing is applied to the subclass code. This enables AppMaker to regenerate the superclass code as new interface elements are added to the application, and the subclass code can remain untouched.

The superclass method (**ZEnsembleDoc::OpenFile**) is responsible for creating a new instance of the **CEnsembleData** class, initializing this instance, and then sending it the necessary messages to implement the required input/output operations. The **CEnsembleData** class is charged with the responsibility for handling all of the physical I/O for the application. This partitioning of tasks between the **CEnsembleDoc** and **CEnsembleData** classes is important. While the former can inherit behavior from the **CDocument** hierarchy, the latter class inherits its behavior from the **CDataFile** and **CFile** classes. This gives the **CEnsembleData** class methods the ability to use the TCL methods for performing file I/O. Following is the **ZEnsembleDoc**'s **OpenFile** method, shown for reference:

None of the code in the 'z' file superclass modules is modified in any way.

```
void    ZEnsembleDoc::OpenFile      (SFReply   *macSFReply)
{
    Str63     theName;

    itsData = new CEnsembleData;
    itsData->IEnsembleData (this);
    itsData->SFSpecify (macSFReply);
    itsData->OpenData (fsRdWrPerm);
    itsFile = itsData;
    BuildWindows ();
    itsFile->GetName (theName);
    if (itsWindow != NULL) {
        itsWindow->SetTitle (theName);
        itsWindow->Select ();
    }
}
```

Notice that the superclass method creates a new instance of **CEnsembleData**, initializes the instance, and sends it **SF-Specify** and then **OpenData** messages. The code for the **IEnsembleData** method is as follows:

```
void   CEnsembleData::IEnsembleData   (CDocument *theDocument)
{
    inherited::IDataFile ();
    hasFile = FALSE;
    itsDocument = theDocument;

    // your application-specific initialization
    itsEditTextData = NULL;

}
```

The **IDataFile** message is handled by the **CDataFile** class in the TCL. The **IDataFile** method initializes the instance variables for the class. The **IEnsembleData** method sets the **hasFile** instance variable to FALSE, saves the reference to **theDocument** into its **itsDocument** instance variable, and includes the line of code to set a new instance variable called **itsEditTextData** to NULL.

The **OpenFile** method for the **ZEnsembleDoc** superclass sends the **SFSpecify** message, which is handled by the **CFile** class in the TCL. The **SFSpecify** method saves the volume, directory, and file name information for the selected file. Finally, the **OpenData** message is sent to the new **CEnsembleData** instance, whose code is as follows:

```
void   CEnsembleData::OpenData   (SignedByte   permission)
{
    Open (permission);
    hasFile = TRUE;
    ReadData ();
}
```

The **permission** argument passed to the **OpenData** method is a constant named **fsRdWrPerm** that gives both read and write permission for the file. The **OpenData** method then calls the **Open** method that is inherited from the **CDataFile** class. The **Open** method performs the toolbox call that opens the file. When it returns, the **OpenData** method sets the **hasFile** instance variable to TRUE and then sends a **ReadData** message.

All of the code for the **OpenData** method was generated by AppMaker and has not been altered. However, AppMaker isn't able to know the format of the data in the file that was just opened. Nevertheless, it generates code that is almost perfect for our application in this stage of its development.

The code for the **ReadData** method is located in the **CEnsembleData** module and is as follows:

```
void   CEnsembleData::ReadData   (void)
{
    //
    // modified to reference itsEditTextData
    //
    itsData = ReadAll ();
    itsEditTextData = itsData;
}
```

The only code we have added to AppMaker's generated code in the preceeding method is the replacement statement that saves the handle to the data (returned by the **ReadAll** method in the **CDataFile** class) into a new instance variable that we've called **itsEditTextData**. At the point when **ReadData** finishes execution, the entire contents of the file have been read, the file is still open and positioned at its end, and a handle to the data has been saved. When **ReadData** returns, it will resume execution in the **OpenFile** method of the **ZEnsembleDoc** class, as shown on page 98. The next action taken by the **ZEnsembleDoc**'s **OpenFile** method (on page 98) is to create the document's window, by sending the **BuildWindows** message. The **BuildWindows** code is found in the **ZEnsembleDoc** module and will be described later, when we discuss the generated code for the **MainWindow** class. After creating the **MainWindow**, the **OpenFile** code gets the file name and places it in the title bar of the window. Then it selects the window, bringing it to the front as the active window. This completes the process of opening an existing file, in response to the **Open** command.

DoSave Method Code

When the user chooses the **Save** command from the **File** menu, the command travels through the route shown in Figure 5-1 and is intercepted by the **DoCommand** method in the **CDocument** class. The code in that method sets the cursor to the watch icon and then sends a **DoSave** message. The **DoSave** method in the **CDocument** class is empty; however, it is overridden in the **ZEnsembleDoc** superclass:

```
Boolean   ZEnsembleDoc::DoSave   (void)
{
   if (itsFile == NULL)
   {
      return (DoSaveFileAs ());
   }
   else
   {
      if (itsData->Save ())
      {
         dirty = FALSE;
         return (TRUE);
      }
      else
      {
         return (FALSE);
      }
   }
}
```

As usual, the **DoSave** code was generated by AppMaker and is unmodified. It first checks whether a file is already associated with the document. If not, it sends a **DoSaveFileAs** message; otherwise, it sends a **Save** message to the class associated with the **itsData** instance variable. In the case of the **Ensemble** application, this is the **CEnsembleData** class.

AppMaker generates code for all of the methods that perform operations on file data in the **CEnsembleData** class. This not only keeps the physical file operations separate from the methods that are appropriate to the document as a whole, but allows the **CEnsembleData** methods to reference the inherited methods directly in the **CDataFile** and **CFile** classes.

The code for the **Save** method is as follows:

```
Boolean   CEnsembleData::Save   (void)
{
   if (hasFile)
   {
      return (WriteData ());
   }
   else
   {
      // shouldn't be called in this case
      return (FALSE);
   }
}
```

The **Save** method code generated by AppMaker has not been modified. It tests to ensure that a file has been opened or previously saved, and if so, it calls the **WriteData** method, the code for which is as follows:

***WriteData** method code (beginning)*

```
Boolean   CEnsembleData::WriteData   (void)
{
   CMainWindow *theTextWindow;
   Handle theData;

   //
   // modified WriteData to get the TextEdit pane's Text Handle
   // and then write out the contents of that handle.
   //
   theTextWindow = ((CEnsembleDoc *)itsDocument)->GetTextWindow();
   theData = theTextWindow->GetEditTextHandle();
```

***WriteData** method code (concluded)*

```
    SetEditTextHandle(theData);
    WriteAll (itsEditTextData);
    return (TRUE);
}
```

The code for the **WriteData** method has been modified to get a handle to the **CAMEditText** pane in the **CMainWindow** instance and then write out the contents of that handle. The data in the handle represents the edited version of the original data and is what we want to save to a file. The **WriteAll** method is located in the **CDataFile** class in the TCL. The method of getting the handle to the data is somewhat complicated by the nature of data isolation afforded by the object-oriented programming methodology. In this case, the **CEnsembleData** class “knows” nothing about the nature of the source of the data, but merely that it needs to write the data out. To get the handle to the **CAMEditText** pane, we first send a message to the document to retrieve a reference to its text window (**GetTextWindow**).

Once we have a reference to the proper window, we can send it a message to return a reference to its **EditText** data (**GetEditTextHandle**). When we have retrieved the handle to the **EditText** data, we also store it into an instance variable in the **CEnsembleData** class by sending it in a **SetEditTextHandle** message. With a handle to the data, we can now write the data out through the **WriteAll** method in the **CDataFile** class of the TCL.

The **WriteAll** method repositions the selected file to its beginning and writes out the entire contents of the text addressed by the handle.

SaveAs Method Code

When the user selects the **Save** command from the **File** menu, and no data file is currently associated with the contents of the **MainWindow**, the **DoSave** method (shown on page 100) sends a **DoSaveFileAs** message. The **DoSaveFileAs** message is also sent by the **CDocument** class when the user selects the **SaveAs** command from the **File** menu. In either case, the **DoSaveFileAs** method (located in the **CDocument** class) displays a standard “Save File” dialog box and allows the user to specify the file into which the data are to be saved.

This is accomplished by sending a **PickFileName** message, which is also handled in the **CDocument** class in the TCL. The method associated with this message calls the **SFPutFile** toolbox function to perform the function of displaying the dialog box and allowing the user to navigate within it to specify the desired file. When the file has been selected, then the **DoSaveFileAs** method sends a **DoSaveAs** message, which is overridden by the **ZEnsembleDoc** class. The code is as follows:

```
Boolean   ZEnsembleDoc::DoSaveAs   (SFReply   *macSFReply)
{
   if (itsData->SaveAs (macSFReply))
   {
      itsFile = itsData;
      if (itsWindow != NULL)
      {
         itsWindow->SetTitle (macSFReply->fName);
      }
      dirty = FALSE;
      return (TRUE);
   }
   else
   {
      return (FALSE);
   }
}
```

When the **DoSaveAs** method executes, it sends a **SaveAs** message to the class associated with the **itsData** instance variable, which is the **CEnsembleData** class in this case. The code for the **SaveAs** method is as follows:

```
Boolean   CEnsembleData::SaveAs   (SFReply   *macSFReply)
{
   OSErr   ignoreErr;
   if (hasFile)
      Close ();
   SFSpecify (macSFReply);
   ignoreErr = HDelete (volNum, dirID, name);   // in case already exists
   CreateNew (gSignature, kFileType);
   Open (fsRdWrPerm);
   hasFile = TRUE;
   return (Save ());
}
```

The first thing the **SaveAs** method does is check whether a file is already associated with the data. If so, it closes that file and then executes the **SFSpecify** method inherited from the **CFile** class to set the new volume, directory, and file name information. It then calls the **HDelete** toolbox method to delete the new file if it currently exists, calls the **CreateNew** method inherited from the **CFile** class, and calls the **Open** method inherited from the **CDataFile** class in the TCL.

After opening the file, it can save the data by calling the **Save** method in the **CEnsembleData** class, as shown on page 101. The data are written out to the new file with the same **WriteData** method used by the **Save** command.

Revert Method Code

The **cmdRevert** command is recognized by the **DoCommand** method in the **CDocument** class. The code for this case sends a **DoRevert** message to the document, which is intercepted by the override method in the **ZEnsembleDoc** class. The code for the **DoRevert** method is as follows:

```
void  ZEnsembleDoc::DoRevert  (void)
{
   itsData->Revert ();
   dirty = FALSE;
}
```

The method accomplishes its task by sending a **Revert** message to the **CEnsembleData** class, represented by the **itsData** instance variable.

As previously mentioned, all the physical file I/O is performed by the **CEnsembleData** class, due to its ability to inherit methods from the **CDataFile** and **CFile** classes.

When the user decides to revert to a previous version of a file, the application must first determine whether a file in fact exists. If not, then the **Revert** method should dispose of the current data and do nothing else. This is the best interpretation of the user's intent.

If the file does exist, then the **Revert** method can read the data from it and proceed to enter its data into the **EditText** pane in the main window. These operations are shown in the following code for the **Revert** method:

This method has been modified quite a bit, to access the handle to the ***EditText*** *data*

```
void   CEnsembleData::Revert   (void)
{
   CMainWindow *theTextWindow;
   Handle theData;

   DisposeData ();
   if (hasFile)
   {
      //
      // reread the original file's data
      //
      ReadData ();
      theTextWindow = ((CEnsembleDoc *)itsDocument)->GetTextWindow();
      if(theTextWindow)
      {
         //
         // set the new EditText handle, and then get it back
         //
         theTextWindow->SetEditTextHandle(itsEditTextData);
         theData = theTextWindow->GetEditTextHandle();
         SetEditTextHandle(theData);
      }
   }
}
```

In order to replace the existing data with the contents of the file, the code must first dispose of the existing data, read the contents of the file, and then store a handle to the **EditText** data into the **CAMEditText** pane in the **MainWindow**. The process of getting a reference to the window and then sending it the message to set the new **EditText** handle is similar to the approach outlined on page 101. In this case, we are, of course, storing the handle, rather than retrieving it.

Once the handle has been set by sending the **SetEdit-TextHandle** message to the window, it is immediately retrieved. This is necessary because the **CEditText** class's **SetTextHandle** method makes a copy of the data and then changes the handle to point to the copy. Our **Revert** method then saves the new handle by calling the **SetEditTextHandle** method in our **CEnsembleData** class.

Adding Methods to the CMainWindow Class

We have written three new methods for this class that implement getting and setting the **EditText** pane's handle and also setting its text font parameters. The code for the **GetEditTextHandle** method is as follows:

```
Handle CMainWindow::GetEditTextHandle (void)
{
   return ((Handle) Field3->GetTextHandle ());
}
```

As is apparent, this method calls the **GetTextHandle** method that is inherited from the TCL's **CEditText** class and returns it to the caller. The code for the counterpart method, **SetEditTextHandle**, is as follows:

```
void   CMainWindow::SetEditTextHandle   (Handle theData)
{
   Field3->SetTextHandle (theData);
}
```

The foregoing code sends a **SetTextHandle** message to the **EditText** field, which is inherited from the TCL's **CAbstractText** class. It is important to bear in mind that the **SetTextHandle** method creates a copy of the data contained in the **itsData** instance variable and installs a new handle into the **EditText** pane. Therefore, the methods that we have previously shown that reference the **SetEditTextHandle** method immediately send a **GetEditTextHandle** message, to acquire the real handle to the text.

The data used to set the **EditText** pane's font, size, style, and alignment are handled by a new method called **SetTextFontInfo**. As you will see in the next section, concerned with implementing the **Format Notebook** command, the **CEnsembleDoc** class's **DoCommand** method calls **SetTextFontInfo** to change the **EditText** pane's parameters, based on the new values obtained from the **DoNotebook** method's execution. The code for the **SetTextFontInfo** method, from page 96, is as follows:

```
void   CMainWindow::SetTextFontInfo   (CFontData *theFontData)
{
   fontInfo   itsFontData;

   itsFontData = theFontData->GetFontData();
   Field3->SetFontNumber(itsFontData.fontNumber);
   Field3->SetFontSize(itsFontData.fontSize);
   Field3->SetFontStyle(0); // reset first
   Field3->SetFontStyle(itsFontData.fontStyle);
   Field3->SetAlignment(itsFontData.fontAlign);
}
```

As you can see, the method sends a **GetFontData** message to the **CFontData** class, which retrieves a handle to the object. When the object's handle is stored into the local **itsFontData** variable, then the various text font, size, style, and alignment methods in the **CEditText** class of the TCL can be referenced to set the new parameters.

Implementing the Format Notebook Command

The first step in implementing the **Format Notebook** command is to devise methods for setting the initial values of the various fields and controls in the **Notebook** dialog and for retrieving and saving these values from one invocation of the dialog to the next.

We decided that the best way to encapsulate these values and also provide access to them was to create a completely new class, whose name was chosen to be **CFontData**.

The **CFontData** object is described in a new header file called **FontData.h**, and its methods are defined in a new source code file called **FontData.c**.

The **FontData.h** file contains the definition of a structure to hold the initial or current font information for the text pane. It also contains instance variables to hold those data, as well as the definition of each of the changeable controls and fields in the **Notebook** dialog. The complete content of the **FontData.h** file is shown in two sections. The header for the file and the definition of the **fontInfo** structure are as follows:

```
/* FontData.h -- font data class */

#define _H_FontData
#include <CObject.h>

typedef struct
{
   short    fontNumber;
   short    fontSize;
   short    fontStyle;
   short    fontAlign;

} fontInfo;
```

The definition of the **CFontData** class and its instance variables and methods are as follows:

```
class CFontData : public CObject
{
public:

   fontInfo fontData;

   short    BoldCheck;
   short    ItalicCheck;
   short    UnderlineCheck;
   short    OutlineCheck;
   short    ShadowCheck;
   short    CondenseCheck;
   short    ExtendCheck;
   Str255   FontNameString;
   Str255   FontSizeString;
   Str255   FontSampleString;
   long     RadioStationID;
   short    FontSelection;
   short    SizeSelection;

   void     IFontData(void);
   fontInfo GetFontData(void);
   void     SetFontData (fontInfo theData);
};
```

The methods declared in the class definition for the new class are **GetFontData**, **SetFontData**, and **IFontData**.

The code for the **CFontData** class's **GetFontData** method is as follows:

```
fontInfo   CFontData::GetFontData   (void)
{
   return fontData;
}
```

As you can see, all this access method does is return the structure holding the current font number, size, style, and alignment settings.

The code for the **SetFontData** method is very similar to that for **GetFontData**. It merely sets the **fontData** structure's content:

```
void  CFontData::SetFontData(fontInfo theData)
{
   fontData = theData;
}
```

Finally, the code to set the default values of all the **CFont-Data** instance's variables, is as follows:

***IFontData** method code*

```
void   CFontData::IFontData   (void)
{
   fontData.fontNumber = 0;
   fontData.fontSize   = 12;
   fontData.fontStyle  = 0;
   fontData.fontAlign  = teFlushLeft;

   BoldCheck= 0;
   ItalicCheck= 0;
   UnderlineCheck= 0;
   OutlineCheck= 0;
   ShadowCheck= 0;
   CondenseCheck= 0;
   ExtendCheck= 0;
   RadioStationID= 139;   // LeftRadioViewID
   CopyPString("\pSystem", FontNameString);
   CopyPString("\p12", FontSizeString);
   CopyPString("\pSample", FontSampleString);
   FontSelection= 0;
   SizeSelection= 3;
}
```

The preceeding code is invoked by the **CEnsembleDoc** class's **IEnsembleDoc** method, to create an instance of the **CFontData** class for each open document—the **Ensemble** application will let you have more than one document open at a time—and then initialize the instance.

The **IEnsembleDoc** method, as customized, is as follows:

```
void   CEnsembleDoc::IEnsembleDoc   (CApplication *aSupervisor,
                                       Boolean printable)
{
   CFontData*aFontData;

   inherited::IEnsembleDoc (aSupervisor, printable);
   aFontData = new CFontData;
   aFontData->IFontData();
   theTextData = aFontData;
}
```

After calling the inherited **IEnsembleDoc** method, which is found in the **ZEnsembleDoc** superclass, the **IEnsembleDoc** method creates a new instance of the **CFontData** class, sends it an **IFontData** initialization message, and then stores the new instance reference into the document's **theTextData** variable. The initial settings for the **Notebook** dialog are:

❖ In the **fontData** structure, the font is set to 0, which refers to the "System Font," the size is set to 12 points, the style is set to 0, which is "plain," and the alignment is set to left-justified.

❖ All the font style checkboxes are initialized as being unchecked.

❖ The "StationID," referring to which radio button is selected in the set of alignment choices, is initialized to enable the left justified button.

❖ The font name string is set to "System", the font size string is set to "12", and the font sample string is set to "Sample".

❖ The number of the highlighted cell in the font name list is set to 0 (the first cell, which refers to the system font), and the number of the highlighted cell in the font size list is set to 3 (the fourth cell, which refers to the size 12 entry).

Initializing the instance variables in the **CFontData** instance provides a firm basis for the initial text font characteristics of the **EditText** pane. When the **MainWindow** instance is created, the **EditText** pane is initialized with the default font settings. Both the **New** and **Open** commands in the **File** menu call a method named **InitTextFormat**, whose code is shown on page 96. This method sends a message to the **MainWindow** method **SetTextFontInfo**, whose code is also shown on page 96. The **SetTextFontInfo** method uses the **CFontData** class's **GetFontData** access method to retrieve the contents of the **fontInfo** structure, so that it can initialize the **EditText** pane with the current font information.

Up to this point, we've discussed how the **EditText** pane's text characteristics are initialized and set. What remains is a discussion of how the default characteristics are changed and what methods are involved in this process. The first link in the modification chain is the **DoCommand** method in the **CEnsembleDoc** class. This method overrides the behavior in its superclass, as follows:

```
void  CEnsembleDoc::DoCommand  (long  theCommand)
{
   switch (theCommand)
   {
      case cmdNotebook:
      {
         DoNotebook(this);
         InitTextFormat();
         break;
      }
      default:
      {
         inherited::DoCommand (theCommand);
         break;
      }
   }
}
```

The override **DoCommand** method specifically tests for the existence of a **cmdNotebook** command, which is sent to the method in response to the user's selection of the **Format** menu's **Notebook** command. The handler then calls the **DoNotebook** function (a global function) with a handle to the **CEnsembleDoc** instance (shown as **this** in the function's argument). The **DoNotebook** function is responsible for creating, initializing, and managing the operation of the **Notebook** dialog, as well as retrieving the values from its controls and fields when the user has completed a set of text format modification actions.

The default-generated code for the **DoNotebook** function is shown on page 78. The following listing of the function is broken into several parts, so that its custom features can be more easily explained.

Initial DoNotebook Code

The first section of code for the **DoNotebook** function is as follows:

*Part 1 of the **DoNotebook** global function*

```
void DoNotebook (CDirectorOwner *aSupervisor)
{
	CNotebook *dialog;
	long		dismisser;
	Str255	fontNameString;
	Str255	fontSizeString;
	Str255	fontSampleString;
	short		aChoice;
	CFontData *theFontInfo;

	dialog = NULL;
	theFontInfo = ((CEnsembleDoc *) aSupervisor)->theTextData;
	TRY {
			dialog = new CNotebook;
			dialog->INotebook (aSupervisor);
```

The code is nearly identical to the default code generated by AppMaker; we have merely added some new local variables to hold the font name, font size, and font sample strings.

A variable has been defined to hold the chosen entry in the font name and font size lists. In addition, the initialized **CFontData** instance is assigned to the **theFontInfo** variable, by accessing the **theTextData** instance variable in the **CEnsembleDoc** class, using the **aSupervisor** argument to reference the instance.

After creating a new instance of the **CNotebook** class, the code calls the **INotebook** method to initialize the instance. The default code for the **INotebook** method is shown on page 79. We have enhanced this code by adding some statements that assign "StationID's" to the text justification radio buttons.

The definition of the "StationID's" is contained in the **Notebook.h** header file, as shown in the following code:

```
enum
{
   LeftRadioViewID=139,
   CenterRadioViewID,
   RightRadioViewID,
   ForceLeftRadioViewID
};
```

The "StationID's" have been assigned an arbitrary sequence of codes, beginning with **139**. These were chosen to coincide with the resource ID's for the radio button controls. The code for the new version of the **INotebook** method is as follows:

```
void  CNotebook::INotebook(CDirectorOwner *aSupervisor)
{
   inherited::IZNotebook (aSupervisor);

   LeftRadio->ID= LeftRadioViewID;
   CenterRadio->ID= CenterRadioViewID;
   RightRadio->ID= RightRadioViewID;
   ForceLeftRadio->ID= ForceLeftRadioViewID;
}
```

When this method calls the inherited **IZNotebook** method, the **Notebook** dialog and all its controls and fields are created and initialized, as shown on page 73. During the execution of the **IZNotebook** method, several of the dialog item instances require additional initialization. AppMaker has generated the skeleton for the **INotebook** method, and we have added the custom code.

Sizing the Font Name List

The font name list is the first to be customized. The declaration for the **CList25** class in the **Notebook.h** module contains the definition of the **fontMenu** instance variable, as shown in the following code:

```
class CList25 : public CAMTable
{
public:
   //
   // new instance variables
   //
   short           numFonts;
   MenuHandle   fontMenu;

   void   IViewTemp(CView*anEnclosure,
                       CBureaucrat*aSupervisor,
                       Ptr        viewData);      // is override
   void   GetCellText(CellaCell,
                       short     availableWidth,
                       StringPtr  itsText);       // is override
};
```

The **IViewTemp** initialization method for the font name list class (**CList25**) must be modified to add the font names to the list:

```
void   CList25::IViewTemp(CView*anEnclosure,
                           CBureaucrat*aSupervisor,
                           Ptr        viewData)
{
   inherited::IViewTemp (anEnclosure, aSupervisor, viewData);

   fontMenu = GetMHandle(FontID);
   numFonts = CountMItems(fontMenu);
   AddRow (numFonts, 0);
}
```

This method makes use of the **Font** menu we created in Chapter 3, beginning on page 54. AppMaker generated code to add the names of all the user's fonts to the menu in the **ZEnsembleApp** class's **SetUpMenus** method, shown on page 70. The override method gets the handle to the **Font** menu, calls the toolbox routine **CountMItems** to determine how many names are in the menu, and then adds that number of rows to the **CList25** instance. When the rows are added, methods in the TCL's **CTable** class will compute the number of cells in the table, and then will repeatedly call the **GetCellText** method to get the text for each of the table's cells.

Initializing the Font Names

We override the **GetCellText** method for the **CList25** instance and provide the font names as follows:

```
void  CList25::GetCellText  (Cell        aCell,
                             short       availableWidth,
                             StringPtr   itsText)
{
   short    index;

   index = aCell.v;
   GetItem(fontMenu, index+1, itsText);
}
```

The **GetCellText** method uses the vertical (row) component of the **aCell** argument to the function to get the menu item associated with that cell number. The toolbox routine **GetItem** retrieves the font name directly into the location pointed to by the **itsText** argument to the method.

Sizing the Font Size List

The **CList29** dialog item is handled in a similar fashion. The code in the **IViewTemp** method generated by AppMaker has been customized to initialize the font size list with a set of constant strings, as follows:

```
void  CList29::IViewTemp  (CView   *anEnclosure,
                           CBureaucrat   *aSupervisor, Ptr viewData)
{
   inherited::IViewTemp (anEnclosure, aSupervisor, viewData);
   CopyPString ("\p 8 9101214161820242832 36", typeSizes);
   AddRow (12, 0);
}
```

After the inherited **IViewTemp** method is called, an instance variable called **typeSizes** is initialized with a string consisting of font sizes in text form, and the **CTable** class's **AddRow** method is called to add 12 rows to the table. When this is done, the **CTable** class will call the **GetCellText** method to access each of the list's font size values.

Initializing the Font Size List

The **GetCellText** method has been rewritten as follows to provide the text for the requested cells:

```
void CList29::GetCellText (Cell aCell, short availableWidth,
                           StringPtr itsText)
{
   short    strIndex;
   strIndex = (aCell.v << 1) + 1;
   *itsText++ = 2;
   *itsText++ = typeSizes[strIndex++];
   *itsText  = typeSizes[strIndex];
}
```

Recall that the text for the font size list was stored as a single string, as shown on page 115. Each size is stored as exactly two characters in the string (note the blank space before the entries of '8' and '9'), so we can build the text entry pointed to by the **itsText** argument simply by storing a string length of 2 and the two characters of the size string that correspond to the selected cell.

When the **Notebook** dialog is invoked for the first time, the settings of the checkboxes, the selected list entries, the radio buttons, and the text fields will be set to the default values established when the **CFontData** instance was first initialized. (See the **IFontData** code on page 109.)

When the user changes the default values, the new code in the **DoNotebook** function copies the new values to the **CFontData** instance, so that these values are shown on the next invocation of the dialog.

Continuing the DoNotebook Code's Initialization

The next section of the **DoNotebook** function performs further initialization of the dialog's items:

Part 2 of ***DoNotebook***

```
dialog->theInfo = theFontInfo->GetFontData();
dialog->BoldCheck->SetValue(theFontInfo->BoldCheck);
dialog->ItalicCheck->SetValue(theFontInfo->ItalicCheck);
dialog->UnderlineCheck->SetValue(theFontInfo->UnderlineCheck);
dialog->OutlineCheck->SetValue(theFontInfo->OutlineCheck);
dialog->ShadowCheck->SetValue(theFontInfo->ShadowCheck);
dialog->CondenseCheck->SetValue(theFontInfo->CondenseCheck);
dialog->ExtendCheck->SetValue(theFontInfo->ExtendCheck);
dialog->Group17->SetStationID(theFontInfo->RadioStationID);
dialog->List25->SetChoice(theFontInfo->FontSelection);
dialog->List29->SetChoice(theFontInfo->SizeSelection);
CopyPString(theFontInfo->FontNameString, fontNameString);
dialog->Field14->SetTextString(fontNameString);
CopyPString(theFontInfo->FontSizeString, fontSizeString);
dialog->Field15->SetTextString(fontSizeString);
CopyPString(theFontInfo->FontSampleString, fontSampleString);
dialog->Field16->SetTextString(fontSampleString);
dialog->Field16->SetFontNumber(dialog->theInfo.fontNumber);
dialog->Field16->SetFontSize(dialog->theInfo.fontSize);
dialog->Field16->SetFontStyle(dialog->theInfo.fontStyle);
dialog->Field16->SetAlignment(dialog->theInfo.fontAlign);
```

The foregoing statements access the data stored in the instance of the **CFontData** class retrieved into the **theFontInfo** variable. They set the values of the controls and fields in the dialog to the previous settings. The fields and controls are addressed via the **dialog** variable, which points to the **CNotebook** instance.

Creating and Operating the Dialog

After the dialog has been initialized, the next step is to make it visible and let the user make any desired changes. This is accomplished by the following code:

Part 3 of ***DoNotebook***

```
dialog->BeginDialog ();
dismisser = dialog->DoModalDialog (cmdOK);
```

These statements show and operate the dialog. The **BeginDialog** and **DoModalDialog** methods are inherited from the **CDialogDirector** class in the TCL.

Handling User Interaction

Once the **Notebook** dialog has been made visible, the next phase of operation is to handle any changes that the user makes to the dialog's setting. The TCL will provide all of the functionality with regard to user feedback when the dialog is operated:

❖ Checkboxes are checked or unchecked automatically.

❖ A previously selected radio button is deselected, and the new one is selected when clicked.

❖ List items are highlighted as they are clicked.

Even though the user feedback for these items is automatically provided, we must also add feedback that shows the *results* of the user's selections. Notice that the **Notebook** dialog (see Figure 3-22) has an **EditText** field just below the **Font** list, another just below the **Size** list, and a third just below the group of **Justification** radio buttons. Our custom code must perform the following actions for these items:

❖ The field below the **Font** list must show the name of the currently selected font.

❖ The field below the **Size** list must show the currently selected size.

❖ The field below the **Justification** radio button group must show an example (the word **Sample** is the default) of the application of all the settings.

In order to accomplish the preceeding objectives, we must become aware of any changes made to the initial settings. This is accomplished in two different ways:

❖ When the checkboxes and radio buttons are defined, AppMaker automatically assigns each of these a "click command" that is sent to the current **gGopher** when the user clicks on the item.

❖ When text is entered into the EditText panes, or if any of the list items is selected, the appropriate class sends a **BroadcastChange** message, which is intercepted by the

TCL's **CBureaucrat** class and reissued as a **Provider-Changed** message to the item's supervisor (the **Notebook** dialog in this case).

For the first case, a **DoCommand** method in the **CNotebook** class is used to intercept clicks in the checkboxes and radio buttons. When the **Notebook** dialog is operational, the **gGopher** variable will be set to point to the **CNotebook** class. It will revert to the **CMainWindow** class when the dialog is dismissed. The concept of the **DoCommand** method is discussed in Chapter 4.

For the second case, a **ProviderChanged** method is generated, and all changes to the font or size list or to the text fields will be handled in this method. The concept of the **ProviderChanged** method is discussed in Chapter 4. The modified **DoCommand** code is as follows:

DoCommand *method code (beginning)*

```
void  CNotebook::DoCommand(longtheCommand)
{
   short style = -100, align = -100;
   switch (theCommand)
   {
      case cmdBoldCheck:
      {
         style = bold;
         break;
      }
      case cmdItalicCheck:
      {
         style = italic;
         break;
      }
      case cmdUnderlineCheck:
      {
         style = underline;
         break;
      }
      case cmdOutlineCheck:
      {
         style = outline;
         break;
      }
      case cmdShadowCheck:
      {
         style = shadow;
```

DoCommand *method code (concluded)*

```
            break;
        }
        case cmdCondenseCheck:
        {
            style = condense;
            break;
        }
        case cmdExtendCheck:
        {
            style = extend;
            break;
        }
        case cmdCenterRadio:
        {
            align = teCenter;
            break;
        }
        case cmdRightRadio:
        {
            align = teFlushRight;
            break;
        }
        case cmdForceLeftRadio:
        {
            align = teFlushDefault;
            break;
        }
        case cmdLeftRadio:
        {
            align = teFlushLeft;
            break;
        }
        default:
        {
            inherited::DoCommand (theCommand);
            break;
        }
    }
    if(style != -100)
    {
        Field16->SetFontStyle(style);
        theInfo.fontStyle ^= style;
        DrawSample();
    }
    if(align != -100)
    {
        Field16->SetAlignment(align);
        theInfo.fontAlign = align;
        DrawSample();
    }
}
```

Notice that we have merely set the values of the **style** and **align** variables in the individual cases and then added code at the end of the method to call the **SetFontStyle** and **SetAlignment** methods for **Field16**, which is the **EditText** field that shows an example of the formatted text, located below the **Justification** radio group. In addition, in each case, once the style or alignment is changed, we call a new method to draw the sample text in the **Field16** pane.

The code for the **DrawSample** method is as follows:

DrawSample *method code (beginning)*

```
void CNotebook::DrawSample(void)
{
   StringPtr fontName;
   short  fontNum;
   long   fontSize, strLength;
   Str255  theFontText, theSizeText, the SampleText;

   strLength = Field14->GetLength();
   if(strLength > 0)
   {
      Field14->GetTextString(theFontText);
      if(EqualString(theFontText, "\pSystem", TRUE, TRUE))
      {
         fontNum = systemFont;
      }
      else if(EqualString(theFontText, "\pApplication", TRUE, TRUE))
      {
         fontNum = applFont;
      }
      else
      {
         GetFNum(theFontText, &fontNum);
      }
   }
   else
   {
      fontNum = systemFont;
   }
   strLength = Field15->GetLength();
   if(strLength > 0)
   {
      Field15->GetTextString(theSizeText);
      StringToNum(theSizeText, &fontSize);
   }
   else
   {
```

***DrawSample* method code (concluded)**

```
        fontSize = 12;
    }
    CopyPString("\pSample", theSampleText);
    Field16->SetTextString(theSampleText);
    Field16->SetFontNumber(fontNum);
    Field16->SetFontSize(fontSize);
    theInfo.fontNumber = fontNum;
    theInfo.fontSize = fontSize;
}
```

The **DrawSample** code has to make special provisions for the **System (systemFont)** and **Application (applFont)** font name choices and also has to handle the case where the user has deleted or typed in a new font name. The font size isn't quite as important, because if the size is too small or too large, the sample will not be visible in the **Field16 EditText** field. Different text can also be typed into the sample field if desired.

The final event-handling method is **ProviderChanged**, which is invoked with changes to the list selections or when the user types into the **EditText** fields. The code for **ProviderChanged** is as follows:

***ProviderChanged* method code (beginning)**

```
void  CNotebook::ProviderChanged (CCollaborator *aProvider,
                                  long     reason,
                                  void*    info)
{
    short index, num;
    Str255   theText;

    if (aProvider == Field14)
    {
        if (Field14->GetLength () != 0)
        {
            // there is some text, so show a sample in Field16
            DrawSample();
        }
    }
    if (aProvider == Field15)
    {
        if (Field15->GetLength () != 0)
        {
            // there is some text, so show a sample in Field16
            DrawSample();
        }
```

ProviderChanged
method code
(concluded)

```
    }
    if (aProvider == List25)
    {
        if (List25->HasSelection ())
        {
            // store selection in EditText field
            if(List25->GetChoice(&index))
            {
                GetItem(((CList25 *)List25)->fontMenu, index+1, theText);
                Field14->SetTextString(theText);
                DrawSample();
            }
        }
    }
    if (aProvider == List29)
    {
        if (List29->HasSelection ())
        {
            // store selection in EditText field
            if(List29->GetChoice(&index))
            {
                index = (index << 1) +1;
                theText[0] = 2;
                theText[1] = ((CList29 *) List29)->typeSizes[index++];
                theText[2] = ((CList29 *) List29)->typeSizes[index++];
                Field15->SetTextString(theText);
                DrawSample();
            }
        }
    }
}
```

In the foregoing, we've modified the version of **ProviderChanged** shown in Chapter 4 by changing the code to test for nonzero text field lengths and eliminating the code for handling changes to the **Field16** (Sample) field entirely. The text written into the **Field16** field will always be the word **Sample**, in the selected font, size, style, and justification. In each case where the font name or font size text field contents have been entered manually by the user, the code in the **ProviderChanged** method calls the **DrawSample** method to show the results of the change. In a similar fashion, when a selection is made in either the font name or font size list, the selected cell is identified, and its contents are written into the corresponding **EditText** field. The **DrawSample** method is also called in these cases.

Retrieving the Modified Dialog Values

Finally, after the user finishes making changes and dismisses the dialog by clicking on either the **OK** or **Cancel** buttons, the following code is executed in the **DoNotebook** function:

Part 4 of ***DoNotebook*** *function*

```
if (dismisser == cmdOK)
{
   theFontInfo->SetFontData(dialog->theInfo);
   theFontInfo->BoldCheck = dialog->BoldCheck->GetValue();
   theFontInfo->ItalicCheck = dialog->ItalicCheck->GetValue();
   theFontInfo->UnderlineCheck = dialog->UnderlineCheck->GetValue();
   theFontInfo->OutlineCheck = dialog->OutlineCheck->GetValue();
   theFontInfo->ShadowCheck = dialog->ShadowCheck->GetValue();
   theFontInfo->CondenseCheck = dialog->CondenseCheck->GetValue();
   theFontInfo->ExtendCheck = dialog->ExtendCheck->GetValue();
   theFontInfo->RadioStationID = dialog->Group17->GetStationID();
   dialog->List25->GetChoice(&aChoice);
   theFontInfo->FontSelection = aChoice;
   dialog->List29->GetChoice(&aChoice);
   theFontInfo->SizeSelection = aChoice;
   dialog->Field14->GetTextString(fontNameString);
   CopyPString(fontNameString, theFontInfo->FontNameString);
   dialog->Field15->GetTextString(fontSizeString);
   CopyPString(fontSizeString, theFontInfo->FontSizeString);
   dialog->Field16->GetTextString(fontSampleString);
   CopyPString(fontSampleString, theFontInfo->FontSampleString);
}
```

If the dialog was dismissed by clicking the **OK** button, then the foregoing code will extract the values of the fields and controls and store them back into the corresponding instance variables of the **CFontData** instance.

If the user dismissed the dialog by clicking the **Cancel** button, then the values would not be replaced in the **CFontData** instance.

Disposing of the Dialog and Handling Failures

The final action of the **DoNotebook** function is unchanged from the default-generated code:

Part-5 of the ***DoNotebook*** *function (beginning)*

```
   dialog->Dispose ();
}
CATCH
```

Part-5 of the ***DoNotebook*** *function (concluded)*

```
    {
        ForgetObject (dialog);
    }
    ENDTRY;
}
```

In the event that a failure is detected during the creation or operation of the **Notebook** dialog, the CATCH block will handle the failure and then propagate the condition up to the next higher level. At the point of this failure, the only thing that we can do is delete the dialog by using the **ForgetObject** method.

When the **DoNotebook** function returns to the **CEnsembleDoc** class's **DoCommand** method, that method calls the **InitTextFormat** method to set the current font, size, style, and justification for the **MainWindow** pane's text (see page 111). Note that although the **InitTextFormat** method is called even if the dialog was cancelled, the previous settings will be intact, so the **InitTextFormat** method will change the pane's settings to their previous values.

The next chapter introduces a completely new feature into the **Ensemble** application, and the succeeding chapters explain the default code generated by AppMaker and the changes needed to make the default code fully operational.

Exercises

1. Explain the THINK Class Library's technique of processing command events. What flexibility does the interception of commands in the **CBureaucrat** class provide?

2. Describe the relationship of the **CEnsembleDoc** and **CEnsembleData** classes. Why are these separate classes in AppMaker's generated code?

3. What is the purpose of the **Revert** method in the **CEnsembleData** class?

4. Explain why the **GetEditTextHandle** and **SetEditText Handle** methods are necessary in the **MainWindow** class. How are these used in the application?

5. Describe the purpose of defining the **CFontData** class. Why isn't the **fontInfo** structure just global to the application as a whole? Under what circumstances could multiple instances of this structure exist?

6. Describe a method for creating a list of font sizes other than the fixed string that is presented. (*Hint*: Think about a similar list of items that is implemented as a resource.)

7. Describe what mechanism is used to direct mouse clicks on buttons and checkboxes in the Notebook dialog to the corresponding **DoCommand** method?

8. Although the generated code for the **ProviderChanged** method only includes tests for the text fields and list instances in the Notebook dialog, describe how this method could be used to advantage to handle events that occur in other user interface elements in the Notebook dialog? (*Hint*: Examine the implementation of the **SetValue** method in the TCL's **CControl** class to aid in forumlating your answer.)

Chapter 6

Adding a Worksheet Window

In this chapter, we are going to add a spreadsheet like window to the Ensemble application. The chapter will focus on the additions to the user interface, providing step-by-step instructions for using AppMaker to construct the window. The following chapter will discuss the default-generated code for this new window, and the chapter after that will document the custom additions to AppMaker's generated code to make the worksheet completely functional.

This and the next two chapters are quite detailed, so you may want to stop and review what we've covered so far before continuing. The addition of the worksheet window to the application is intricate; however, with the facilities of AppMaker and the THINK Class Library, its implementation is quite straightforward.

Beginning in this chapter, we will discontinue showing pictures of menu command choices being made. We assume that, by now, you have become familiar with the menus in AppMaker and THINK C and will not need these pictures as a reference.

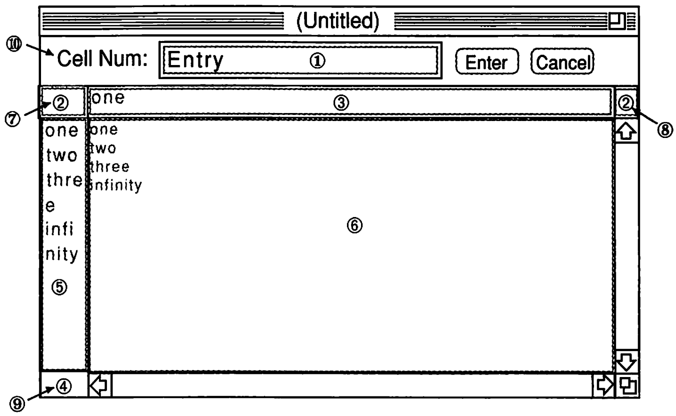
We will also introduce the use of Apple's ResEdit application in the latter part of the chapter to "fine-tune" the resources that AppMaker constructs. The instructions on how to accomplish the necessary modifications will be detailed in a step-by-step approach.

The result at the end of this and the next two chapters will be an operational worksheet that handles the calculation of formulas that include constants, operators, and worksheet cell references. The contents of a worksheet can be saved and read, along with the text in the original main window.

Creating a New Window for Ensemble

Adding a new window to Ensemble is a very simple procedure when AppMaker's tools are used. However, the window we will be adding has quite a few components that must be placed in particular locations, so the construction process is somewhat detailed. The diagram in Figure 6-1 shows the fully constructed window, so that you will have an idea of its final appearance.

Figure 6-1
CalcWindow appearance inside AppMaker



The numbered panes in the figure correspond, in general, to composite structures. The only exception to this is the **CStaticText** field, shown as item ⑩.

Basically, the window consists of three layers of objects on top of a standard Macintosh window. Notice that the window has no close box. We will construct each item in the steps that follow and show the exact position and measurements of the item, according to AppMaker's **Item Info** dialog. The following steps are liberally illustrated with screen shots of the **Item Info** dialog boxes.

Before beginning the step-by-step discussion of the construction of the window, it is appropriate to explain the nature and composition of the numbered panes in Figure 6-1. The details of each item are summarized in Table 6-1.

Table 6-1
Component definitions for **CalcWindow** items shown in Figure 6-1

No.	Outside	Inside	Dimensions	Sizing
①	Bord 132	AETx 135	(4,80 24,192)	H;4, V:4
②	Bord133	n/a	(32,-1 21,405)	H:5, V:4
③	Bord 134	ATbl 137	(0,32 21,358) (1,1 19,356)	H:5, V:4 H:5, V:4
④	Bord 135	n/a	(52,-1 181,33)	H:4, V:5
⑤	Bord 136	ATbl 138	(0,0 165,33) (1,1 163,30)	H:4, V:5 H:4, V:5
⑥	ScPn 134	ATbl 136	(53,32 180,372) (0,0 164,356)	H:5, V:5 H:5, V:5
⑦	②	Pane 129	(1,2 19,29)	H:4, V:4
⑧	②	Pane 130	(1,390 19,15)	H:4, V:4
⑨	④	Pane 131	(165,1 15,30)	H:4, V:5
⑩	AETx 134	n/a	n/a	n/a

The information in the table is rather detailed, but it will come in handy later, for "tweaking" the locations and dimensions of the **CalcWindow** components. The columns of the table are described as follows:

❖ The **No.** column contains the number corresponding to the **CalcWindow** item in Figure 6-1.

❖ The **Outside** column contains the resource type and number of the outermost item of the item group. In most cases, this is a **Bord** (CBorder) resource, but it may be a **ScPn** (CScrollpane) or **AETx** (CEditText) resource. In the three cases where this column contains a **No.** item, the number refers to the "owning" item number.

❖ The **Inside** column identifies the resource type and number of the inside item of the item group. This may be an **AETx** (EditText), **ATbl** (CTable), or **Pane** (CPane) item.

❖ The **Dimensions** column contains the position and dimensions of the components of each item. They are given, in order, as Top, Left, Height, and Width values (in pixels).

When two rows of dimensions are given, the top row corresponds to the **Outside** element, and the bottom row corresponds to the **Inside** element.

❖ The **Sizing** column specifies the final sizing characteristics for each element of the item. If two rows are given, the top row contains the sizing specification for the **Outside** element, and the bottom row specifies the sizing for the **Inside** element. Only two different sizing characteristics are used:

- A value of **4** indicates that the corresponding dimension (H = horizontal, V = vertical) is fixed and does not stretch or shrink as the window is resized.
- A value of **5** indicates that the corresponding dimension is able to stretch or shrink in proportion to the degree to which the window is resized.

The final point of interest, before we get into the step-by-step construction, is an exploded view of the **CalcWindow**, showing the base window and the three layers of elements. This view is shown in Figure 6-2.

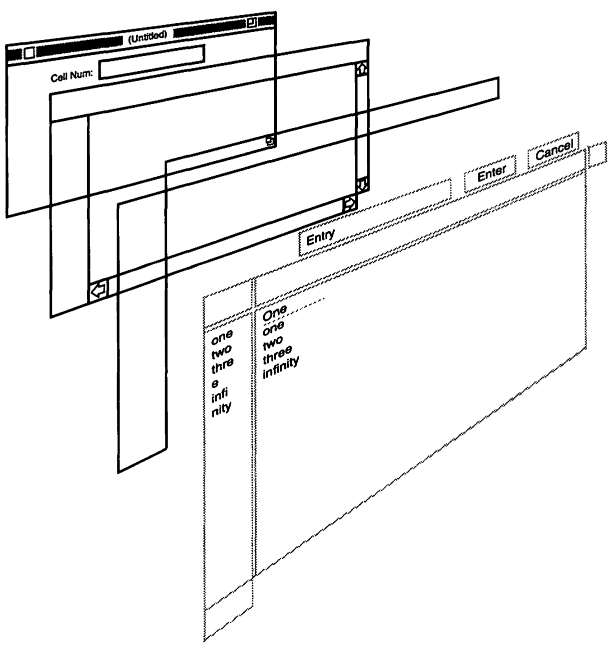


Figure 6-2
Exploded view of the construction of the **CalcWindow** element

Beginning Construction of the CalcWindow

The purpose of the following steps is to show how to add a new window to the Ensemble application's user interface. This window will perform many of the functions of a standard spreadsheet, so it will have a pane to enter a value into a specified cell, a main spreadsheet pane with individual rows and columns, and corresponding row and column label panes. The window will be added to the resource file that we zzused in the previous chapters of this book, that we have called **Ensemble.π.rsrc**. The steps for constructing the window are as follows:

1. Launch AppMaker, by double-clicking on the **Ensemble.π.rsrc** file, and then choose the **Windows** command from the **Select** menu.

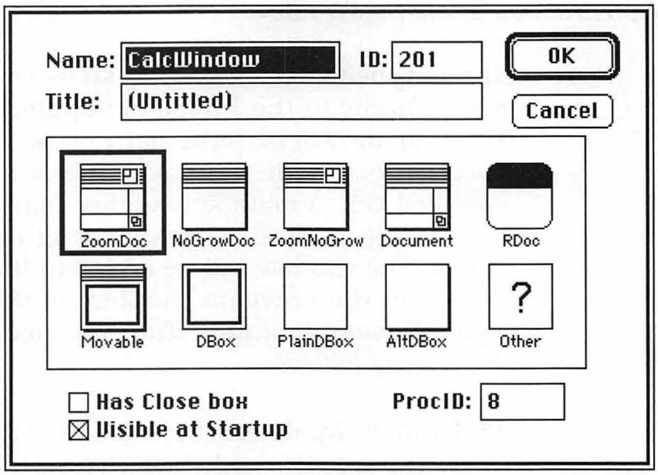
2. Selecting **Windows** allows you to modify an existing window or create a new window. To make all the window tools available, choose the **Tools as Text** command from the **View** menu.

3. Pull down the **Edit** menu and choose the **Create Window** command.

4. When you choose the **Create Window** command, AppMaker will display a dialog that shows pictures of the various types of windows that are available and the optional accessories for each. We want to select a standard document window, but one without a **close box**. The title and name of the window and the selected options are shown in Figure 6-3. Notice that except for the lack of a **close box**, this is a standard Macintosh document window. We've given the window the name **CalcWindow** and made its title **(Untitled)**. AppMaker will use the name you type to create the name of the corresponding source code modules, in this case, **CalcWindow**. The title is displayed, by default, in the title bar of the window when it is first opened.

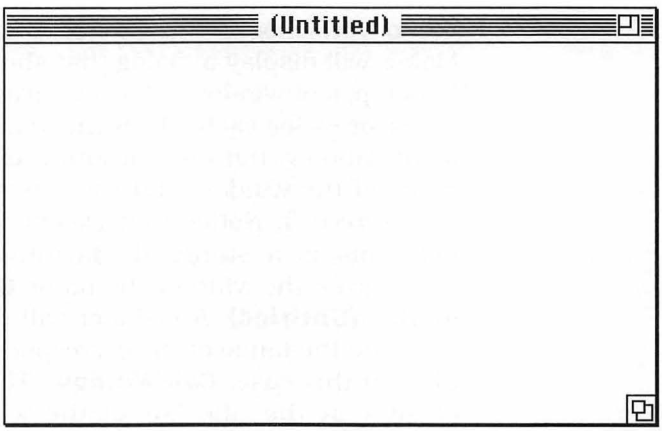
5. The appearance of the new window is shown in Figure 6-4. This window is almost identical to the one we created for the **MainWindow** in Chapter 3 (see Figure 3-3). The only difference is the absence of a **close box** in the new

Figure 6-3
CalcWindow
information dialog box



CalcWindow. We will later add a third window to the Ensemble application that will be identical to this window. In our user interface, no individual window can be closed without closing all of the windows. This is accomplished by clicking in the close box of the **MainWindow** or by choosing the **Close** command in the **File** menu when the **MainWindow** is in front.

Figure 6-4
Untitled
CalcWindow's
appearance



6. Position the default window near the upper left corner of your screen, and make its dimensions approximately 5.9 inches wide by 3.5 inches tall. It is rather important that you size the window fairly close to these dimensions, as

the various elements that make up the window's contents will need to be positioned and sized as the window is constructed. You can lay a ruler against your screen, and if your display has a resolution of 72 dots per inch (which is the case for most Mac displays), then the measurement will be quite accurate. Select the **CalcWindow** window by double-clicking on its name in the Selection List, as shown in Figure 6-5.

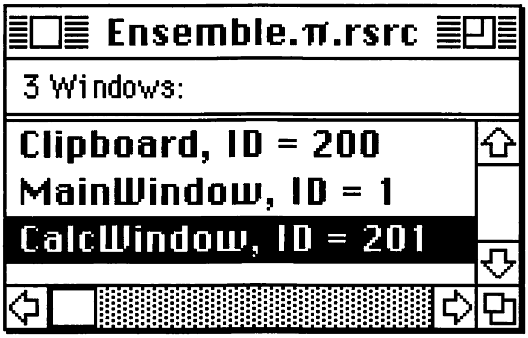


Figure 6-5 Selecting **CalcWindow** from AppMaker's Selection window

7. With the window active (its title bar is not dimmed), pull down the **Tools** menu and choose the **CBorder** tool. You are going to create the border element for the cell entry pane, shown as item ① in Figure 6-1. Position the mouse with the cursor at the approximate position of the top left corner of the border frame, and draw the border down and to the right, so that it has approximately the appearance shown in Figure 6-1.

8. Next, choose the **Item Info** command from the **View** menu.

9. You will see a dialog box with settings for the position and size of the element that is currently selected. This is identified at the bottom of the dialog box as a **CAMBorder** element. Change the settings in the dialog box to match those shown in Figure 6-6 by selecting and typing the new values into the corresponding fields.

10. The next item is the wide horizontal border that is identified as item ② in Figure 6-1. This item spans the entire width of the window and is approximately 20 pixels tall. With the **CBorder** tool still selected, position the mouse

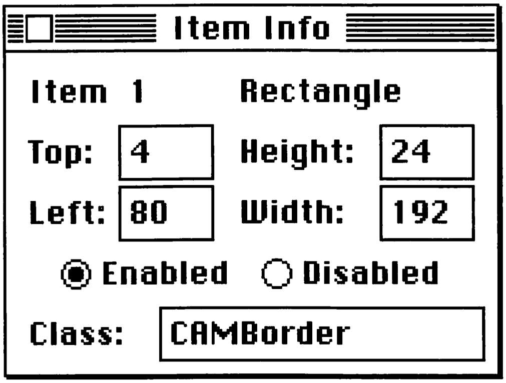


Figure 6-6
Item Info settings for the Entry pane border

on top of the window's left border, at the approximate position of the top left corner of the new border, and draw down and across to the right window border. The **Item Info** settings for this element are shown in Figure 6-7. If the border doesn't seem to be in quite the right relation to the window border, resize the window slightly by dragging on its **resize box**, so that the border's right edge overlaps the window's right border. Notice that the left edge of the new border is at position **-1**, with respect to the window coordinates. This ensures that the left edge of the new border overlaps the window's left border.

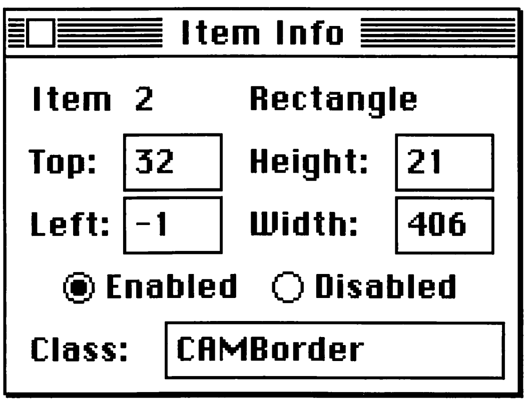


Figure 6-7
Item Info settings for the wide horizontal border

11. The next element is also a border and is identified as item ③ in Figure 6-1. This border overlaps the wide horizontal border, beginning about 31 pixels to the right of the window's left edge, and extends to about 16 pixels from the window's right edge. It lays right on top of the wide border, so when you draw it, position the mouse right on top of the previous border's top line, at approximately the position of the new border's top left corner. Click and drag the mouse to draw a border that is the same height, but not quite as wide as the wide border. This creates a border within a border and is the easiest way to create the appearance and functionality we desire. The **Item Info** for this border is shown in Figure 6-8. Change the settings for your border to match those shown in the figure. This border will enclose the column labels for the spreadsheet.

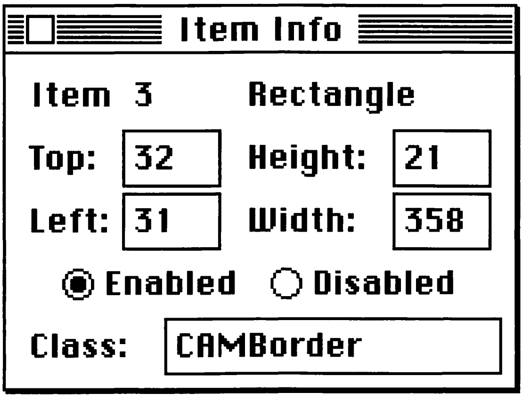
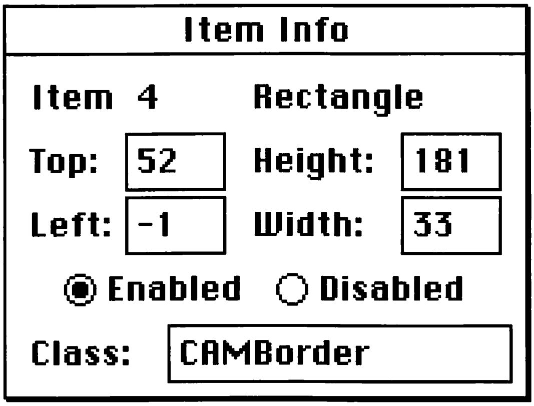


Figure 6-8
Item Info settings for **CalcWindow's** column label border

12. The next element is also a border and is identified as item ④ in Figure 6-1. This is a tall, vertical border that is approximately 30 pixels wide and extends from the bottom of the wide horizontal border, at the left side of the window, to the bottom of the window. Position the mouse at the bottom left corner of the wide horizontal border, and drag down and to the right, to overlap both the bottom of the wide horizontal border and the bottom window border. The **Item Info** data for this border is shown in Figure 6-9. Make sure that the settings for your border match those shown in the figure. Like the wide horizontal border, this border is largely decorative, but will enclose another border, described next.

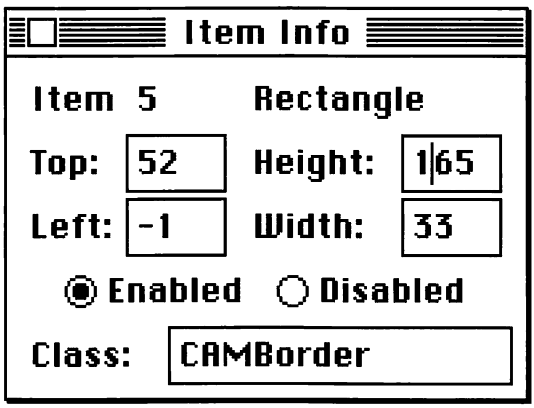
13. The final border lies on top of the previous border to form the enclosure for the spreadsheet's row label pane. The border's top left corner lies exactly on top of the top left

Figure 6-9
Item Info settings for tall vertical border



corner of the previous (tall vertical border) but isn't quite as tall. This new border is identified as item ⑤ in Figure 6-1. It extends to within 16 pixels of the bottom window border. The **Item Info** settings for this border are shown in Figure 6-10. Make sure that your settings agree.

Figure 6-10
Item Info settings for row label pane border



14. This completes the drawing of the **CAMBorder** elements. When you're done with the preceding steps, the window should have the appearance shown in Figure 6-11. Notice that the wide horizontal border overlaps the window's right and left borders and that the tall vertical border overlaps both the bottom of the horizontal border and the bottom window border. The horizontal and vertical label borders, which will hold the column and row label panes, exactly overlap their corresponding wide horizontal and

tall vertical borders. The next series of steps will show you how to add the panes that go inside the border elements.

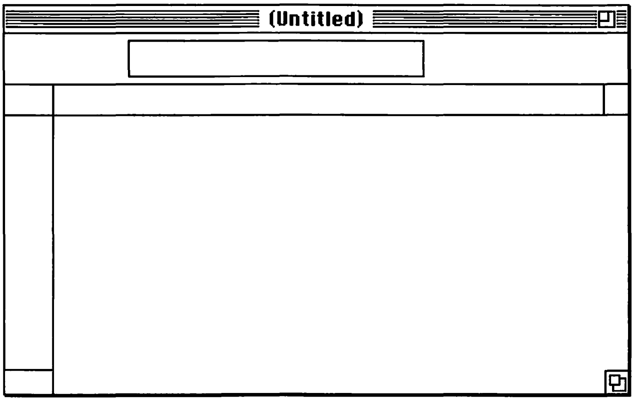


Figure 6-11
CalcWindow with all borders drawn

15. Pull down the **Tools** menu and choose the **CEditText** tool.

16. Position the mouse cursor just within the Entry border, identified as item ① in Figure 6-1. Click and drag the mouse to create a CAMEditText pane that is entirely within, but about 1 pixel smaller on each side, of the Entry border. After you release the mouse, click inside the pane and type the name **Entry** inside it. The **Item Info** settings for this pane are shown in Figure 6-12. Notice that this item is **Enabled**.

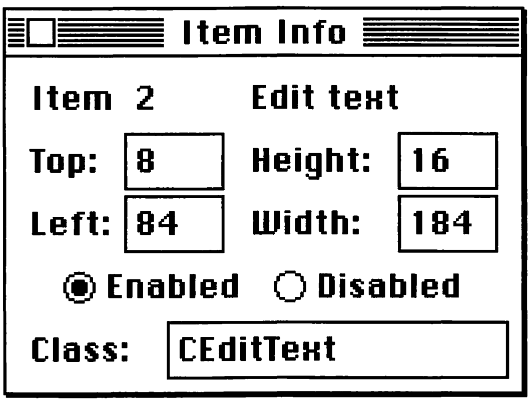


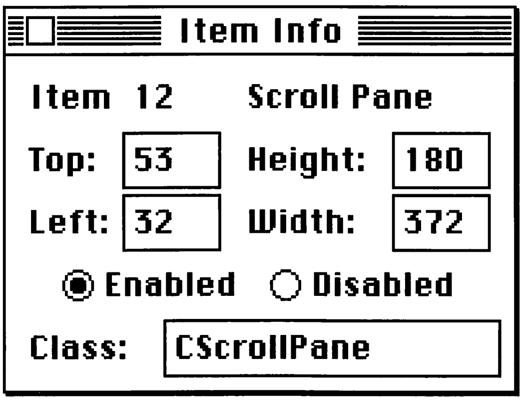
Figure 6-12
Item Info settings for **CAMEditText** Entry pane

17. The next element is identified as item ⑥ in Figure 6-1. This is the CScrollPane that permits the spreadsheet to scroll both horizontally and vertically. Choose the **CScrollPane** tool from the **Tools** menu.

18. The **CScrollPane** object is going to cover the majority of the bottom portion of the window. Its top left corner is positioned about 1 pixel below and 1 pixel to the right of the intersection of the horizontal and vertical label panes (shown as items ③ and ⑤ respectively, in Figure 6-1).

Position the mouse cursor near the point of this intersection, click, and drag down and to the right, until the scroll pane overlaps the right and bottom borders of the window frame. The **Item Info** settings for this element are shown in Figure 6-13. The scroll pane provides the horizontal scroll bar only. We will add the vertical scrollbar in the next step. The scroll pane also provides a framework within which the spreadsheet panorama can be installed.

Figure 6-13
Item Info settings for **CScrollPane**

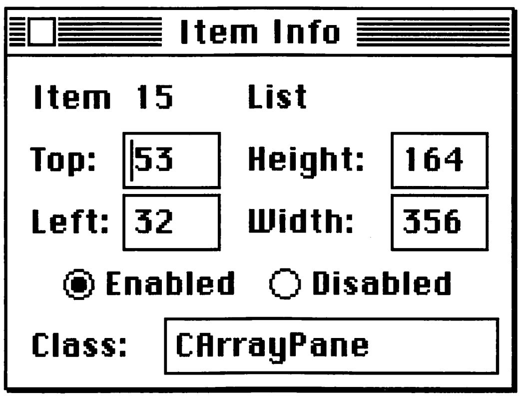


19. Create the horizontal scroll bar by choosing **CScrollBar** from AppMaker's **Tools** menu. Position the cursor on the middle bottom edge of the window, so that the cursor still retains the shape of a cross, and click the mouse button once. A horizontal scroll bar that fills the width of the scroll pane should be automatically drawn. If you don't achieve the desired results in the first try, delete the imperfect scroll bar and try the procedure again.

20. Now that the scroll pane is installed, along with its scroll bars, we can place the spreadsheet pane on top of it as its panorama. For the purpose of the Ensemble application, the TCL's **CArrayPane** class provides an excellent basis for our spreadsheet. Basically, **CArrayPane** is a subclass of the **CTable** class in which the data associated with each of the table's cells is kept in a separate array. The array and the table are associated, however, by an explicit dependency connection via the **CCollaborator** class. Whenever an element in the array is changed, the associated table will get a **ProviderChanged** message to trigger redrawing the affected cell. To create the spreadsheet pane, pull down the **Tools** menu and choose the **CArray-Pane** tool.

21. To draw the **CArrayPane** in the window, position the mouse at the top left corner of the **CScrollPane** element (shown as item ⑥ in Figure 6-1), and drag down and to the right until the pane covers the entire blank portion of the scroll pane element (excluding the scroll bars). The **Item Info** settings for the **CArrayPane** are shown in Figure 6-14.

Figure 6-14
Item Info settings for **CArrayPane**



22. After the **CArrayPane** is installed, we want to set its text style so that the text is smaller and in a different font than the default 12 point Chicago system font. Make sure that the **CArrayPane** is still selected, and then pull down the **Edit** menu and choose the **Text Style** command. Match the settings with those in the dialog box depicted in Figure 6-15 (9-point Geneva, plain style, with left justification).

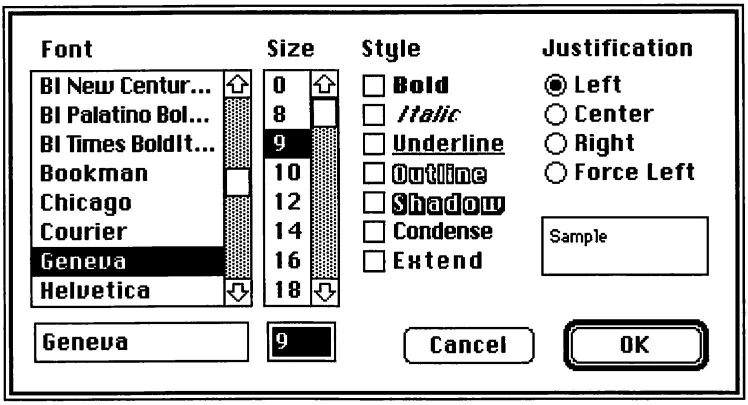


Figure 6-15
Setting the **Text Style** for the **CAMArrayPane** element

23. This and the next two steps are concerned with installing the **CTable** panes for the row and column labels. First, pull down the **Tools** menu and choose the **CTable** tool.

24. With the **CTable** tool selected, position the mouse cursor just inside the horizontal label border element (shown as item ③ in Figure 6-1), at its top left corner, and then click and drag until the table fills the inside of the border. You may experience some difficulty in creating a **CTable** pane that fits inside the border. Don't worry; just draw it the best you can, and then use the **Item Info** settings in Figure 6-16 to modify your settings to correct the table's position and size.

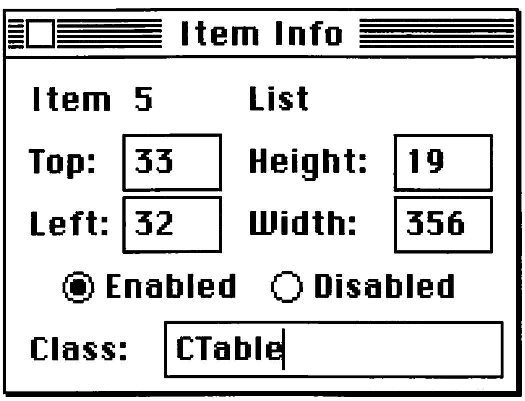


Figure 6-16
Item Info settings for Horizontal "column" label **CTable** element

25. The vertical row label **CTable** pane is constructed in the same fashion as in step 24. Position the mouse at the top

left corner, inside the border shown as item ⑤ in Figure 6-1. Click and drag the mouse down and to the right, until the entire row label border is filled. The **Item Info** settings for this element are shown in Figure 6-17.

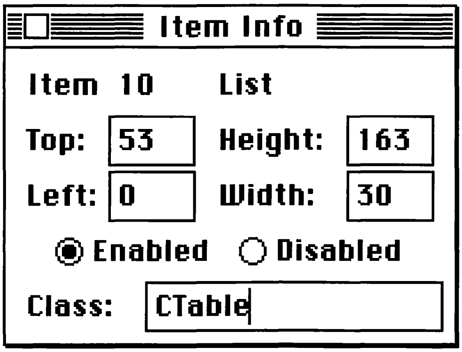


Figure 6-17
Item Info settings for Vertical "row" label **CTable** element

26. If you look carefully at the illustration in Figure 6-11, you will see that there are small "holes" in the horizontal and vertical border construction, at the top left, top right, and bottom left. These are currently unused portions of the window but we are going to fill them in with panes that could have useful purposes as we enhance the Ensemble application later. To "plug" these "holes," choose the **CPane** tool from the **Tools** menu.

27. Click within the top left hole of the horizontal border, and drag down and to the right to fill in the small "hole" in that border with a **CPane** element. The settings for this element are shown in Figure 6-18.

28. The next "hole" we're going to fill in is the top right pane. Click inside the border of that hole, and drag down and to the right to fill in that portion of the border with a **CPane** element. The **Item Info** settings for this element are shown in Figure 6-19.

29. The final "hole" that we will fill is at the bottom left corner of the window. Click the mouse inside the small border, and drag it down and to the right to fill in the "hole." The **Item Info** settings for this element are shown in Figure 6-20. This completes the procedure for filling in the holes in the **CBorder** panes.

Figure 6-18
Item Info settings
for top left **CPane**

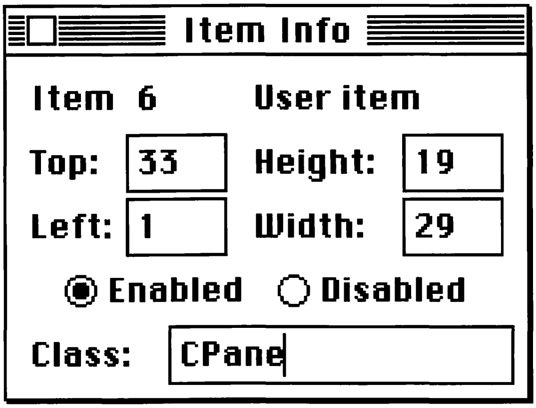


Figure 6-19
Item Info settings
for top right **CPane**

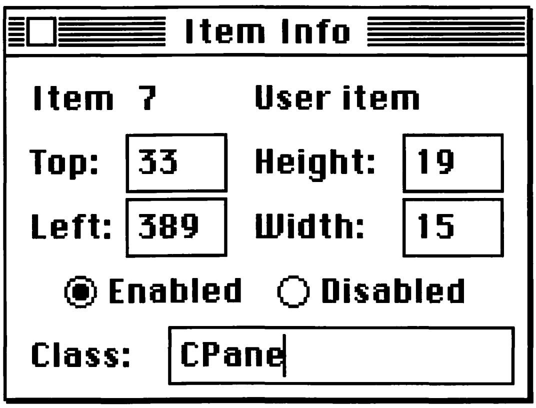
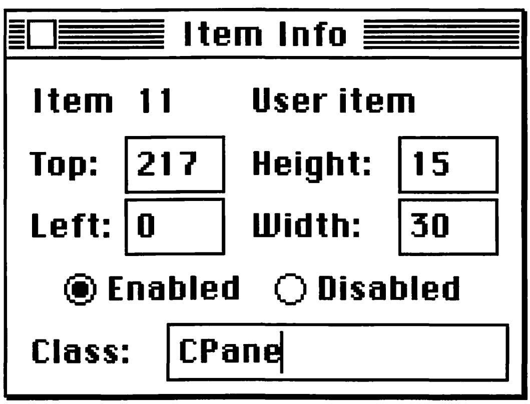


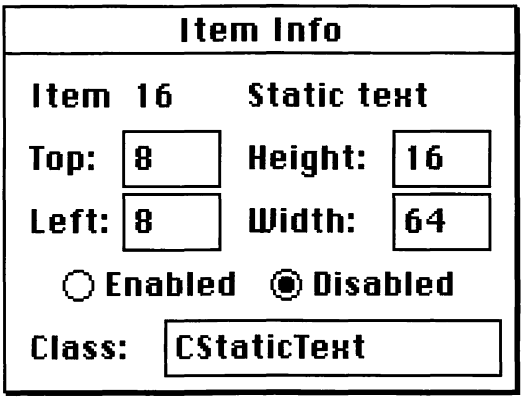
Figure 6-20
Item Info settings
for bottom left **CPane**



30. The last set of steps completes the construction of the remaining three user interface elements. Pull down the **Tools** menu and choose the **CStaticText** tool.

31. Position the mouse cursor at the top left of the window, about 10 pixels in from the left window border and centered vertically within the top and bottom boundaries of the **Entry** pane. Click the mouse button once. This will cause AppMaker to set the position of the leftmost character in the static text field. Type the characters **CellNum** at this time. The **Item Info** settings for this element are shown in Figure 6-21. Notice that the item is **Disabled**. This will prevent it from reacting to mouse clicks.

Figure 6-21
Item Info settings for **CStaticText CellNum** element



32. Select the **CButton** tool from the **Tools** menu.

33. Position the mouse cursor at the approximate location of the **Enter** button, as shown in Figure 6-1, and click the mouse button once. This will create a standard-size Macintosh button element. We are going to use a smaller version of this, so change the settings for this button to match those shown in Figure 6-22. Make sure that the button is **Enabled**, as it won't accept mouse clicks if it is set to **Disabled**.

34. Now, pull down the **Edit** menu and select the **Text Style** command, which will show the dialog pictured in Figure 6-23. Change the font and style of the text to correspond with the settings in the dialog (a 10-point, plain-style system font with center justification). Click **OK**, click inside

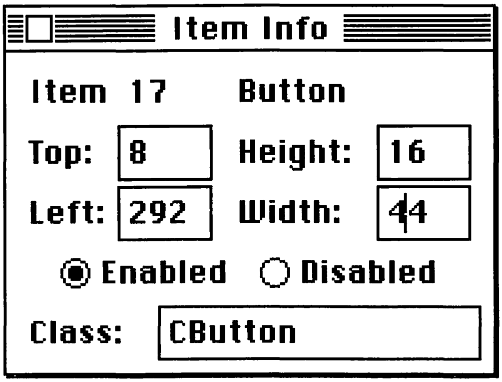


Figure 6-22
Item Info settings for the **Enter** button element

the button once or twice to ensure that you see a vertical bar cursor, and then type the word **Enter**.

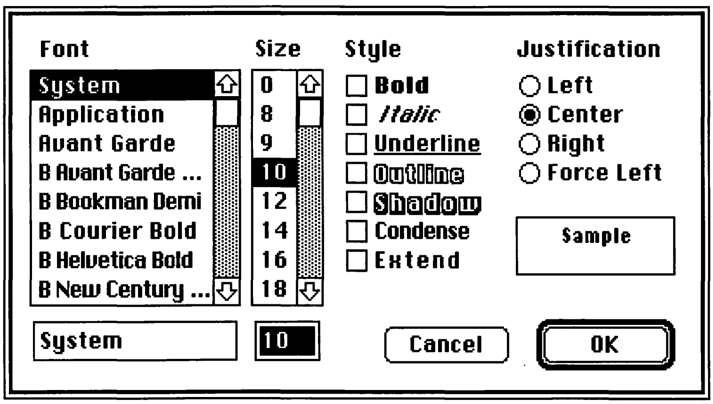


Figure 6-23
The **Text Style** dialog from the **Edit** menu

35. Next, create the **Cancel** button in the same way you created the **Enter** button. Click the mouse at the left edge of where the button is supposed to be situated, and then modify the **Item Info** settings to correspond to those shown in Figure 6-24. Once again, pull down the **Edit** menu and select the **Text Style** command, causing the dialog shown in Figure 6-23 to be displayed. The proper settings for the **Cancel** button duplicate those for the **Enter** button. Click inside the button and type the word **Cancel**.

The preceding set of steps completes the construction of the **CalcWindow** window. At this point, we are ready to generate

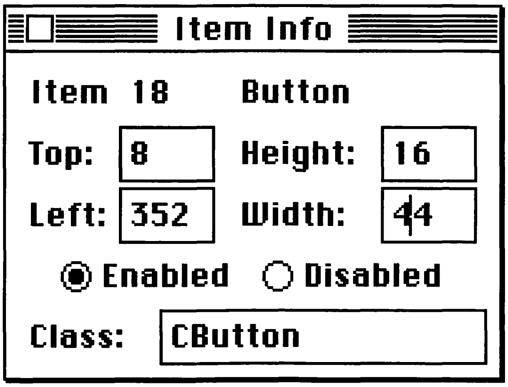


Figure 6-24
Item Info settings for **Cancel** button

code, but it will be helpful for you to look once again at the illustration in Figure 6-1 to make sure that your window appears as shown in the figure. If so, you're ready to generate code for the new version of the Ensemble application.

Generating Code for the CalcWindow Addition to Ensemble

To generate code for the newly added window, pull down the **File** menu and choose the **Generate** command.

When the **Generate** command is chosen, AppMaker will display a dialog that lists all the files that it intends to generate, as shown in Figure 6-25. In most cases, you will want to gen-

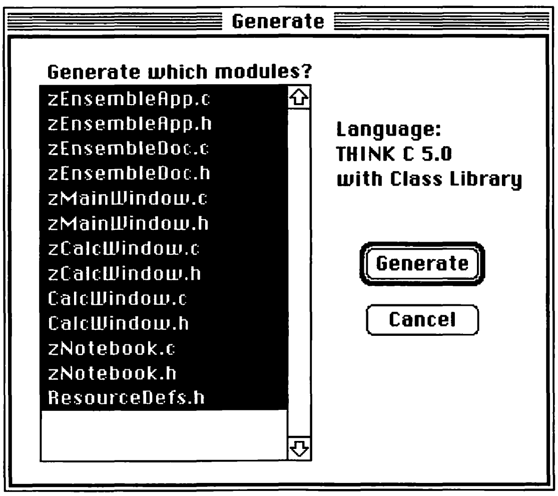


Figure 6-25
AppMaker's suggested list of files to generate

erate all the files that it suggests; however, in some cases, where all you have done is "tweak" a user interface element setting, or move an element within the window, you may want to generate code for only the particular affected modules. With practice using AppMaker, and observing the code that it generates, you will be able to make that determination. For our purposes, all of the suggested modules will be generated. Notice that two new files called **CalcWindow.c** and **CalcWindow.h**, have been added and that AppMaker also intends to regenerate all the files whose names begin with the letter **z**. These are the superclass files, many of which will be modified to take into account the new window we've added. Click the **Generate** button for this window, and when AppMaker is finished, choose the **Quit** command from the **File** menu and click **Save** to save the changes to the resource file.

After the files have been generated, you will want to recompile the project. Figure 6-26 shows all the files for the new version of the project, as seen in the Finder's small icon view.

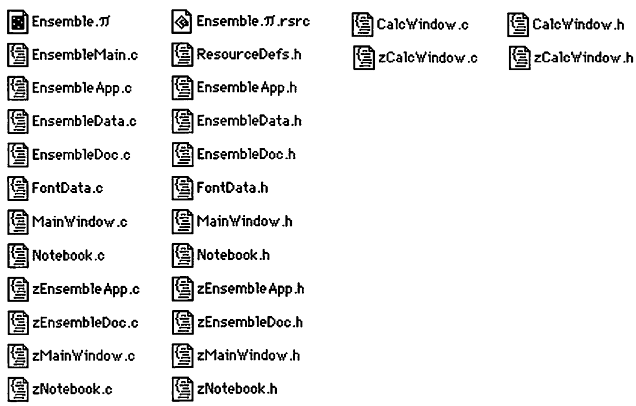


Figure 6-26
Complete set of files for the new version of the Ensemble application

To recompile the project, follow these simple directions:

1. Launch the THINK C application by double-clicking on the **Ensemble.π** project file. Pull down the **Source** menu and select the **Add** command.

2. When the **Add** command is chosen, THINK C will display a dialog that shows all of the source files in the current folder that are not present in the project. This is illustrated in Figure 6-27.

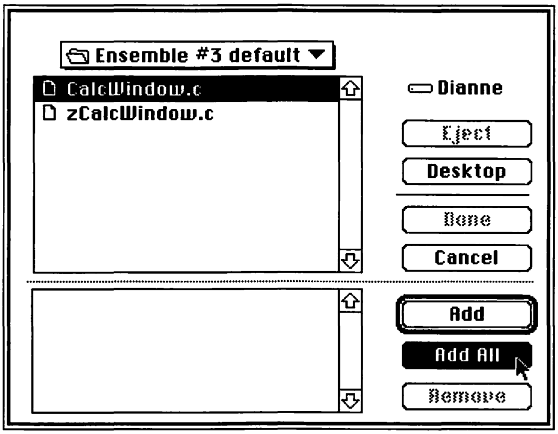


Figure 6-27
Selecting **Add All** in the **Add** dialog

3. After the **Add All** button is clicked, and the file names all show in the bottom window in the dialog, click the **Done** button, as shown in Figure 6-28. This will dismiss the dialog and cause all the files in the bottom window to be added to the project.

Notice that the **Add** dialog doesn't list any of the header files. THINK C will automatically add the header files that are needed by each source file as it is compiled.

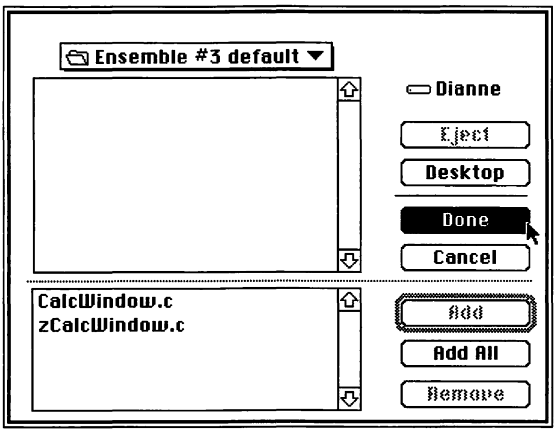


Figure 6-28
Clicking the **Done** button in the **Add** dialog

4. Figure 6-29 shows a portion of the project window, with all of the files for the Ensemble project added. At this point, none of the newly modified files has been compiled.

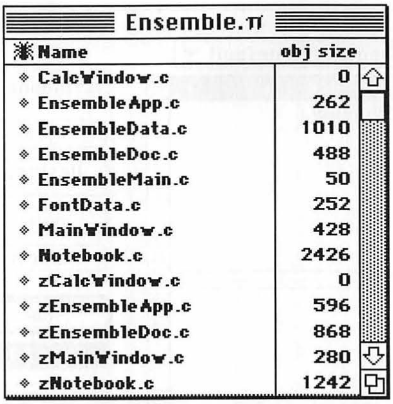


Figure 6-29
Ensemble.π project file showing all files added

5. The next step is to pull down the **Source** menu and choose the **Make** command. Use this command, rather than the **Bring Up To Date** command from the **Project** menu, because changes to files are not recorded in THINK C, unless they have been made with its internal editor or unless the files have never been compiled.

6. Choosing the **Make** command will cause THINK C to display a dialog, as shown in Figure 6-30. This dialog lists all the source files in the project. Click the **Use Disk** button, as shown in the figure. This will cause THINK C to scan the files for any changes that may have been made since it was last invoked. In this way, you can make THINK C subsequently recompile the modified files.

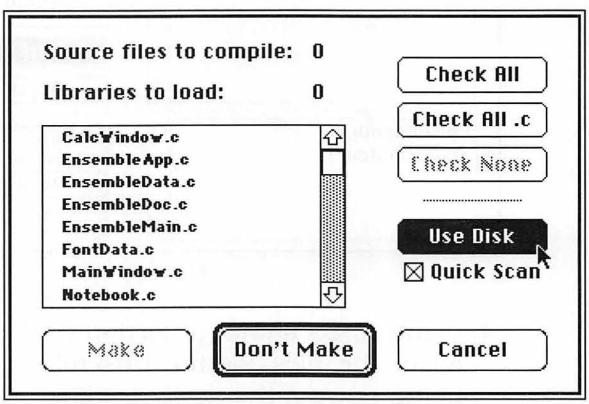


Figure 6-30
Clicking the **Use Disk** button to force THINK C to check whether files have been modified

7. After THINK C has scanned the files in the project and determined which ones need to be updated (indicated by check marks next to their names), you should click the **Make** button at the bottom of the dialog (which is enabled only if one or more files needs to be updated), as shown in Figure 6-31.

   If you are sure that one or more files needs to be recompiled, and THINK C fails to enable the **Make** button, click the **Quick Scan** checkbox to get rid of the check mark, and click the **Use Disk** button once again.

   Rather than just perform a quick scan of the files, THINK C will do a more thorough job and will undoubtedly check the files that need recompilation. In the worst case, where even this step fails, you can check files manually, by clicking at the left of their names in the **Make** dialog.

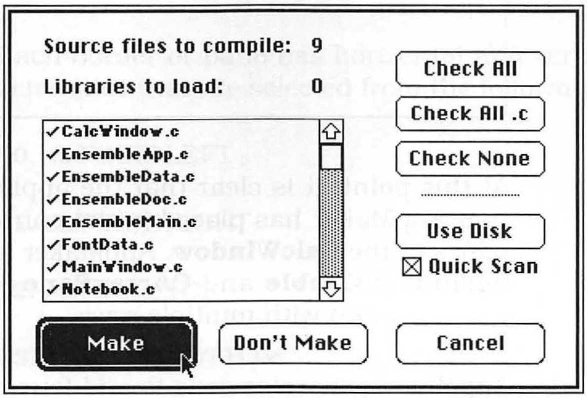


Figure 6-31
Clicking the **Make** button in the **Make** dialog

8. THINK C will commence compiling each of the files that it has determined need to be recompiled. In many cases, files that were not modified will require recompilation, because they refer to header files that have changed.

9. You are now ready to run the default version of the new Ensemble application. Pull down the **Project menu** and choose the **Run** option. Since the debugger is enabled by default, you will also have to click the **Go** button in the debugger's window. At this point, the initial **EditText** window will be in front. Resize and move that window so that it is below the **CalcWindow** that appears on the

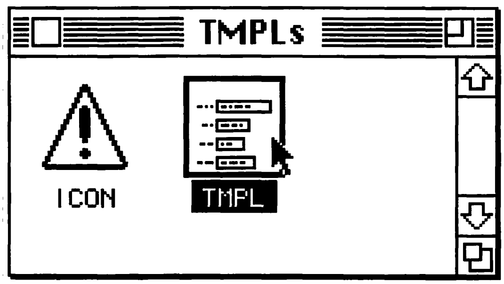
When AppMaker writes the parameters for border and pane elements, it chooses the most likely values for the sizing characteristics in the corresponding resource. In most cases, this choice will be correct; however, for the complex overlapping borders and panes of the **CalcWindow** design, we need to correct a few of the default-generated sizing values.

The process of modifying the generated resources is presented in a step-by-step fashion. In this case, we will be using the ResEdit program that is shipped with the THINK C version 5.0 product. This should be ResEdit version 2.1 or later. The process is as follows:

1. First, make a copy of the **TMPLs** file that is shipped with AppMaker version 1.5. We will not be altering the copy, but will hold it aside, in case we run into problems while performing the following steps. The original version of the file can easily be replaced with the copy.

2. Launch ResEdit, and locate and open the **TMPLs** file. You will see that ResEdit displays the existing resource types as a series of icons in a window, as shown in Figure 6-33. The **TMPLs** file only contains the **ICON** and **TMPL** resource, as pictured.

Figure 6-33
AppMaker v1.5 resources shown by ResEdit



3. Make sure that the '**TMPL**' resource icon is selected, and then pull down the **Edit** menu and choose the **Copy** command (or press Command-C). This will copy the **TMPL** resources to the clipboard.

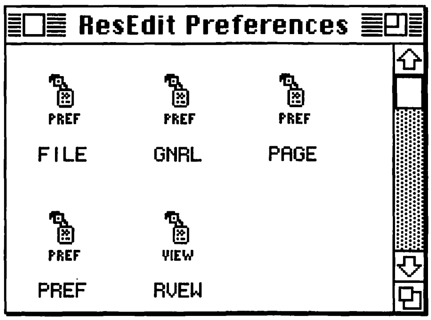
---

1. For more information on the sizing parameters of panes and other interface elements, see the *THINK C Object-Oriented Programming Manual*, version 5.0, pages 107 through 113.

4. Pull down the **File** menu and choose the **Quit** command to quit ResEdit. If ResEdit asks you whether you want to save changes before you quit, click No. You can discard the copy of the **TMPLs** file at this time.

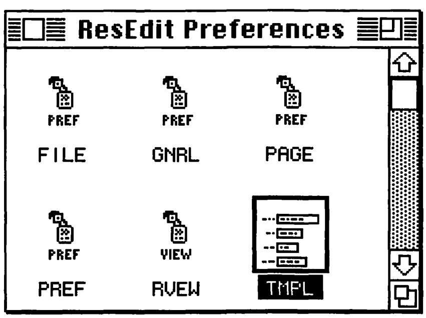
5. Locate the **ResEdit Preferences** file. It will be in the **Preferences** subfolder if you are running System 7.0 or directly in the **System** folder if you are running System 6.0.x. Make a copy of this file, and then double-click the original file to launch ResEdit. When it is launched, ResEdit will display the resources that currently exist in its **Preferences** file, as shown in Figure 6-34.

Figure 6-34
ResEdit Preferences file opened



6. Pull down the **Edit** menu and choose the **Paste** command (or press Command-V). This will paste the **TMPL** resources copied from the **TMPLs** file to the **ResEdit Preferences** file, as shown in Figure 6-35.

Figure 6-35
TMPL resources pasted



7. Pull down the **File** menu and choose the **Save** command. Then pull down the **File** menu and choose the **Quit** command to terminate the execution of ResEdit. At this point,

you can throw away the *copy* of the **ResEdit Preferences** file.

Before continuing, it is useful to explain what the preceding steps have accomplished. Because AppMaker creates quite a few resource types that are undefined in the version of ResEdit that ships with THINK C, it is necessary to provide ResEdit with templates that describe the various fields of these AppMaker-specific resources.

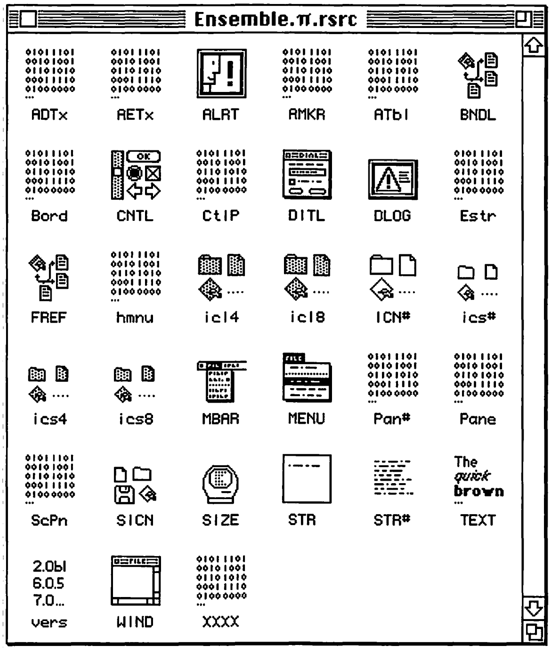
If you looked inside the set of **TMPL** resources, you would find that there is a template for a **Bord** resource, a **Pane** resource, an **AETx** resource, and many other resources. Each template describes the sizes and types of the various fields in the corresponding resource type. Therefore, with AppMaker's templates installed into ResEdit, you can inspect and modify the resources generated by AppMaker by looking at values in named fields, rather than by decoding hexadecimal values in ResEdit's general editor.

AppMaker's templates have to be installed into ResEdit's preferences file only once. If you have occasion to modify AppMaker-generated resources in the future, the templates you have just installed can be used without following the preceeding steps.

The next series of steps will describe the exact modifications to the resources that AppMaker generated for the **CalcWindow** window.

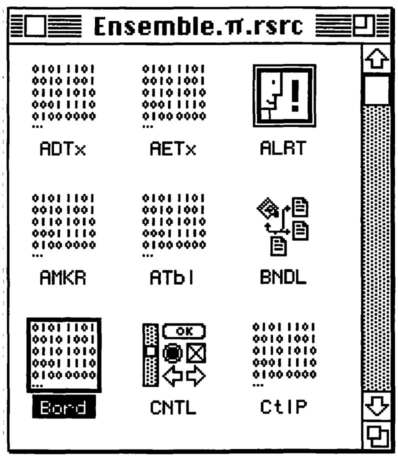
1. You should begin by duplicating the **Ensemble.π.rsrc** file and then launching ResEdit and opening the original **Ensemble.π.rsrc** file. You should see a window that looks something like that shown in Figure 6-36. As you can see, a great number of different resource types are generated, and all of these can be edited to modify the functionality of the Ensemble application. In the steps that follow, we will be modifying only a few of these resources; however, in the later stages of development, we will come back to ResEdit to personalize the application with its own Finder icon and '**BNDL**' resources, to make it a true stand-alone and unique application.

Figure 6-36
Ensemble.π.rsrc
resource file contents



2. With the ResEdit window open, showing the resources in the **Ensemble.π.rsrc** file, double-click on the icon whose type name is **Bord**, as shown in Figure 6-37. Note that the window has been resized in this figure, to show only a few of the resource types.

Figure 6-37
Selecting the **Bord**
resource type



3. Instead of double-clicking on the **Bord** resource, you can click to select it and then choose **Open** from ResEdit's **File** menu. In any event, once the resource category has been opened, you will see a list of **Bord** resources, as shown in Figure 6-38. In the next few steps, you will be modifying only two of these resources.

Figure 6-38
A list of the **Bord** resources in the Ensemble application

Bords from Ensemble.π.rsrc

ID	Size	Name
130	26	
131	26	
132	26	
133	26	
134	26	
135	26	
136	26	

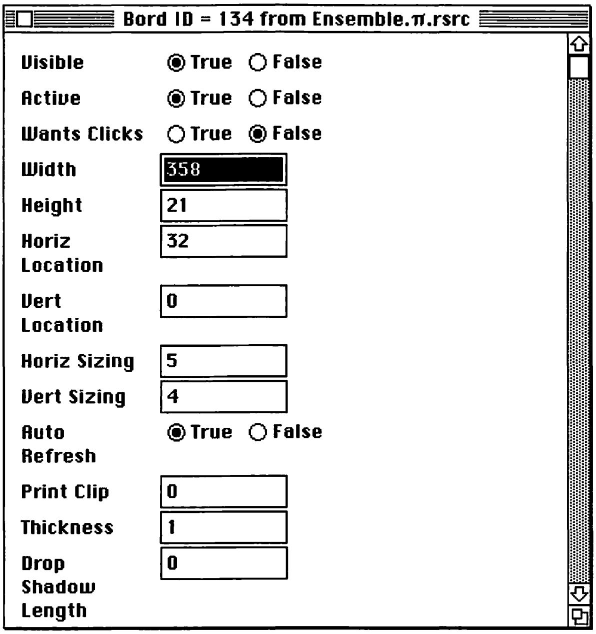
4. Double-click on the **Bord** resource whose ID is 134 in the list. This is the border that corresponds to item ② in Figure 6-1, the wide horizontal border. We discovered the number corresponding to this resource by looking in the generated code for the **zCalcWindow** superclass module, in the **IZCalcWindow** method. This code is as follows:

```
Rect4 = new CAMBorder;
Rect4->IViewRes ('Bord', 134, Rect3, supervisor);

List5 = NewList5 ();
List5->IViewRes ('ATbl', 137, Rect4, supervisor);
```

Notice that the **NewList5 (ATbl** 137) element is enclosed by the **Rect4** element in the last line of the code. This is proof that it is **Rect4**, and thus **Bord** 134, that we need to modify. Because you have installed AppMaker's **TMPL** (template) resources in ResEdit's **Preferences** file, you will be able to see all of the parameters that govern the position, size, appearance, functionality, and sizing for this resource. The settings for the **Bord** 134 resource are shown in Figure 6-39. Note that the figure shows radio buttons for the Boolean variables, such as **Visible** and **Active**, and has decimal values for the numeric parameters, such as the border's **Height** and **Width**. The two sizing parameters that you will need to modify are listed as **Horiz Sizing** and **Vert Sizing**. Change these to the values 5 and 4, as shown in the figure. This indicates that the border is elastic in the horizontal direction when the window is resized, but is fixed in the vertical direction. If you think about it, that makes sense. You want the horizontal border to always reach from the left to the right window border, but you don't want its vertical position to change when the window is resized. You will also need to click the **False** selection for the **Wants Clicks** item. Doing so disables mouse clicks in this pane. After changing these values, click in the window's close box.

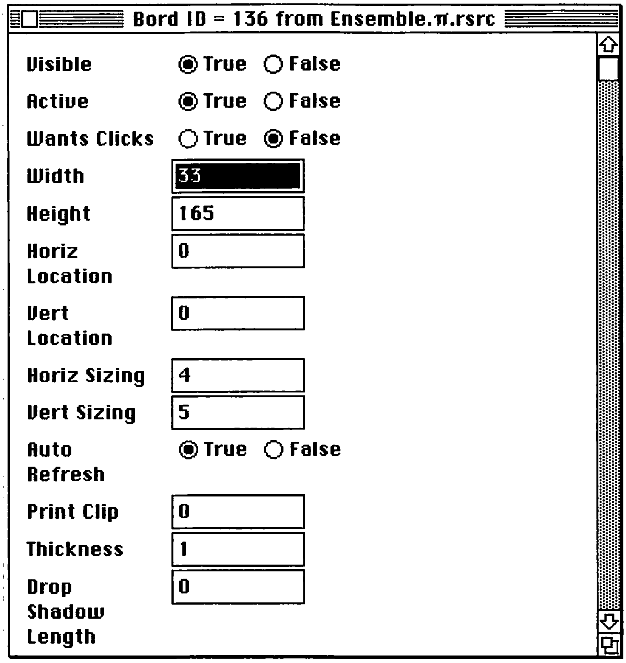
Figure 6-39
Settings for **Bord** 134



5. You should still have the list of **Bord** resources, as shown in Figure 6-38, on your screen. Double-click on the resource whose ID is 136. This is the vertical (row label pane) border (which you can verify by consulting the generated code, as described before). Its settings are shown in Figure 6-40. Change the **Horiz Sizing** and **Vert Sizing** parameters to 4 and 5, respectively. This will allow the border to stretch vertically, but remain fixed horizontally. The settings are shown in the figure. Also, click the **False**

radio button for the **Wants Clicks** setting. This will disable mouse clicks from being recognized in this pane. When you have made the indicated changes, click the window's close box, and then click the close box of the list of **Bord** resources. You should still have the window showing all of the resources in the **Ensemble.π.rsrc** file on your screen.

Figure 6-40
Settings for **Bord** 136



6. Scroll to the icon that shows the **ATbl** resource type, and double-click on that icon, as shown in Figure 6-41. This set of resources contains the parameters of all the **CTable**-oriented user interface elements. Among them are the two **CTable** panes (row and column labels), as well as the **CArrayPane** that occupies the majority of the window.

7. When you open the **ATbl** resource, you will see a list of all the tables that have currently been defined in the Ensemble application. The list is shown in Figure 6-42. Note that in addition to the three tables we defined in the **CalcWindow** design, there are two additional tables in the list. These correspond to the Font and Size tables in the Notebook dialog design, from Chapter 3.

Figure 6-41
ATbl resources selected

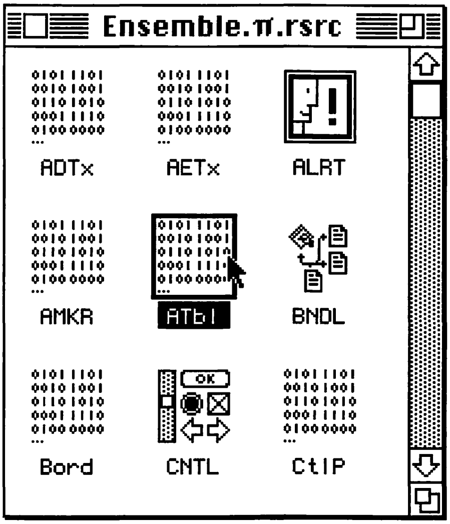
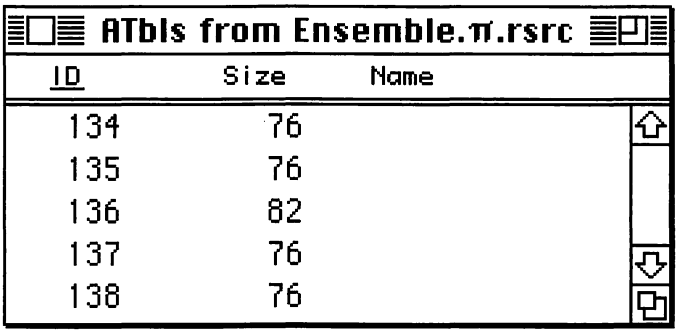


Figure 6-42
A list of the **ATbl** resources in the Ensemble application



8. Double-click on the **ATbl** resource whose ID is 137 to open it for the purpose of changing its sizing parameters. This is the table that corresponds to the column labels associated with item ③ in Figure 6-1, the table that will hold the column labels for the spreadsheet displayed in the **CalcWindow**. There are a great number of parameters associated with an **ATbl** resource, so we will just focus on the sizing parameters, but you may want to look at all the various settings that are available to be modified. Change the **Horiz Sizing** and **Vert Sizing** parameters to match the settings in Figure 6-43. This will enable the column labels to stretch or shrink horizontally, but remain fixed vertically. When you have made these changes, click in the close box of the window. You should still have the list of **ATbl** resources on your screen, as shown in Figure 6-42.

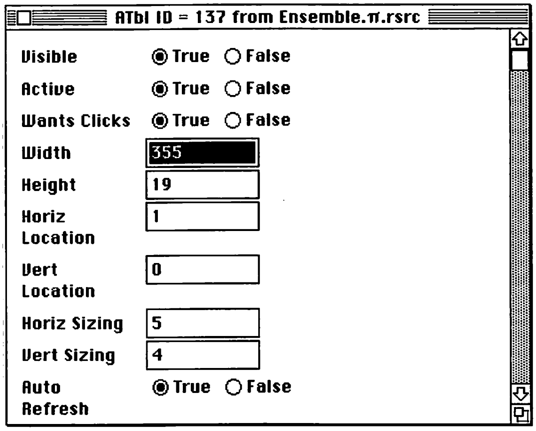


Figure 6-43
Settings for **ATbl** 137

9. Double-click (or select and then choose the **Open** command from ResEdit's **File** menu) the **ATbl** resource whose ID is 138. This is the table corresponding to the vertical (row labels) pane, identified as item ⑤ in Figure 6-1. The settings for the sizing characteristics for this table are shown in Figure 6-44, as 4 and 5 for the **Horiz Sizing** and **Vert Sizing** parameters, respectively. These settings enable the pane to stretch and shrink vertically, but remain fixed horizontally. When you are finished making these changes, close the window and also the list of **ATbl** resources by clicking in their respective close boxes.

10. The next series of steps will modify the sizing characteristics of several **Pane** resources. Scroll the window showing the resources in the **Ensemble.π.rsrc** file until the icon with the name **Pane** is visible, and then double-click this icon to open the list of these resources, as shown in Figure 6-45.

11. A window listing the **Pane** resources in the Ensemble application should appear, as shown in Figure 6-46.

12. Double-click on the **Pane** resource whose ID is 129 to open a window containing the settings of this resource, as shown in Figure 6-47. The figure shows the complete set of parameters for the **Pane** resources. Change the **Horiz**

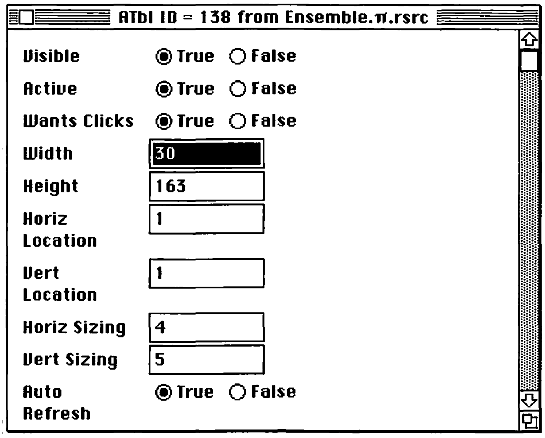


Figure 6-44
Settings for **ATbl** 138

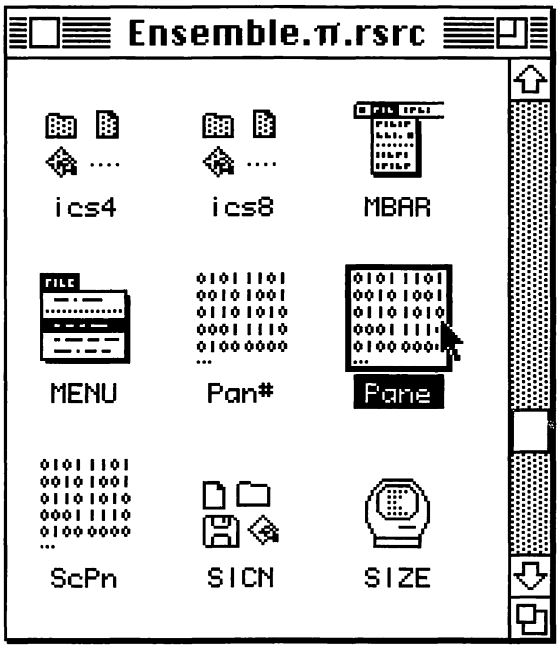


Figure 6-45
Selecting the list of **Pane** resources

Sizing and **Vert Sizing** parameters to 4 and 4, respectively, to force the pane to remain fixed in size and location when the window is resized. This pane corresponds to the "hole" at the upper left corner of the horizontal border and is identified as item ⑦ in Figure 6-1. When the changes have been made, close the window.

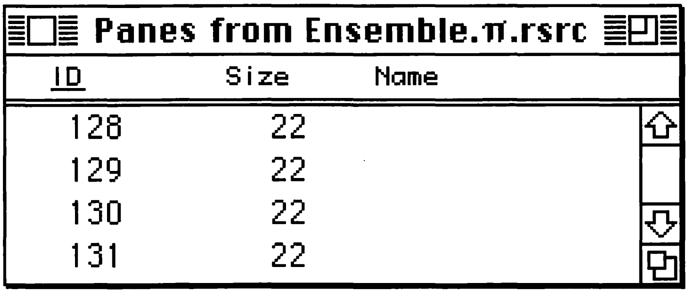


Figure 6-46
A list of **Pane** resources in the Ensemble application

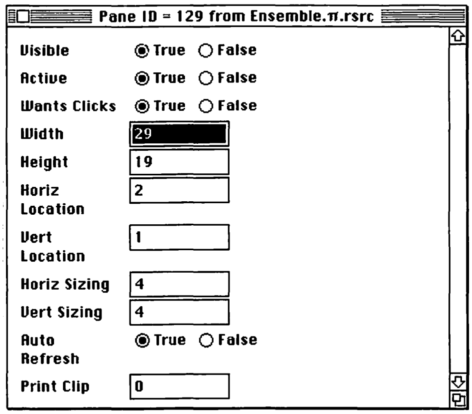


Figure 6-47
Settings for **Pane** 129

13. Double-click to open the **Pane** resource whose ID is 130, the settings of which are shown in Figure 6-48. Change the sizing settings for this pane to match those shown in the figure. The pane is shown as item ⑧ in Figure 6-1, and it fits in the upper right "hole" in the horizontal border. Both sizing parameters are set to 4, indicating that the pane is fixed horizontally and vertically in size and position. When you have completed these changes, close the window.

14. Double-click to open the **Pane** resource whose ID is 131, the settings of which are shown in Figure 6-49. Change the sizing settings for this pane to match those shown in the figure. The pane is shown as item ⑨ in Figure 6-1, fitting into the "hole" at the lower left corner of the window. The sizing, both horizontally and vertically, should be set to 4, indicating that the pane is fixed in size and position. When you have completed the changes to this pane, click

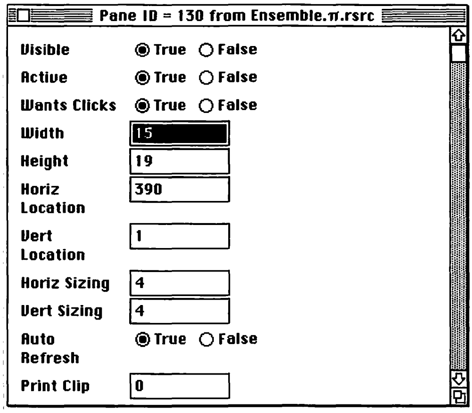


Figure 6-48
Settings for **Pane** 130

the close box to dismiss the window, and also click the close box to dismiss the list of **Pane** resources.

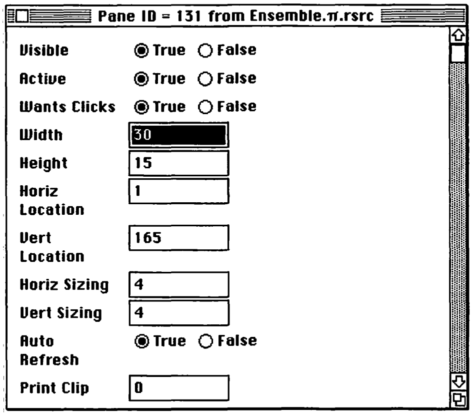


Figure 6-49
Settings for **Pane** 131

15. The final change modifies the **Line Width** parameter of the EditText pane that is identified as item ① in Figure 6-1. This is the Entry pane, represented by an **AETx** resource. Double-click on the **AETx** resource icon, as shown in Figure 6-50.

16. When the **AETx** resource list is opened, a window containing a list of these resources is displayed, as shown in Figure 6-51.

17. Double-click on the **AETx** resource whose ID is 135. This resource contains the settings for the Entry pane. We are

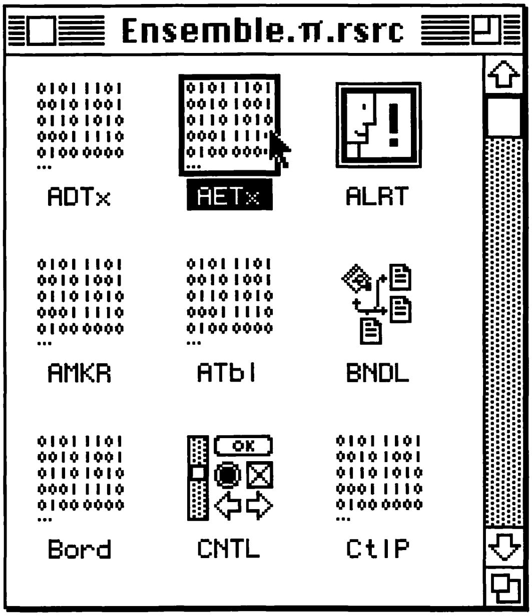


Figure 6-50
Opening the **AETx** resources

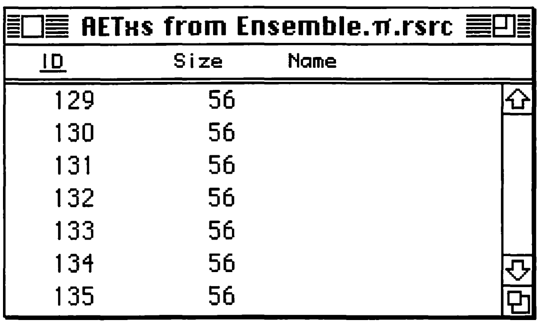


Figure 6-51
A list of **AETx** resources in the Ensemble

interested only in changing the **Line Width** parameter to the value 400, as shown in Figure 6-52. This setting will allow the Entry pane to scroll, after we have customized the code to enable scrolling for this EditText field. The value 400 is a little more than twice the width of the element, as indicated in the **Width** field of Figure 6-12.

When you have completed this change, close the window, and also close the window containing the list of **AETx** resources, by clicking in their respective close boxes.

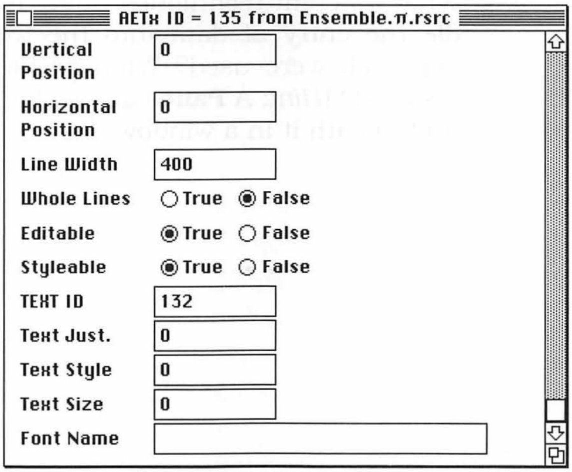


Figure 6-52
Settings for **AETx** 135

18. Pull down ResEdit's **File** menu and choose the **Save** command, to preserve the changes that have been made to the resources. After saving the changes, pull down the **File** menu and choose the **Quit** command to terminate execution of the ResEdit application.

After following the preceding steps, you can discard the copy of the **Ensemble.π.rsrc** file if all of the steps were completed successfully.

Exercises

1. Describe the rationale for using instances of the **CTable** class to represent the column and row labels in the worksheet window.

2. Explain what purpose *could* be served by the **CPane** elements that were placed in the top left, top right, and bottom left corners of the worksheet window. (*Hint*: Think about scrolling a very large worksheet.)

3. Explain how the various sizing parameters affect their corresponding elements in a window's design. For example, what would happen if the sizing characteristics of the column label border weren't modified?

4. What different techniques would have to be used to handle the entry of data into the worksheet if a different approach were used? What problems would have to be resolved? (*Hint*: A **Pane** can overlay and obscure anything underneath it in a window.)[1]

1. This exercise has far-reaching consequences in the overall design of the application and would be an excellent extra-credit project. The goal would be to provide the means for handling in-cell entry and editing of worksheet data.

Chapter 7

Examining the CalcWindow Code

This chapter describes the modifications to the Ensemble applications code generated by AppMaker after adding the CalcWindow user interface elements. It should be apparent that none of the subclass files, whose names *do not* begin with the letter **z**, were modified. Thus, their contents are safe. Only the superclass files, whose names *do begin* with the letter **z**, have been regenerated. This is a tremendous help, as we will never (well... hardly ever) have to make any changes to the superclass files, and they are available to be reconstructed at AppMaker's will.

In the course of generating new files, after the **CalcWindow** was added to the user interface in Chapter 6, AppMaker generated two new source files and their companion header files:

- **CalcWindow.c** contains the subclass methods that override and supplement the methods generated in the superclass and will also be the target file for the custom code that is described in Chapter 8.

- **CalcWindow.h** contains the class declarations for the subclasses defined in the **CalcWindow.c** source file.

- **zCalcWindow.c** contains the superclass methods, that implement the initialization and default behavior of the new user interface elements.

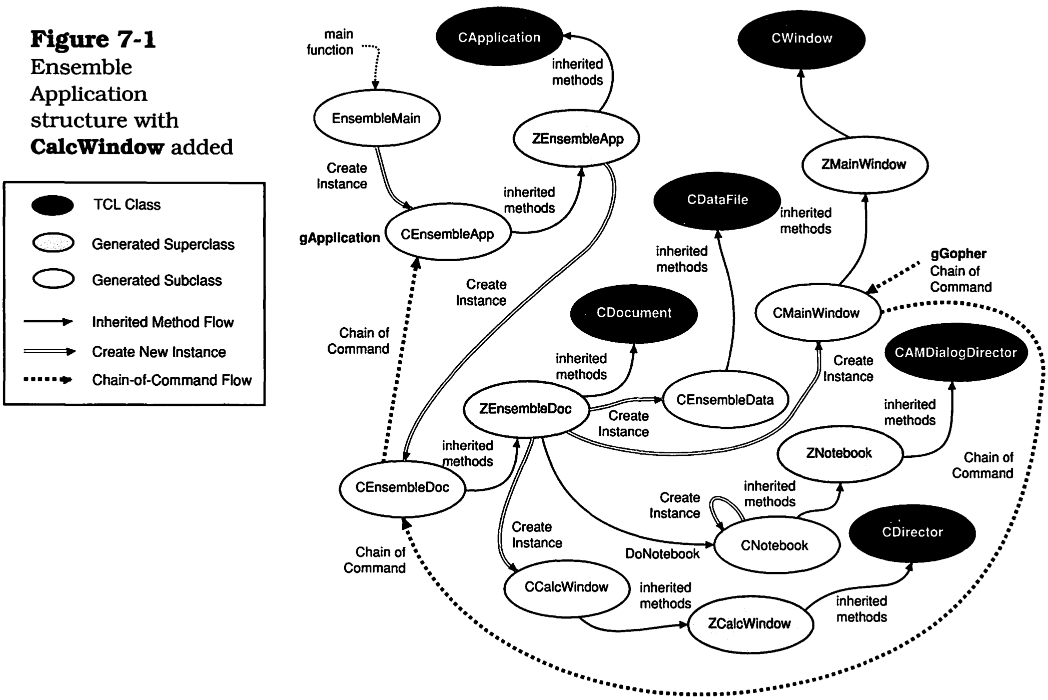
- **zCalcWindow.h** contains the class declarations for the superclasses defined in the **zCalcWindow.c** source file.

All of the superclass source and header files have been regenerated; however, we will only describe the differences in them brought about by the addition of the new user interface elements.

The CalcWindow's Code Structure

As indicated in Chapter 4, the best way to see how the newly generated code files are associated with the other parts of the Ensemble application is to look at the diagram shown in Figure 7-1 and compare it with Figure 4-1.

Figure 7-1 Ensemble Application structure with **CalcWindow** added



It is clear from the figure that the **CCalcWindow** subclass instance is created from the **ZEnsembleDoc** class—specifically, by the **BuildWindows** method. Its superclass, **ZCalcWindow**, is a subclass of the TCL's **CDirector** class, as is the **CDocument** class (although that relationship isn't shown in the diagram). It is important to note that all subsidiary windows will be created from the **ZEnsembleDoc** class and will inherit their behavior from the **CDirector** class. The structure of all AppMaker-generated applications have only one window that is managed by the TCL's **CDocument** class. This implies that, because the document creates the **CDataFile** class instance, all windows will share a common file. If you need to open a different type of file for each window, it is better to design the application so that it has a single window, with multiple instances of that window (created by the **New** and **Open** commands in the application's **File** menu). This model is perfect for our purposes. The modified classes and methods are shown in Table 7-1.

Table 7-1
Generated code changes for the **CalcWindow** user interface element

Class	Method	Description
ZEnsembleDoc	BuildWindows	includes code to create an instance of **CCalcWindow** and initialize it
ZCalcWindow	IZCalcWindow	contains all the code to create and initialize all of the interface elements in the **CalcWindow** defined in AppMaker
ZCalcWindow	various	methods for creating the **CTable** and **CArrayPane**, as well as the *hole filler* panes. Meant to be overridden
CCalcWindow	ICalcWindow	calls the inherited **IZCalcWindow** initialization method, and contains post initialization code
CCalcWindow	various	methods to create, initialize and supply the cell text for each of the **CTable** panes (e.g., **NewList5**, **IViewTemp**, and **GetCellText**)
CCalcWindow	various	creating, initialization, draw, and other methods for each of the *hole filler* panes (e.g., **NewUser6**, **IViewTemp**, and **Draw**)
CCalcWindow	DoEnterButton DoCancelButton	methods to handle mouse clicks on the respective buttons
CCalcWindow	UpdateMenus	calls inherited method
CCalcWindow	DoCommand	recognizes mouse commands for the **Enter** and **Cancel** buttons
CCalcWindow	ProviderChanged	intercepts **ProviderChanged** messages from the **CTable** and **EditText** instances

The classes and methods listed in the table provide only the default appearance shown in Figure 6-32 when the generated code was compiled and executed. It is this default functionality that the following sections discuss.

Newly Generated Code in ZEnsembleDoc

In the newly generated code, none of the superclass modules except **zEnsembleDoc**.c has been modified, even though the other modules were regenerated. There is no harm to this, and user interface modifications could very well affect other modules.

BuildWindows Method Code

The **ZEnsembleDoc** class's **BuildWindows** method has been modified to create and initialize the new **CalcWindow** that we defined. The new version of this code is as follows:

```
void ZEnsembleDoc::BuildWindows (void)
{
   CWindow *mainWindow;
   CDirector *subWindow;

   mainWindow = new CMainWindow;
   itsWindow = mainWindow;
   ((CMainWindow *)mainWindow)->IMainWindow (this, itsData);
   itsMainPane = ((CMainWindow *)mainWindow)->itsMainPane;

   subWindow = new CCalcWindow;
   ((CCalcWindow *)subWindow)->ICalcWindow (this, itsData);
}
```

The foregoing code creates the **MainWindow**, in the same way as shown in Chapter 2, on page 34. In addition, it creates a new instance of **CCalcWindow** and calls its initialization method.

If additional windows are added to the application, the **BuildWindows** method will be enhanced to create and initialize these as well.

Newly Generated Code in ZCalcWindow

The **zCalcWindow.c** file has been generated to contain the initialization method and other superclass methods that establish the default appearance and functionality of the **CalcWindow** window.

IZCalcWindow Method Code

The code to create and initialize all of the interface elements that form a part of the **CalcWindow** design is contained in the **IZCalcWindow** method. This is a rather large method, because of all the elements we defined. The code is as follows:

*The **IZCalcWindow** method creates and initializes all the user interface elements in the **CalcWindow***

```
void ZCalcWindow::IZCalcWindow(CDirectorOwner *aSupervisor)
{
   CView   *enclosure;
   CBureaucrat *supervisor;
   CSizeBox *aSizeBox;

   inherited::IDirector (aSupervisor);
   itsWindow = new CWindow;
   itsWindow->IWindow (CalcWindowID, FALSE, gDesktop, this);
   enclosure = itsWindow;

   supervisor = this;
   Rect1 = new CAMBorder;
   Rect1->IViewRes ('Bord', 132, enclosure, supervisor);

   EntryField = new CAMEditText;
   EntryField->IViewRes ('AETx', 135, Rect1, supervisor);

   Rect3 = new CAMBorder;
   Rect3->IViewRes ('Bord', 133, enclosure, supervisor);

   Rect4 = new CAMBorder;
   Rect4->IViewRes ('Bord', 134, Rect3, supervisor);

   List5 = NewList5 ();
   List5->IViewRes ('ATbl', 137, Rect4, supervisor);

   User6 = NewUser6 ();
   User6->IViewRes ('Pane', 129, Rect3, supervisor);
   User7 = NewUser7 ();
   User7->IViewRes ('Pane', 130, Rect3, supervisor);
   Rect8 = new CAMBorder;
   Rect8->IViewRes ('Bord', 135, enclosure, supervisor);
   Rect9 = new CAMBorder;
   Rect9->IViewRes ('Bord', 136, Rect8, supervisor);
   List10 = NewList10 ();
   List10->IViewRes ('ATbl', 138, Rect9, supervisor);
   User11 = NewUser11 ();
   User11->IViewRes ('Pane', 131, Rect8, supervisor);
   ScrollPane12 = new CScrollPane;
   ScrollPane12->IViewRes ('ScPn', 134, enclosure, supervisor);
```

IZCalcWindow *method code (concluded)*

```
    List15 = NewList15 ();
    List15->IViewRes ('ATbl', 136, ScrollPane12, supervisor);
  ScrollPane12->InstallPanorama (List15);
  CellNumLabel = new CAMStaticText;
  CellNumLabel->IViewRes ('AETx', 134, enclosure, supervisor);
  EnterButton = new CAMButton;
  EnterButton->IViewRes ('CtlP', 144, enclosure, supervisor);
  CancelButton = new CAMButton;
  CancelButton->IViewRes ('CtlP', 145, enclosure, supervisor);
  aSizeBox = new CSizeBox;
  aSizeBox->ISizeBox (enclosure, supervisor);
}
```

The **IZCalcWindow** code creates and initializes each of the user interface elements in the **CalcWindow**. Borders are given names beginning with the word **Rect**, lists begin with **List**, and user items (such as *hole filler* panes) begin with **User**. The single **CScrollPane** instance carries a name beginning with **ScrollPane**, and the **Enter** and **Cancel** buttons are named **EnterButton** and **CancelButton**, respectively. The **CAMEditText Entry** field is named **EntryField**, and the **CAMStaticText** item holding the words **Cell Num** has been named **CellNumLabel**.

__Note__: The author made a list of each resource type and ID and then examined them with ResEdit to determine their generated settings.

You'll notice that the elements are numbered in ascending sequence, regardless of their types. Also, each is created from the parameters in a template of a particular type and resource ID. The **IViewRes** method contains the type name and resource ID of the resource template. These correspond to the template type names and IDs whose sizing characteristics were modified in Chapter 6.

Several of the **IViewTemp** methods are overridden in the **CCalcWindow** class. We'll be looking at these methods shortly.

NewList$_i$ Method Code

AppMaker generates code in the superclass to create each of the **List**$_i$ elements. The sole purpose of generating this code is so that it can be overridden by the corresponding subclass method. Notice that the superclass method creates an instance of the **CAMTable** class while the subclass override method creates a new subclass of that table instance (see the subclass code for **NewList5** on page 177). An example of the superclass code is the following:

```
CAMTable *ZCalcWindow::NewList5(void)
{
   CAMTable *theList;

   theList = new CAMTable;
   return (theList);
}
```

This code creates the **List5** element (the horizontal cell label **CAMTable** instance) and returns the instance to the caller (**IZCalcWindow**). The method is overridden in the subclass, as will be shown. The **zCalcWindow.c** module contains nearly identical code for the other two lists; the only difference being in the creation of the **List15** element, which is created as a **CAMArrayPane** object.

NewUser$_i$ Method Code

The *hole filler* panes are created in much the same way as the lists. A method is provided for each, so that it can be overridden by the subclass if desired.

An example of one of these methods is the following:

```
CPane *ZCalcWindow::NewUser6(void)
{
   CPane   *pane;

   pane = new CPane;
   return (pane);
}
```

Each *hole filler* pane is created in a fashion identical to that shown in the preceding code. The pane there is initialized by a resource named **Pane129**, which corresponds to the top left *hole filler*, shown as item ⑦ in Table 6-1.

Remember, the creation methods merely create the object instance and return it to the caller (**IZCalcWindow**).

UpdateMenus Method Code

The default code for the **UpdateMenus** method merely calls the inherited method:

```
void ZCalcWindow::UpdateMenus(void)
{
   inherited::UpdateMenus ();
}
```

Code is also generated in the subclass for this method; however, by default, it also merely calls the inherited method, as we will show later.

DoCommand Method Code

The code for the superclass's **DoCommand** method, also very simple, is provided as a method that the subclass can override:

```
void ZCalcWindow::DoCommand(longtheCommand)
{
   switch (theCommand)
   {
      default:
            inherited::DoCommand (theCommand);
         break;
   }
}
```

As is apparent, only the default case is handled, by calling the inherited method to handle the command.

Newly Generated Code in CCalcWindow

The generated code in the **CalcWindow.c** module provides very little additional functionality, but serves as a framework for customizing the behavior of the **CalcWindow** user interface element. Many of the methods that override or supplement those in the superclass merely call the inherited method. Thus, the default execution functionality of the window is provided almost entirely by the code generated into the superclass. This section discusses the subclass methods, which will be the basis for all the custom code that will be added and described in Chapter 8.

ICalcWindow Method Code

When the **ZEnsembleDoc** class's **BuildWindows** method first creates the **CalcWindow** instance, it calls the **ICalcWindow** method to perform the initialization for this new window. The code for the **ICalcWindow** method saves the handle to the **CEnsembleData** instance (**theData**), calls the inherited **IZCalcWindow** method, and sends the **gDecorator** a message to stagger the new window with respect to the other windows currently on the screen.

The method also serves as a placeholder for additional custom code that we will be adding in the next chapter. The code for the **ICalcWindow** method is as follows:

```
void CCalcWindow::ICalcWindow(CDirector *aSupervisor,
                              CEnsembleData *theData)
{
   itsData = theData;
   inherited::IZCalcWindow (aSupervisor);
   gDecorator->StaggerWindow (itsWindow);
   // any additional initialization for your window
}
```

Notice that AppMaker has inserted a comment in the generated code, indicating where additional initialization code can be placed. This is one of the most useful features of the generated code. Such comments are sprinkled liberally throughout the code, to aid you in placing modifications and custom additions.

$List_i$ IViewTemp Method Code

Each of the list elements is accompanied by three generated methods in the subclass: an **IViewTemp**, a **GetCellText**, and a **NewList$_i$** method. The code for the **IViewTemp** method is as follows:

```
void CList5::IViewTemp (CView *anEnclosure,
                        CBureaucrat *aSupervisor, Ptr viewData)
{
   inherited::IViewTemp (anEnclosure, aSupervisor, viewData);
   // any additional initialization for your subclass
   AddRow (4, 0);   // e.g., add 4 rows at the beginning of the list
}
```

List GetCellText Method Code

The **GetCellText** method is called by the TCL's **CTable** class whenever the contents of a list cell need to be redrawn. The **GetCellText** code for the **CList5** class instance is as follows:

```
void CList5::GetCellText (Cell aCell,
                          short availableWidth, StringPtr itsText)
{
   // replace with your own code, which uses the cell coordinates
   // to access your private data structures;
   // then convert the cell data to a Str255
   switch (aCell.v) {
      case 0:
            CopyPString ("\pOne", itsText);
         break;
      case 1:
            CopyPString ("\pTwo", itsText);
         break;
      case 2:
            CopyPString ("\pThree", itsText);
         break;
      default:
            CopyPString ("\pInfinity", itsText);
         break;
   }
}
```

Note that AppMaker has once again generated comments which indicate that the code included in the **GetCellText** method is only an example of what is needed and should be replaced with code pertinent to your application. We will be replacing all of this code in the next chapter.

List NewList$_i$ Method Code

When a new list element is created, AppMaker generates code to create this element both in the superclass and the subclass, so that the code can be overridden if desired.

The code to create a new list element is rather simple; however, being able to override it allows you a lot of flexibility in how the list is created. It also gives you the opportunity to add code to perform related tasks inside the creation method. The code for the **NewList5** method is as follows:

```
CAMTable *CCalcWindow::NewList5(void)
{
   CList5   *theList;
   theList = new CList5;
   return (theList);
}
```

The superclass method for **NewList5** creates an instance of **CAMTable**, as shown on page 173. The corresponding overriding method in the subclass creates a unique class instance. This, in turn, is a subclass of **CAMTable** as illustrated by the following class declaration taken from the **CalcWindow.h** header file:

```
class CList5 : public CAMTable
{
public:
   void IViewTemp(CView *anEnclosure,
                  CBureaucrat *aSupervisor,
                  Ptr viewData);   // is override
   void GetCellText(Cell aCell,
                  short availableWidth,
                  StringPtr itsText);   // is override
};
```

The foregoing declaration for **CList5** defines it as a direct descendant of the AppMaker library class **CAMTable**, which, in turn, is a direct descendant of the TCL's **CTable** class. Generating a unique class name for each list (or table) is necessary, so that each type of list can have its own **IViewTemp** and **GetCellText** methods.

The **CalcWindow.c** module also contains **IViewTemp**, **GetCellText**, and **NewList$_i$** methods for lists **CList10** and **CList15**. The direct ancestor of the **CList15** class is **CAMArrayPane**, rather than **CAMTable**.

Incidentally, AppMaker interposes its own library classes in between many of its generated classes and the TCL, to provide text styles for almost every element. For example, in Chapter 6, the text style for the main spreadsheet table (shown in the code as **CList15**) was changed to 9-point Geneva, rather than the **System** font (12-point Chicago). The

IViewTemp method inherited from most AppMaker library classes initializes the text size, style, and justification of the corresponding elements. Therefore, when the generated code calls the inherited **IViewTemp** method first, it is allowing AppMaker's library method to set up the specified text style information from the resource template.

User IViewTemp Method Code

For each *hole filler* pane that is created, AppMaker generates a class beginning with the word **User**, with a number appended to make it unique. The **IViewTemp** initialization method for one such pane is as follows:

```
void CUser6::IViewTemp (CView *anEnclosure,
                        CBureaucrat *aSupervisor,
                        Ptr       viewData)
{
   inherited::IViewTemp (anEnclosure, aSupervisor, viewData);
   // any additional initialization for your subclass
}
```

Note that AppMaker has generated a comment in this method, after the call to the inherited method, to the effect that if additional initialization is appropriate for this item, the code can be inserted at that location.

User Draw Method Code

AppMaker also generates a **Draw** method for **User** panes. The code shown for this method is only an example of what might be needed; it is benign and needn't be changed if nothing special is required to draw the contents of the element. Sample code for one such **User** element is as follows:

***Draw** method code for **CUser6** class (beginning)*

```
void CUser6::Draw (Rect *area)
{
   // replace with your own code which draws the pane
   // note that 'area' is usually ignored; it has no relationship
   // to the size of the pane; it merely indicates what portion
   // (in QuickDraw coordinates) of the pane needs to be drawn

   Rect   theFrame;
   PenState savePen;
```

***Draw** method code for **CUser6** class (concluded)*

```
    GetPenState (&savePen);
    PenNormal ();
    FrameToQDR (&frame, &theFrame);
    SetPenState (&savePen);
}
```

The preceding code calls the **GetPenState** toolbox routine to get information on the current pen state, including the pen location, size, transfer mode, and pattern. The **PenNormal** toolbox call resets the pen state to the initial (default) settings. The **FrameToQDR** method (located in the **CPane** class) converts the instance variable **frame** (describing the top left and bottom right coordinates of the pane) from frame coordinates to Quickdraw coordinates.

The final statement calls the **SetPenState** toolbox routine to reset the pen state to the value it had before the **Draw** method was entered. As generated, the method performs no useful function; however, if you wished to draw something in the pane, you would insert the appropriate code right after the call to the **FrameToQDR** method. We will not be modifying this code, as the panes are simply *hole fillers* at this point. They could serve other useful functions, however, so we elected to create panes, rather than just leave the corresponding border areas empty.

User NewUser$_i$ Method Code

The final method generated for each user item creates the method instance and returns it to the caller. The code for the **NewUser6** method is as follows:

```
CPane *CCalcWindow::NewUser6(void)
{
    CUser6   *pane;

    pane = new CUser6;
    return (pane);
}
```

Each **User** element is a direct descendant of the TCL's **CPane** class, as shown by the class declaration for the **User6** element, taken from the **CalcWindow.h** module:

```
class CUser6 : public CPane
{
public:
   void IViewTemp(CView *anEnclosure,
                  CBureaucrat *aSupervisor,
                  Ptr viewData);   // is override

   void Draw(Rect *area);    // is override
};
```

In addition to the **IViewTemp**, **Draw**, and **NewUser6** methods generated for the **User6** element, AppMaker has also generated nearly identical declarations and methods for the **User7** and **User11** elements.

UpdateMenus Method Code

AppMaker's generated code for the **UpdateMenus** method merely calls the inherited **UpdateMenus** method; however, if additional changes to the state of the menus on the screen are needed, then, when the **CalcWindow** is active, this method provides an appropriate place to insert the necessary code.

The default-generated code is as follows:

```
void CCalcWindow::UpdateMenus(void)
{
   inherited::UpdateMenus ();
}
```

In the next chapter, we will be enhancing this code to disable the **Close** command in the **File** menu when the **CalcWindow** is active.

DoCommand Method Code

When the user chooses a command from one of the application's menus, or if a "click command" is assigned to a button or other item, the **DoCommand** method associated with the current **gGopher** will be called with the command number parameter. The generated code for the **CCalcWindow** class's **DoCommand** method is as follows:

```
void CCalcWindow::DoCommand(long theCommand)
{
   switch (theCommand)
   {
      case cmdEnterButton:
      {
         DoEnterButton ();
         break;
      }
      case cmdCancelButton:
      {
         DoCancelButton ();
         break;
      }
      default:
      {
         inherited::DoCommand (theCommand);
         break;
      }
   }
}
```

AppMaker assigns "click commands" to all buttons as a standard procedure, so these were created for the **Enter** and **Cancel** buttons in the resource templates for those elements. When the user clicks on either of these buttons, the TCL will generate a **DoCommand** message containing the command number of the button that is clicked.

When the **DoCommand** method is called with either the **cmdEnterButton** or the **cmdCancelButton** command, the method will invoke the appropriate corresponding method.

The default-generated code for both of these methods is as follows:

```
void CCalcWindow::DoEnterButton (void)
{
}

void CCalcWindow::DoCancelButton (void)
{
}
```

As you can see, both methods are completely empty; however, we will be adding code to them in the next chapter.

ProviderChanged Method Code

The collaboration mechanism, defined as part of the TCL, was described in full in Chapter 4. This mechanism is used by a range of providers, including the descendants of the TCL's **CTable** class.

As explained in Chapter 4, if a selection changes, a **BroadcastChange** message is sent to the list instance, which inherits the functionality of the corresponding method in the **CCollaborator** class; however, in addition, the **CBureaucrat** class overrides this method and also sends a **ProviderChanged** message to the table's supervisor, which, in the case of our tables (main spreadsheet, row labels, and column labels), will cause the **CCalcWindow** instance's **ProviderChanged** method to be invoked. The message includes the instance handle of the provider that issued the **BroadcastChange** message, the reason for the broadcast, and any other appropriate data, which are addressed via a pointer.

AppMaker generates a **ProviderChanged** method for each window or dialog in the application. We did not need to use this method in the **CMainWindow** class, but did in the **CNotebook** and will in the **CCalcWindow** class.

The generated code for the **ProviderChanged** method is as follows:

ProviderChanged *method code (beginning)*

```
void CCalcWindow::ProviderChanged(CCollaborator *aProvider,
                                  long    reason,
                                  void*   info)
{
    if (aProvider == List5) {
        if (List5->HasSelection ()) {
            // perhaps activate some buttons
        } else {
            // perhaps deactivate
        }
    }
    if (aProvider == List10) {
        if (List10->HasSelection ()) {
            // perhaps activate some buttons
        } else {
```

ProviderChanged
method code
(concluded)

```
            // perhaps deactivate
        }
    }
    if (aProvider == List15) {
        if (List15->HasSelection ()) {
            // perhaps activate some buttons
        } else {
            // perhaps deactivate
        }
    }
}
```

This code tests whether the **aProvider** parameter is one of the lists (**List5**, **List10**, or **List15**) and then checks whether a cell is selected for that table. No action code is included; however, we will be customizing the method in the next chapter to handle selections in the main spreadsheet table (**List15**).

Exercises

1. Figure 7-1 shows the "chain of command" when the **MainWindow** instance is frontmost on the screen. Describe the "chain of command" if the **CalcWindow** is frontmost.

2. Explain why each of the column labels, row labels, and main worksheet lists must be a separate subclass of the TCL's **CTable** class.

3. Compare the features of the TCL's **CTable** class with the built-in Macintosh **List Manager**.

4. What mechanism causes the **GetCellText** methods of the column, row, and worksheet lists to be invoked?

5. Explain the rationale for having both superclass and subclass **NewList**$_i$ and **NewUser**$_i$ methods? (*Hint*: Look at the **IZCalcWindow** method for a clue to this organization of classes and methods.)

6. What functions are performed by the **Enter** and **Cancel** buttons in the worksheet window? What methods will need to be enhanced to implement these functions? Outline the features of the code to do so.

7. The worksheet's column and row label lists are implemented as subclass instances of AppMaker's **CAMTable** class. In what way does the user interact with these user interface elements? How must the code in the **Provider-Changed** method, as generated by AppMaker, be modified to support the required interactions? (*Hint*: Examine the steps for modifying the **Bord** resource panes that enclose these lists, as described in Chapter 6.)

8. Assuming that you have decided to draw a pattern in the empty panes at the top left, top right, and bottom left corners in the worksheet, how would you implement your intentions, based upon the classes and methods shown in this chapter? Modify the appropriate methods to do so.[1]

1. This is a relatively simple task, but it will require some thought and knowledge of the Macintosh toolbox routines. It could be assigned as an extra-credit project.

Chapter 8

Customizing the Worksheet Code

This chapter describes the classes and methods that have been customized to implement the full functionality of the **CalcWindow** interface component. In the course of describing the implementation of a functional spreadsheet, a great amount of detail was required. Such detail is justified in order to present the complete, step-by-step documentation of this nontrivial addition to the Ensemble application.

Table 8-1
Customized methods to implement the I/O for EditText and spreadsheet data

Class	Method	Description
CEnsembleData	IEnsembleData	Create CCluster to hold spreadsheet data
CEnsembleData	ReadData	Rewritten to handle text and spreadsheet data
CEnsembleData	ReadWSEntries	New method to read spreadsheet entries
CEnsembleData	WriteData	Rewritten to handle text and spreadsheet data
CEnsembleData	WriteWSEntries	New method to write spreadsheet entries
CEnsembleData	DisposeData	Disposes of EditText and spreadsheet data entries
CEnsembleData	GetCluster	Access method to return spreadsheet cluster instance

Table 8-1 lists the methods and modifications that support input/output of both the EditText and worksheet data in a shared file.

Customizing the CEnsembleData Code

Chapter 5 contains descriptions of all of the existing methods in the **CEnsembleData** class. This class is responsible for performing all of the physical input/output for the application. It contains methods to open, close, save, save as, and revert to a previous version of a file. It also contains methods to initialize the class and dispose of all the data. The **CEnsembleData** class is created to support the **CEnsembleDoc** class, which contains the methods that are called by the TCL to create a new document, open an existing document, and read from and write to the application's windows.

Because our new user interface model has two windows, each holding a different type of data, it becomes necessary to be able to perform all of the input/output operations on a composite file. There are very few methods that need revision to support this new concept, and the modifications are straightforward.

Modifying the Initialization code

Because all of the input/output for the application is carried out in the **CEnsembleData** class, it is natural for the data to be owned by this class. The data will need to be accessed by other classes, but the **CEnsembleData** class owns both the EditText and the worksheet data.

IEnsembleData Code

The code for initializing the **CEnsembleData** instance has been modified to create a **CCluster** to hold the worksheet data. The modified code is as follows:

```
void CEnsembleData::IEnsembleData(CDocument *theDocument)
{
    inherited::IDataFile ();
    hasFile = FALSE;
    itsDocument = theDocument;
    // your application-specific initialization
    itsEditTextData = NULL;
    itsCluster = new CCluster;
    itsCluster->ICluster();
}
```

The code is not very different from what was shown in Chapter 5, on page 98. The main difference is that a new instance variable has been defined to contain an instance of **CCluster**—a data collection class in the TCL—and the cluster is initialized.

The **itsCluster** instance variable has been added to the **CEnsembleData** class declaration in the **CEnsembleData.h** header file as a *protected* variable, along with the existing **itsEditTextData** variable.

Modifying the Input/Output Code

Because of the addition of the new window (**CalcWindow**), the input/output code must be modified to make provision for storing both worksheet and text data in the same file. This is quite easy, and you'll find the custom code additions straightforward and simple. The modified methods (and new, custom methods) are listed in Table 8-1. The following subsections discuss the new code.

ReadData Method Code

The **ReadData** method has been substantially changed, to reflect the fact that three different types of data are stored in the single data file owned by the **CEnsembleData** class. The three types of data are text font information (font, style, size, justification), text data, and worksheet cell data.

The file format has been completely changed to make provision for the existence of either or both of the text or worksheet data. Both need not be present, but both are accommodated.

The code for the new **ReadData** method is as follows:

***ReadData** method code (beginning)*

```
void CEnsembleData::ReadData(void)
{
   long  textLength, WSEntryCt;
   fontInfo  theFontInfo;

   //
   // modified to handle both the EditText and Worksheet data
   // in the file. The file format is:
   //
```

***ReadData** method code (concluded)*

```
    //  char. pos.      description
    //  -------------   ------------------------------
    //      0 - 3        text length (bytes)
    //      4 - 7        worksheet cell count
    //      8 - 15       text style information
    //     16 - n        text data bytes
    //   n + 1 - m       worksheet entries
    //
    TRY
    {
        //
        // get text and worksheet data sizes
        //
        FailOSErr (SetFPos( refNum, fsFromStart, 0L));
        ReadSome((Ptr)&textLength, sizeof(long));
        ReadSome((Ptr)&WSEntryCt, sizeof(long));
        //
        // read in the EditText data
        //
        if(textLength > 0)
        {
            //
            // read the font info
            //
            ReadSome((Ptr)&theFontInfo, sizeof (fontInfo));
            ((CEnsembleDoc *) itsDocument)->theTextData
                ->SetFontData (theFontInfo);
            //
            // now, read the text
            //
            itsEditTextData = NewHandleCanFail(textLength);
            FailNIL(itsEditTextData);
            ReadSome(*itsEditTextData, textLength);
        }
        //
        // read in the worksheet data
        if(WSEntryCt > 0)
        {
            ReadWSEntries(WSEntryCt);
        }
    }
    CATCH
    {
        ForgetHandle (itsEditTextData);
    }
    ENDTRY;
}
```

The comments at the beginning of this code describe the new file format. The file begins with two long integers. The first contains the length of the text portion of the file, and the second contains the number of worksheet entries in the file. Following the long integers are the text data, if present. If so, the data are preceded by the style information that was applied to the text before it was last saved. The style information takes up 8 bytes and is immediately followed by the text itself. Following the text (or immediately after the worksheet cell count if the text isn't present) are the individual worksheet entries (if any).

The code for the **ReadData** method is placed inside a TRY block, so that if an error occurs during reading of the file, the data can be properly disposed. The error will also be propagated to the error handler defined by the **IApplication** method, which will show an alert, informing the user of the nature of the error.

The sequence of steps taken by the **ReadData** method is as follows:

1. The first task is to reset the file position to its beginning and read in the two long integer values. The contents of these values will determine which additional functions of the method will be performed.

2. If the text length is nonzero, the text style information will be read. The **CEnsembleDoc** class's **SetFontData** method is called to store the font style information, so that it can be applied when the EditText window is opened. (The file is usually read before the window is open. The only exception is when a **Revert** command is executed.)

3. After the text style information has been read, the text that follows (whose length is specified in the first long integer) is read into a handle allocated to hold the data. The handle is stored in the **itsEditTextData** instance variable.

4. If the number of worksheet cells is nonzero, a separate method is called to read the cell entries. This method is described next.

After the entire contents of the file have been read without error, the text and/or worksheet data will have been filed away for reference by other classes and methods.

ReadWSEntries Method Code

The method that reads worksheet entries is called by the **ReadData** method if worksheet data are present in the input file.

The **ReadWSEntries** method is passed only one parameter, indicating the number of entries to be read. Prior to displaying the method itself, we will discuss the format of a worksheet entry by showing its structure and expected contents.

The worksheet entry consists of a header record that is defined by a structure called **WSCellEntry**. The header is immediately followed by the ASCII text of the corresponding cell's contents. The contents of the **WSCellEntry** structure are as follows:

```
typedef struct
{
   Cell     WSCell;
   short    WSType;
   short    WSSize;

} WSCellEntry;
```

In the structure, the **Cell** type is the same as a **Point** and is used with all of the TCL's **CTable** methods, instead of the Macintosh **Point** data type. The **WSCell** identifies the column and row of the cell to which the rest of the entry applies. The column is stored in the **WSCell.h** component, and the row is stored in **WSCell.v**. The **WSType** field identifies numeric versus string entries, and the **WSSize** field specifies the length of the entry string that follows. Rather than keep a lot of nonessential data for each worksheet cell, only the entry text that defines the contents of the entry is stored. The code to read these entries is as follows:

ReadWSEntries *method (beginning)*

```
void CEnsembleData::ReadWSEntries (long entryCount)
{
   WSCellEntry  anEntry;
   short  index;
```

ReadWSEntries
method (concluded)

```
	Str255	entryData;
	CWSEntry	*aWSEntry;

	for(index = 0; index < entryCount; index++)
	{
		ReadSome((Ptr)&anEntry, sizeof(WSCellEntry));
		ReadSome((Ptr)&entryData[1], (long) anEntry.WSSize);
		entryData[0] = anEntry.WSSize;
		TRY
		{
			//
			// create a worksheet cell entry, putting the
			// entry text that was read into the entry field
			// of the worksheet cell, then set the value field
			// to 0.0. If the entry type is a value, then the
			// value will be recalculated when the worksheet is
			// displayed. Enter the worksheet cell into the Cluster.
			//
			aWSEntry = new CWSEntry;
			aWSEntry->IWSEntry ();
			aWSEntry->SetWSCell (anEntry.WSCell);
			aWSEntry->SetWSType (anEntry.WSType);
			aWSEntry->SetWSValue (0.0);
			aWSEntry->SetWSEntry (entryData);
			if(anEntry.WSType == 1)
			{
				aWSEntry->SetWSText(entryData);	// string
			}
			else
			{
				aWSEntry->SetWSText("\p0.00");		// value
			}
			itsCluster->Add(aWSEntry);
		}
		CATCH
		{
			ForgetObject (aWSEntry);
		}
		ENDTRY;
	}
}
```

After the **WSCellEntry** header structure has been read, the string defining the contents of the cell is read. (Its size is specified by the **WSSize** field.)

When the header and entry string have been read, the method creates a new instance of the **CWSEntry** class and initializes its instance variables by calling the access methods to set its cell, type, value, entry string, and text representation string. If the cell is intended to hold a string, the text representation is a copy of the entry string. If the cell holds a value, the text representation is set to **0.0**. The **CWSEntry** class will be discussed in more detail later.

After the **CWSEntry** has been built, it is added to the cluster that was allocated by the **IEnsembleData** method for storage of the worksheet cell data.

The process of reading a new header and its entry string, creating a new **CWSEntry** instance, initializing the instance variables, and adding the **CWSEntry** instance to the cluster is repeated until the entry count is exhausted. If an error occurs during this process, the CATCH block of the code will be executed, disposing of the entry that may have been allocated. The error is propagated to the error handler created by the **IApplication** method, where an error alert is posted, notifying the user of the problem.

WriteData Method Code

The **WriteData** method has been substantially rewritten to write the EditText and worksheet data in the new file format (defined in the **ReadData** method on page 187). The code for the **WriteData** method is as follows:

***WriteData** method code (beginning)*

```
Boolean CEnsembleData::WriteData(void)
{
   CMainWindow   *theTextWindow;
   long    textLength, WSEntryCt, fileLength;
   fontInfo   theFontInfo;

   //
   // modified WriteData to get the TextEdit pane's Text Handle
   // and then write out the contents of that handle.
   //
   // additional modifications to handle writing out worksheet
   // cell entries into a composite file.
   //
   theTextWindow = ((CEnsembleDoc *)itsDocument)
      ->GetTextWindow();
   itsEditTextData = theTextWindow->GetEditTextHandle();
```

***WriteData** method code (concluded)*

```
    textLength = GetHandleSize(itsEditTextData);
    WSEntryCt = itsCluster->GetNumItems();

    //
    // write out the textLength & WSEntryCt values
    //
    FailOSErr (SetFPos( refNum, fsFromStart, 0L));
    WriteSome ((Ptr)&textLength, (long) sizeof(long));
    WriteSome ((Ptr)&WSEntryCt,  (long) sizeof(long));

    //
    // now, write out the text data, if any
    //
    if(textLength > 0)
    {
        //
        // first, write out the fontInfo structure's contents
        //
        theFontInfo = ((CEnsembleDoc *) itsDocument)
            ->theTextData->GetFontData();
        WriteSome ((Ptr)&theFontInfo, sizeof (fontInfo));

        //
        // now, write the text
        //
        WriteSome (*itsEditTextData, textLength);
    }

    //
    // finally, write out the worksheet
    // cell entries, if any
    //
    if(WSEntryCt > 0)
    {
        WriteWSEntries (WSEntryCt);
    }
    fileLength = GetLength();
    FailOSErr(SetEOF( refNum, fileLength));
    FailOSErr( FlushVol( NULL, volNum));
    return (TRUE);
}
```

The **WriteData** method follows essentially the same sequence of operations as the **ReadData** method, except that it writes data to the file, rather than reading from the file. The steps are as follows:

1. The **WriteData** method sends a message to the **CEnsembleDoc's GetTextWindow** method, to get a handle to the **CMainWindow** instance. It uses this handle to call the **MainWindow's GetEditTextHandle** method, to access the handle to the current EditText data in that window (if any). **WriteData** then calls the toolbox routine to return the handle size and stores this value in a long integer variable called **textLength**.

2. A **GetNumItems** message is sent to the cluster (**itsCluster**) to determine the number of entries in the worksheet cluster. The number is stored in a long integer variable called **WSEntryCt**.

3. The file is positioned at its start, and the contents of the two long integer variables' (**textLength** and **WSEntryCt**) are written to the file.

4. The code tests whether the **textLength** variable holds a value greater than 0, and if so, it accesses the **CEnsembleDoc** instance's **theTextData** variable and sends it the **GetFontData** message, storing the result in a local variable called **theFontInfo**. This is the following 8-byte **fontInfo** structure:

```
typedef struct
{
   short fontNumber;
   short fontSize;
   short fontStyle;
   short fontAlign;

} fontInfo;
```

5. After acquiring the **fontInfo** structure, the **WriteData** method writes it out to the file.

6. The text itself is written to the file, immediately following the **fontInfo** structure. The length of the text is contained in the **textLength** variable.

7. The **WriteData** method then checks whether any worksheet entries are present by testing the **WSEntryCt** value.

If there are entries, it calls the **WriteWSEntries** method (to be described shortly) to write these entries to the file.

8. Before the **WriteData** method finishes, it gets the length of the file, calls the **SetEOF** method to set the end-of-file marker at that point, and then calls **flushVol** to write the contents of the buffer out to the file.

WriteWSEntries Method Code

The **WriteWSEntries** method writes all of the worksheet entries in the cluster to the file, in the proper format. This method is fairly simple, compared with the **ReadWSEntries** code previously described. The code for **WriteWSEntries** is as follows:

```
void CEnsembleData::WriteWSEntries (long entryCount)
{
   WSCellEntry   anEntry;
   short    index;
   long     WSEntryCt;
   Str255     entryData;
   CWSEntry    *aWSEntry;
   for(index = 1; index <= entryCount; index++)
   {
      itsCluster->GetItem (&aWSEntry, index);
      FailNIL (aWSEntry);
      anEntry.WSCell = aWSEntry->GetWSCell();
      anEntry.WSType = aWSEntry->GetWSType();
      aWSEntry->GetWSEntry(entryData);
      anEntry.WSSize = entryData[0];
      WriteSome ((Ptr)&anEntry, sizeof(WSCellEntry));
      WriteSome ((Ptr)&entryData[1], (long) entryData[0]);
   }
}
```

All of the needed information is contained in the cluster entries. The method is passed a single parameter specifying the number of entries, and it proceeds to get each item, in turn, accessing the cell, type, entry string, and entry size information from the entry by calling the appropriate access methods. Once acquired, these data are written out to the file. Each entry consists of a **WSCellEntry** structure (described on page 190), followed by the entry string itself. As previously indicated, the structure and the entry string are all that is required to reconstitute the contents of the worksheet cell.

DisposeData Method Code

The **DisposeData** method has been modified to handle the deletion of the worksheet data from memory. The method is called when, for example, a **Revert** or **Close** operation is performed. The code is as follows:

```
void CEnsembleData::DisposeData(void)
{
   long WSEntryCt;
   long  index;

   if (itsEditTextData != NULL)
   {
      DisposHandle (itsEditTextData);
      itsEditTextData = NULL;
   }
   if (itsCluster != NULL)
   {
      WSEntryCt = itsCluster->GetNumItems();
      for (index = 1; index <= WSEntryCt; index++)
      {
         itsCluster->DeleteItem (1);
      }
   }
}
```

The method tests whether the **itsEditTextData** handle is NULL, and if not, it disposes of the handle. It also tests whether the **itsCluster** instance is NULL, and if not, it sends the cluster a message to delete item 1 continually, until the number of items has been depleted.

Adding a New Access Method

Other classes in the application need to access the data stored in the worksheet cluster, so an access method to return the handle to the cluster's instance has been added.

GetCluster Method Code

The code for accessing the worksheet's cluster is provided as a public access method of the **CEnsembleData** class. The code is as follows:

```
CCluster *CEnsembleData::GetCluster (void)
{
   return itsCluster;
}
```

All that this method does is return the value of the **itsCluster** instance variable.

Summary: Customizing CEnsembleData

The modifications to various methods in the **CEnsembleData** module described in the preceding sections were required because a new file format was adopted and we needed to enhance the **ReadData** and **WriteData** methods greatly, compared with their versions described in Chapter 5. It's important to note, however, that although we made a significant change to the file format and its contents, the modifications to effect this change are localized in the **CEnsembleData** class. None of the methods in the **CEnsembleDoc** or **CMainWindow** classes were affected.

The principle of keeping changes localized is a side effect (or natural consequence) of object-oriented design. Changes that affect one area of the application (one object) need only be made to that area.

In order to ensure the insulation (encapsulation) of the elements in one object from others, special methods to permit other objects to access the private data are provided. With these methods, we can modify the internal behavior of a given class without making a single change to other classes in the application. This principle is upheld throughout the design of the Ensemble application.

Customizing the CCalcWindow Code

The code to implement the full functionality of a capable worksheet is contained in the **CalcWindow.c** module. This module contains quite a few classes. To describe the customizing procedures, we will break this section into a number of appropriate subsections, each of which will discuss an aspect of implementing the worksheet.

Customizing the Lists

The worksheet contains three lists that implement the column labels, row labels, and main worksheet cells, respectively. The column and row label lists work in concert with the main worksheet list to provide synchronized scrolling and autodrag selection. Although the model we've implemented allows only a single cell at a time to be selected, this behavior could be modified to allow selecting rectangular contiguous cells without much difficulty.

Table 8-2 defines the classes and methods that implement the list handling chores for the worksheet. Recall that when

Table 8-2
Custom code modifications to list classes

Class	Method	Description
CList5	IViewTemp	Initializes column label list
CList5	GetCellText	Returns specified column label
CList5	DrawCell	Draws a column label cell
CList10	IViewTemp	Initializes row label list
CList10	GetCellText	Returns specified row label
CList10	DrawCell	Draws a row label cell
CList15	IViewTemp	Initializes main worksheet table
CList15	GetCellText	Returns worksheet cell text entry
CList15	GetContents	Extracts entry string, text string, value, and type from a cell
CList15	Scroll	Scrolls the worksheet
CList15	SetLists SetCluster SetArray	Provides access methods to store instances needed by the methods in the class.
CList15	ProviderChanged	Handles selections in the main worksheet table

the **Bord** resources were modified, in Chapter 6, we disabled mouse clicks (set the **Wants Clicks** parameter to FALSE) for both the **Bord 134** and **Bord 136** borders (see Figure 6-39 on page 157 and Figure 6-40 on page 158). We did this purposely, because we didn't want the column and row label list cells to become highlighted when the mouse was clicked inside the border. Setting the **Wants Clicks** field to FALSE for a border disables clicks for anything inside that border.

Each of the lists has the **IViewTemp** and **GetCellText** methods. The column and row label lists also have a new **DrawCell** method, which overrides that method in the TCL's **CTable** class. The main worksheet has several additional methods, including an override of the **CTable Scroll** method, several new access methods, and a **ProviderChanged** method to intercept the **BroadcastChange** messages sent by the **CTable** class in response to selection changes in the worksheet.

The subsections that follow describe the individual methods for each of the lists and display the modified code that implements the intended behavior for the method. Each of the lists is set up according to some definitions that we have added to the code, to make it fairly easy to change for different numbers of rows or columns. The following **#define** statements establish the current settings for these parameters:

Definitions of worksheet parameters

```
//
// added definitions
//
#define tblCellWidth        48      // worksheet cell width
#define tblCellHeight       14      // worksheet cell height
#define horLabWidth         48      // column label width
#define horLabHeight        20      // column label height
#define vertLabWidth        32      // row label width
#define vertLabHeight       14      // row label height
#define vertLabMargin        5      // row label margin

#define numRows             50      // number of rows
#define numCols             26      // number of columns
```

The comments are self-explanatory.

CList5 IViewTemp Method Code

The **CList5** class implements the column label list, which will be scrolled in sync with the horizontal scroll bar of the main worksheet. The code is as follows:

```
void CList5::IViewTemp(CView     *anEnclosure,
                       CBureaucrat *aSupervisor,
                       Ptr    viewData)
{
   inherited::IViewTemp (anEnclosure, aSupervisor, viewData);

   // any additional initialization for your subclass
   DeleteCol(1, 0);
   SetDefaults(horLabWidth, horLabHeight);
   SetColBorders(1, patCopy, black);
   AddRow(1, 0);
   AddCol(numCols,0);
}
```

When AppMaker generates the resources for the list elements, it makes the assumption that you will be creating a single-column list with multiple rows and that you will be using the default settings for the row or column height and width.

The **IViewTemp** code for **List5** (column labels) deletes the first column (columns and rows in lists are numbered beginning with 0), and then applies the new default settings for the column labels (**horLabWidth** and **horLabHeight**). It also sets column borders to 1-point black lines, with a transfer mode of **patCopy**, which will overwrite anything else in that position.

Finally, the **IViewTemp** method addsone row, beginning with row 0, and then adds the number of columns specified by the **numCols** definition. This sets up a horizontal table that consists of 1 row and 26 columns (using the specified definitions).

The columns are 20 pixels tall and 48 pixels wide. In this version of the Ensemble application, the columns and rows have fixed sizes. In the next chapter, we will be adding the user interface features to permit the worksheet format to be modified.

CList5 GetCellText Method Code

The **GetCellText** method generated by AppMaker for the **CList5** class has been rewritten as follows:

```
void CList5::GetCellText (Cell      aCell,
                          short     availableWidth,
                          StringPtr itsText)
{
   short col;

   col = aCell.h;
   CopyPString("\pA", itsText);
   itsText[1] += col;
}
```

This code makes provision for a maximum of 26 columns. It changes column numbers in the range 0–25 to A–Z and stores the string representation in the **itsText** variable. The method could easily be modified to handle a larger number of columns.

CList5 DrawCell Method Code

This method overrides and takes the place of the corresponding method in the **CTable** class. The code is as follows:

```
void CList5::DrawCell (Cell theCell, Rect *cellRect)
{
   Str255 cellText;
   short availWidth, textWidth;

   availWidth = cellRect->right - cellRect->left;
   GetCellText(theCell, availWidth, cellText);
   textWidth = StringWidth(cellText);
   indent.h = (availWidth - textWidth) >> 1;

   if (cellText[0] > 0)
   {
      MoveTo( cellRect->left + indent.h, cellRect->top + indent.v);
      DrawString( cellText);
   }
}
```

The code calculates the available width of the column (**availWidth**), using its full width. It calls the **GetCellText** method, calculates the number of pixels occupied by the column label, and then calculates a horizontal **indent.h** value that will center the label in the column. If the label width is greater than

0, the column label string is drawn at the appropriate position within the cell.

CList10 IViewTemp Method Code

The **CList10** class implements the row label table for the worksheet. The **IViewTemp** code for this class initializes the row label list to accommodate a single column and multiple rows, specified by the definitions listed earlier. The code for the **IViewTemp** method is as follows:

```
void CList10::IViewTemp (CView      *anEnclosure,
                         CBureaucrat *aSupervisor,
                         Ptr     viewData)
{
   inherited::IViewTemp (anEnclosure, aSupervisor, viewData);

   // any additional initialization for your subclass
   DeleteCol(1, 0);
   SetDefaults(vertLabWidth, vertLabHeight);
   SetRowBorders(1, patCopy, black);
   AddCol (1, 0);
   AddRow (numRows, 0);
}
```

As was indicated for the column label list, AppMaker's implied single-column, multiple-row table, with default settings, is modified by deleting the first (only) column and then setting the default values for the label width and height (**vertLabWidth** and **vertLabHeight**) to 32 and 14 pixels, respectively. The row borders are set to 1-point black lines, using the **patCopy** mode to overwrite anything in the border's position. The single column is added, followed by the number of rows specified by the **numRows** definition, whose value here is 50.

CList10 GetCellText Method Code

The **GetCellText** method for the **List10** table has been rewritten to convert the row number to a string value between 1 and 50. The code is as follows:

GetCellText *method code (beginning)*

```
void CList10::GetCellText  (Cell    aCell,
                            short   availableWidth,
                            StringPtr itsText)
{
```

***GetCellText* method code (concluded)**

```
    short row;

    row = aCell.v+1;
    NumToString(row, itsText);
}
```

In this code, the row is increased by 1, so that we won't have a row 0, and then the toolbox **NumToString** utility is used to convert the number to a string in the **itsText** parameter.

CList10 DrawCell Method Code

This method overrides and replaces the corresponding method in the **CTable** class. The code is as follows:

```
void CList10::DrawCell (Cell theCell, Rect *cellRect)
{
    Str255 cellText;
    short availWidth
    short textWidth;

    availWidth = cellRect->right - cellRect->left;
    GetCellText(theCell, availWidth, cellText);
    textWidth = StringWidth(cellText);
    indent.h = availWidth - textWidth - vertLabMargin;
    if (cellText[0] > 0)
    {
        MoveTo( cellRect->left + indent.h, cellRect->top + indent.v);
        DrawString( cellText);
    }
}
```

For the row labels, we want to right-justify the row number, so this method calculates the available width, calls the **GetCellText** method and calculates its text width, and then indents the text so that it is right-justified in the row, with the exception of a small (5-pixel) right margin (**vertLabMargin**). If the label string has a length greater than 0, the row label string is drawn at the calculated position.

CList15 IViewTemp Method Code

The **CList15** class implements the body of the worksheet, which has 26 columns and 50 rows, by using the definitions described earlier. The code for the **IViewTemp** method initializes the table so that it has the proper number of cells, with widths and heights that correspond to the settings for the column and row label tables. The code for this method is as follows:

```
void CList15::IViewTemp  (CView     *anEnclosure,
                          CBureaucrat *aSupervisor,
                          Ptr     viewData)
{
   inherited::IViewTemp (anEnclosure, aSupervisor, viewData);

   // any additional initialization for your subclass
   DeleteCol(1, 0);
   SetDefaults(tblCellWidth, tblCellHeight);
   SetColBorders(1, patCopy, ltGray);
   SetRowBorders(1, patCopy, ltGray);
   AddRow(numRows, 0);
   AddCol(numCols, 0);
}
```

The **IViewTemp** method for the **CList15** class follows essentially the same pattern as the corresponding methods in the **CList5** and **CList10** classes. The initial single column is deleted, and the default settings are changed to the column label width and the row label height, so that the cells will match the dimensions of the corresponding column and row label tables.

In this case, we are setting 1-point column and row borders, in light gray (**ltGray**) rather than black, with the **patCopy** transfer mode. Finally, the number of rows and columns specified by the **numRows** and **numCols** variables is allocated for the table. Note that although the full number of rows and columns is allocated, no extra storage is set aside for the contents of these cells. In essence, they are assumed empty until they are explicitly filled with values.

CList15 GetCellText Method Code

The code for the **GetCellText** method of the **CList15** class is as follows:

GetCellText *method code (beginning)*

```
void CList15::GetCellText (Cell    aCell, short availableWidth,
                           StringPtr   itsText)
{
   double   itsValue, newValue;
```

***GetCellText* method code (concluded)**

```
    short   itsType, index;
    long   aParam;
    Str255   itsEntry, itsCellText;
    decform   aFormat;
    extended temp;
    CWSEntry  *anObj;

    if((CWSEntry *)itsCluster == NULL)
    {
        CopyPString("\p", itsText);
        return;
    }
    aParam = *(long*) &aCell;
    anObj = (CWSEntry *)itsCluster->FindItem1 (FindWSCell, aParam);
    if(anObj)
    {
        if((itsType = anObj->GetWSType()) == 2)
        {
            index = 1;
            anObj->GetWSEntry(itsEntry);
            newValue = ((CCalcWindow *)itsSupervisor)->GetExpression
                (itsEntry, &index, 0);
            itsValue = anObj->GetWSValue();
            if (newValue != itsValue)
            {
                aFormat.style  = FIXEDDECIMAL;
                aFormat.digits = 2;
                x96tox80(&newValue, &temp);
                num2str(&aFormat, temp, itsCellText);
                anObj->SetWSText(itsCellText);
                anObj->SetWSValue(newValue);
            }
        }
        anObj->GetWSText(itsCellText);
        CopyPString(itsCellText, itsText);
    }
    else
        CopyPString("\p", itsText);
}
```

Following is an explanation of the operation of the **GetCell-Text** code:

1. If the instance variable **itsCluster** is NULL, then an empty string is written into the **itsText** parameter and the method returns. This case can (and will) occur when the table is first initialized, because the **IViewTemp** method

for the table will execute before the **ICalcWindow** method has an opportunity to store the handle to the cluster.

2. The **aCell** parameter is cast into a long variable (**aParam**) so that the TCL's **FindItem1** method can be used (it requires a pointer to a single long variable) to search for a cell in the cluster whose cell number matches the one being sought. We have supplied a search function called **FindWSCell**, whose definition is as follows:

***FindWSCell** global function code*

```
Boolean  FindWSCell (CObject *anEntry, long param)
{
   Cell  eCell, pCell;

   pCell = *(Cell*) &param;
   eCell = ((CWSEntry *) anEntry)->GetWSCell();

   if((pCell.h == eCell.h) && (pCell.v == eCell.v))
   {
      return TRUE;
   }
   else
   {
      return FALSE;
   }
}
```

The **FindWSCell** function recasts the incoming **param** argument to a **Cell** type, placing it into the variable **pCell** (parameter cell). It then accesses the cell addressed by the **anEntry** argument, storing this cell into the variable **eCell** (entry cell). Next, it compares the column and row components of the two cells, and if they match exactly, the function returns a result of TRUE; otherwise, it returns a result of FALSE. The **FindItem1** method continues executing, calling the **FindWSCell** function for each entry in the cluster until a result of TRUE is returned, in which case it returns a handle to the matched entry. If no cells match, the **FindItem1** method returns a NULL handle.

3. The **GetCellText** method stores the object handle returned by the **FindItem1** method in a variable called **anObj**. If the handle is NULL, an empty Pascal string is

copied into the **itsText** parameter, and the **GetCellText** method returns.

4. If the specified cell exists in the cluster, the **GetCellText** method calls the access method to get the type of entry, storing this in the **itsType** variable. There are two types of entries: type 1, a string, and type 2, a formula (value). If the type is *not* equal to 2, then it is a string, and the method accesses the text representation of the cell's contents, copies it to the **itsText** parameter, and returns.

5. If the type of the entry *is* equal to 2, then the entry contains a formula (which may be a simple numeric value) that must be parsed to obtain its current value. The parsing operation is handled by a new method called **GetExpression**, which will be described later. This method takes the entry handle, a starting index that points to the first position of the formula string (the **GetExpression** method may be called recursively), and a nesting depth value, which is initially set to 0. When the method returns, the double-precision floating-point result is stored in a variable called **newValue**.

6. If the existing value differs from the **newValue** (determined by using the access method to obtain the existing value and then comparing the two), the cell's text must be updated. A format of FIXEDDECIMAL, with two digits of precision after the decimal, is established, the 96-bit floating-point value is converted to a compatible 80-bit Standard Apple® Numerics Environment (SANE) extended value, and then the SANE **num2str** function is called to convert the value to a string. The string is stored back into the entry by calling the **SetWSText** access method, and the value is stored into the entry by calling the **SetWSValue** access method. Finally, the code to copy the string (**itsCellText**) into the **itsText** parameter is executed and the method returns.

The last step makes use of the SANE functions to convert the value returned by the **GetExpression** method to a string. The first step is to convert the double-precision 96-bit floating-point value to an 80-bit SANE extended type and then perform the conversion to a string. The ANSI **sprintf** function could have been used instead; however, this would require that the large ANSI library be a permanent part of the project, instead of the relatively small SANE routines.

Another important point that was mentioned in passing, but bears some amplification, is the handling of non existent cells. When the **GetCellText** method is called by the **CTable** class's **DrawCell** method, it expects an existing cell, whose contents are to be drawn. By not storing dummy empty cells in the cluster for non existent cells, we have greatly reduced the memory requirements for the worksheet. In fact, if you define values for the top left and bottom right cells in the worksheet, only two entries will be stored in the cluster.

CList15 GetContents Method Code

The **GetContents** method is called by other methods to access the current values of a cell's instance variables. The method determines whether the cluster exists.If it does, the method uses the **CCluster FindItem1** method to attempt to locate the cell in the cluster. If the cluster doesn't exist or the cell can't be found, then the method stores empty strings for the entry and text representation and a value of 0.0.

If the cell exists, then its entry and text strings and the current value are returned. The code for the **GetContents** method is as follows:

GetContents *method code (beginning)*

```
void CList15::GetContents (Cell aCell, StringPtr entry, double *itsValue,
                        short *itsType, StringPtr cellText)
{
   long  aParam;
   CWSEntry *anEntry;
   aParam = *(long*) &aCell;
   if((CWSEntry *) itsCluster)
   {
      if((anEntry = (CWSEntry *)itsCluster->FindItem1 (FindWSCell,
          aParam)) == NULL)
      {
         CopyPString ("\p", entry);
         CopyPString ("\p", cellText);
         *itsValue = 0.0;
         *itsType = -1;
      }
      else
      {
         anEntry->GetWSEntry (entry);
```

GetContents *method code (concluded)*

```
            anEntry->GetWSText (cellText);
            *itsValue = anEntry->GetWSValue();
            *itsType = anEntry->GetWSType();
        }
    }
    else
    {
        CopyPString ("\p", entry);
        CopyPString ("\p", cellText);
        *itsValue = 0.0;
        *itsType = -1;
    }
}
```

If the cell exists, the **GetContents** method merely accesses the current settings for the cell's instance variables. No attempt is made to parse the entry string to recompute the value, even if the cell type is **2** (a formula).

CList15 SetLists Method Code

The **SetLists** method is called by the **ICalcWindow** initialization method to pass the handles of the column label and row label lists to the main worksheet list. The function **SetLists** is commonly called an *access method*, because it provides the means to access something that isn't normally accessible. In this case, access to the label lists is provided *to* the main worksheet list. The code for **SetLists** is as follows:

```
void CList15::SetLists  (CTable *hLabelList, CTable *vLabelList)
{
    itsHList = hLabelList;
    itsVList = vLabelList;
}
```

As is apparent, the code merely stores the incoming parameters into instance variables that we've added to the **CList15** class declaration.

CList15 SetCluster Method Code

The **SetCluster** code is also an access method used by the **ICalcWindow** code to pass a handle—for the cluster holding the cell entries—to the main worksheet list class. The code for the **SetCluster** method is as follows:

```
void CList15::SetCluster (CCluster *aCluster)
{
    itsCluster = aCluster;
}
```

Once again, the code merely stores the incoming cluster handle into an instance variable for the **CList15** class. This provides access to the cluster from within the class.

CList15 SetArray Method Code

The cluster to hold the cell entries for the main worksheet was allocated in the **CEnsembleData** class (see page 186), but it must be *installed* into the collaboration mechanism as the provider for the **CList15** class. This is done so that changes to the elements of the array will cause a **BroadcastChange** message to be sent to the CCollaborator, which calls the **ProviderChanged** method of the **CList15** class.

In the case of the **CAMArrayPane** that makes up the main worksheet's table, the collaboration connection must be established explicitly. Once the **CList15** instance is established as an explicit *dependent* of the cluster, any changes to the cluster will immediately be reflected by the receipt of a **ProviderChanged** message.

The code for the method to install the cluster as the worksheet's provider is as follows:

```
void CList15::SetArray ( CArray  *anArray, Boolean  fOwnership)
{
    itsArray = anArray;
    ownsArray = fOwnership;
    DependUpon( itsArray);
}
```

This method is provided to override the **SetArray** method in the **CArrayPane** class. That method requires that the array contain entries for the scope of the companion table. Because we are handling nonexistent entries and want the entire table to be allocated, we must override the superclass method. The instance variable called **ownsArray** establishes ownership of the array by the table. We don't want this connection, so a value of FALSE is passed to **SetArray** when it is called. The **DependUpon** method establishes the fact that the **CList15** class *depends upon* **itsArray**.

CList15 ProviderChanged Method Code

The **ProviderChanged** method is called in response to changes to the cluster holding the cell entries for the main worksheet. It overrides the functionality of the default method for two of the potential **BroadcastChange** messages sent by the array.

Specifically, we don't want to observe the default behavior when elements are inserted or deleted from the array. In those cases, the standard behavior is to expand or shrink the table. Because we would like the appearance of the table to remain constant, these messages are ignored by our override method. The code for the **ProviderChanged** method is as follows:

```
void CList15::ProviderChanged (CCollaborator *aProvider,
                               long reason, void *info)
{
  if (aProvider == itsArray)
  {
    switch( reason)
    {
      case arrayInsertElement:
        //
        // handle this case as an NOP (no operation)
        //
        break;

      case arrayDeleteElement:
        //
        // handle this case as an NOP (no operation)
        //
        break;

      default:
        inherited::ProviderChanged( aProvider, reason, info);
        break;
    }
  }
  else
    inherited::ProviderChanged( aProvider, reason, info);
}
```

Note that we specifically handle the **arrayInsertElement** and **arrayDeleteElement** messages as do-nothing cases. For messages other than these, the inherited method is called.

CList15 Scroll Method Code

The scroll bars for the worksheet are associated with **CScroll-Pane**, which only covers the area of the main worksheet. However, when the scroll bars are used, we want the column and/or row label lists to scroll in synchrony with the main worksheet. In order to synchronize the scrolling operation, we have created an override for the **CTable** class's **Scroll** method. The following code shows a very simple method for synchronizing the operation of multiple lists with a single set of scroll bars:

```
void CList15::Scroll (long hDelta, long vDelta, Boolean redraw)
{
   inherited::Scroll (hDelta, vDelta, redraw);
   if(hDelta)
   {
      itsHList->Scroll(hDelta, 0, TRUE);
   }
   if(vDelta)
   {
      itsVList->Scroll(0, vDelta, TRUE);
   }
}
```

The **Scroll** method requires us to include the **SetLists** access method (page 209) in the **CList15** class. The inherited method is called to scroll the main worksheet, and, depending on whether the **hDelta** or **vDelta** value is nonzero, the corresponding column or row (**itsHList** or **itsVList**) handle is used to call the **Scroll** method for that class.

Customizing the CCalcWindow Code

The bulk of the code that supports the functionality of the worksheet is contained in the **CCalcWindow** class. This class contains the methods that initialize the worksheet elements, handle the selection of cells, and process the strings and formula entries that constitute a cell entry's contents.

Defining a Cell's Contents

In order to provide reasonable functionality, it was decided that cells could contain any of the following elements, in an appropriate order:

- If the cell entry text begins with a single quote, the entry is assumed to be a string and the characters that follow the single quote are stored, as is, into the cell.

- If the cell entry begins with anything other than a single quote, then it is assumed to be a formula. Formulas can contain the following elements:

  - Balanced left and right parenthesis ( ), for grouping terms.

  - Simple numeric constants, specified either with or without an embedded decimal point. All constants are converted to floating-point values and no scientific notation is allowed. Values such as 10 or 43.95 are examples of the acceptable notation.

  - References to other cells (e.g., B13, Z5, or A1).

  - Standard arithmetic operators for addition, subtraction, multiplication, and division, entered as **+**, **–**, *****, and **/**, respectively.

Worksheet cell formulas are evaluated in a strict left-to-right sequence, without regard to any implied precedence of the operators. When the order of evaluation is important, parentheses can be used to enclose the terms to be evaluated before a succeeding operator is applied. Examples of legal cell entries are shown in Table 8-3.

Table 8-3
Example worksheet entries

Cell	Cell Entry	Displayed Contents
A1	'This is a very long string	This is a very long string
B10	10	10.00
C4	B10 + 15	25.00
D3	(C4 + 5) / 3 * B10	100.00

Table 8-4
CCalcWindow customized and new methods

Class	Method	Description
CCalcWindow	ICalcWindow	Initializes the window and its interface elements
CCalcWindow	UpdateMenus	Disables **Close** command in **File** menu
CCalcWindow	ProviderChanged	Handles selection of a cell
CCalcWindow	DoEnterButton	Makes a cell entry
CCalcWindow	DoCancelButton	Reverts to original cell contents
CCalcWindow	**ParseEntry**	Parses the **Entry** field and creates a cell object of the correct type
CCalcWindow	**GetExpression**	Evaluates a formula entry
CCalcWindow	**GetToken**	Returns the next token while evaluating an entry
CCalcWindow	**isConst**	Determines whether a token is a constant
CCalcWindow	**isCell**	Determines whether a token is a reference to another cell
CCalcWindow	**MakeStringObj**	Creates a string-type cell entry
CCalcWindow	**MakeValueObj**	Creates a value-type cell entry
CCalcWindow	**Activate**	Refreshes the table on an activate event

Strings that exceed the width of a cell will overlap the adjacent cells to their right. This allows you to create headings that span a number of cells. The entry is anchored in the beginning cell. You can justify string entries by inserting an appropriate number of spaces in between the single-quote mark and the first character of the string. Looking at the formulas in the table should give you an idea of how to construct even more complex forms. Parentheses can be nested to any depth, as desired.

The Customized Methods

In order to implement the functions of a worksheet in the **CCalcWindow** class, several new methods have been added to those generated by AppMaker. The full list of methods in the **CCalcWindow** class is shown in Table 8-4.

The methods shown in boldface type in the table are newly created. The names in plain type were generated by AppMaker, but have been customized for our purposes.

ICalcWindow Method Code

The **ICalcWindow** method is called by the **BuildWindows** method in the **zEnsembleDoc** module when the window is created. The code for the **ICalcWindow** method is as follows:

```
void CCalcWindow::ICalcWindow (CDirector *aSupervisor,
                               CEnsembleData *theData)
{
   Str255 theFilename;

   itsData = theData;
   inherited::IZCalcWindow (aSupervisor);
   gDecorator->StaggerWindow (itsWindow);
   if(((CEnsembleDoc *) aSupervisor)->itsFile != NULL)
   {
      ((CEnsembleDoc *) aSupervisor)->itsFile
         ->GetName(theFilename);
      itsWindow->SetTitle(theFilename);
   }
   EntryField->SetTextString("\p");
   TEAutoView (TRUE, EntryField->macTE);
   ((CList15 *)List15)->SetLists (List5, List10);
   wsCluster = theData->GetCluster();
   ((CList15 *)List15)->SetCluster (wsCluster);
   ((CList15 *)List15)->SetArray(wsCluster, FALSE);
   ((CList15 *)List15)->Refresh();
}
```

In the preceding code, the first three executable statements were generated by AppMaker. We have added the remaining code. The first statement saves the handle to the **CEnsembleData** instance in an instance variable called **itsData**. Next, the inherited **IZCalcWindow** method is called to create and initialize all the interface elements in the window. It is at this time that all the **IViewTemp** methods for the borders, lists, buttons, scroll pane, and user panes are called. After the **IZCalcWindow** method returns, the window and all its elements have been created and initialized. The **gDecorator** is sent a message to stagger the window, with respect to the other windows on the screen.

Following the AppMaker-generated code, there are a few things that need to be done before the worksheet is ready for use:

1. If a file is associated with the document, the code accesses its title and also writes it into the title bar of the **CalcWindow**.

2. The contents of the **Entry** field are set to an empty string, and the toolbox **TEAutoView** function is called with a handle to the TextEdit record for the **Entry** field, so that the field will scroll when a long entry is typed into the field.

3. The series of access methods is called. The **SetLists**, **GetCluster**, **SetCluster**, and **SetArray** methods were described previously. The **Get** method accesses an existing handle, and the corresponding **Set** method passes the handle to another class, when it can be stored in an instance variable, for easy access.

4. The **Refresh** method forces the worksheet to be redrawn. The **GetCellText** method for the main worksheet (**CList15**) is called, whereupon it reevaluates the contents of each cell and redraws it on the screen.

UpdateMenus Method Code

The **UpdateMenus** method has been modified as follows:

```
void  CCalcWindow::UpdateMenus(void)
{
   inherited::UpdateMenus ();

   //
   // disable Close if CalcWindow is
   // the frontmost window
   //
   gBartender->DisableCmd (cmdClose);
}
```

A single statement has been added to the code generated by AppMaker. When the **CalcWindow** is frontmost on the screen, we want to disable the **Close** command in the **File** menu, so that the worksheet alone cannot be closed.

ProviderChanged Method Code

The **ProviderChanged** method for the **CCalcWindow** class is called when a mouse click occurs inside the main worksheet. The **CTable** class sends a **BroadcastChange** message that is intercepted by the **CBureaucrat** class and sent to the table's supervisor as a **ProviderChanged** message, which in this case is the **CCalcWindow** class. Our worksheet design interprets a mouse click in a cell as a selection of that cell and as a prelude to changing or making a new entry into the cell. The code in the modified **ProviderChanged** method eliminates quite a bit of the superfluous code generated by AppMaker pertaining to the column and row lists (**CList5** and **Clist10**), as these lists are not operated on directly by the user. The modified code for the **ProviderChanged** method is as follows:

ProviderChanged *method code (beginning)*

```
void CCalcWindow::ProviderChanged (CCollaborator *aProvider,
                                   long reason,
                                   void* info)
{
   Str32 itsCellTitle;
   Str255 entry, cellText;
   long length;
   Cell aCell;
   short row, col, type;
   double value;

   if (aProvider == List15)
   {
      if (List15->HasSelection ())
      {
         SetPt (&aCell, 0, 0);
         List15->GetSelect (TRUE, &aCell);
         row = aCell.v;
         col = aCell.h;
         CopyPString("\pA", itsCellTitle);
         itsCellTitle[1] += col;
         NumToString (row+1, entry);
         ConcatPStrings (itsCellTitle, entry);
         ConcatPStrings (itsCellTitle, "\p:");
         CellNumLabel->SetTextString (itsCellTitle);
         ((CList15 *)List15)->GetContents (aCell, entry, &value,
            &type, cellText);
         if(type == 1)
         {
            CopyPString("\p'", entry);
```

ProviderChanged *method code (concluded)*

```
            ConcatPStrings(entry, cellText);
         }
         EntryField->SetTextString (entry);
         EntryField->BecomeGopher(TRUE);
         EntryField->SelectAll(TRUE);
      }
   }
}
```

This code deals only with messages that relate to the **CList15** instance. All others are ignored. If a cell is selected in the list, then the method proceeds; otherwise, it ignores the message. Following are the steps taken to handle a selection:

1. The selected cell is accessed via the **GetSelect** method, which returns the first (and only) selected cell. The components of the cell are saved as **row** and **col** variables.

2. The next series of statements formats the **col** and **row** values to take on the appearance of a cell number (e.g., **B13**, corresponding to **col=1** and **row=12**). The cell number is written to the static text field (**CellNumLabel**).

3. The next series of statements checks whether the selected cell holds a string (**type=1**), and if so, the **Entry** field is written with a single quote appended to the front of the entry text; otherwise, for a formula, the entry text is copied to the **Entry** field using the **SetTextString** method.

4. Sending the **BecomeGopher** message to the **EntryField** allows the field to accept all subsequent events (such as keystrokes and mouse clicks). The **SelectAll** message causes the entire contents of the **EntryField** to become highlighted. Pressing the **delete** key will delete all of the text in the entry. Entering any other text when the entry is highlighted will replace the contents of the **EntryField**.

Once the contents of the **EntryField** are changed, they can be stored by clicking on the **Enter** button. If you change your mind about making changes to the entry, you can click the **Cancel** button to restore the original contents of the cell to the **EntryField**.

DoEnterButton Method Code

The **DoEnterButton** method is called by the **DoCommand** method generated by AppMaker. The latter method does not require any changes. The **DoEnterButton** method was generated as an empty method by AppMaker, and we have added the necessary code to make it fully functional:

```
void CCalcWindow::DoEnterButton (void)
{
    long      length, param;
    Cell      aCell;
    Str255    theText;
    Handle    theTextHandle;
    CWSEntry  *anEntry, *anObj;

    length = EntryField->GetLength();
    if(length > 0)
    {
        SetPt(&aCell, 0, 0);
        if(List15->GetSelect(TRUE, &aCell))
        {
            theTextHandle = EntryField->GetTextHandle();
            BlockMove((*theTextHandle), &theText[1], length);
            theText[0] = length;
            if(anObj = ParseEntry (aCell, theText))
            {
                param = *(long *)&aCell;
                if((anEntry = (CWSEntry *)wsCluster->FindItem1
                    (FindWSCell, param)) != NULL)
                {
                    wsCluster->Remove(anEntry);
                }
                wsCluster->Add(anObj);
                List15->Refresh();
                ((CEnsembleData *) itsData)->SetDirty (TRUE);
            }
            else
                SysBeep(30);
        }
    }
}
```

The main function of the **DoEnterButton** code is to validate the entry and then store it in its corresponding cell as a valid string or formula entry. The code behaves as follows:

1. The **DoEnterButton** method first checks the length of the **EntryField** text string. If it is 0, then the method terminates. If the length of the field is greater than 0, the method continues executing.

2. The cell number of the current selection is obtained by calling the **GetSelect** method for **List15** (the main worksheet table). If the selection is not NULL, the method continues executing.

3. The handle to the entry text is accessed by calling the **GetTextHandle** method for the **EntryField**. The text string is moved to a local string variable using the toolbox **BlockMove** function, and its length byte is set.

4. The **DoEnterButton** method then calls **ParseEntry** with the cell number and the entry text. If **ParseEntry** returns a NULL object, then an error was found in the entry and the **SysBeep** toolbox function is called. If the object is not NULL, then the method searches the cluster for an existing object with the identical cell number (row and column) and deletes that entry if it is found.

5. Whether or not an entry existed in the cluster, the new object is added to the cluster, and the **dirty** flag is set for the file. This last action ensures that if the user attempts to quit the application or close the file without saving its contents, an alert will be displayed, offering the opportunity to save the file at that time.

DoCancelButton Method Code

The purpose of the **Cancel** button is to provide the means to restore the original cell contents into the **EntryField** if that field has been modified or changed and the user decides to make a different modification to the entry. Recall that an entry is displayed when the worksheet cell is clicked. The entry text is highlighted as it is stored in the **EntryField**. If the user strikes a single key at this point, the entry text will be erased and replaced with the character corresponding to that key. The **Cancel** button provides the means to restore the entry's original contents into the field. In truth, clicking on the same cell will accomplish the identical effect; however, the **Cancel** button is much closer to the user's focus at the time the entry is being made. The code for the **DoCancelButton** method is as follows:

```
void CCalcWindow::DoCancelButton (void)
{
   Cell aCell;
   Str255 entry, cellText;
   long length;
   short type;
   double value;

   if (List15->HasSelection ())
   {
      SetPt (&aCell, 0, 0);
      List15->GetSelect (TRUE, &aCell);
      ((CList15 *)List15)->GetContents (aCell, entry, &value,
          &type, cellText);
      if(type == 1)
      {
         CopyPString("\p'", entry);
         ConcatPStrings(entry, cellText);
      }
      EntryField->SetTextString (entry);
   }
}
```

Basically, the **DoCancelButton** method determines whether there is a current selected cell and returns to the calling method if not. If so, it gets the cell number of the selection and calls **GetContents** to get the current values of all the cell's instance variables. If the cell type is a string, a single-quote character is appended to the front of the entry string; otherwise, the entry string is written to the **EntryField** with the **SetTextString** method.

ParseEntry Method Code

The **ParseEntry** method is completely new. Its purpose is to determine the type of entry that has been keyed in, verify its validity, and return an object representing the newly created **CWSEntry** object. In the course of determining the entry's type, if it is a formula, the **ParseEntry** method also calls the **GetExpression** method to evaluate the formula. If an error is discovered (either the object is not a string or formula, or if it is a formula, it is improperly formed), a NULL object is returned. The code for **ParseEntry** is as follows:

***ParseEntry** method code (beginning)*

```
CWSEntry * CCalcWindow::ParseEntry (Cell aCell, StringPtr anEntry)
{
   Str255 aString;
   double value;
   short token, index = 1;
   CWSEntry *anObj;

   token = GetToken (anEntry, &index, &value, aString);
   if (token == 1)
   {
      //
      // token is a string
      //
      TRY
      {
         anObj = MakeStringObj (aCell, aString);
      }
      CATCH
      {
         ForgetObject (anObj);
      }
      ENDTRY;
      return anObj;
   }
   if (token == 2 || token == 3)
   {
      //
      // token is a value, so we need to
      // back up and evaluate the possible
      // expression
      //
      index = 1;
      value = GetExpression (anEntry, &index, 0);
      if(index > 0)
      {
         TRY
         {
            anObj = MakeValueObj (aCell, value, anEntry);
         }
         CATCH
         {
            ForgetObject (anObj);
         }
         ENDTRY;
         return anObj;
      }
      return NULL;
```

***ParseEntry** method code (concluded)*

```
    }
    else
    {
        return NULL;
    }
}
```

The **ParseEntry** method uses the exception-handling mechanism to guard against a situation in which there is not enough memory to allocate another object. If the CATCH section of the TRY-CATCH block is entered, it will ensure that the object is disposed of. The error is also propagated to the application, where the user is informed of it.

ParseEntry begins by calling a method called **GetToken**, which returns the type of the token that occurs next in the input string. **GetToken** is called with a pointer to the entry string, a pointer to an index value (initialized to 1), which it increments as it evaluates the string, and pointers to a value field and a string field. It returns a numeric integer that identifies the type of the token that is found. The following token types are defined:

1 Identifies a single-quote token

2 Identifies a value token

3 Identifies a left-parenthesis token

4 Identifies a right-parenthesis token

10 Identifies a + operator token

11 Identifies a – operator token

12 Identifies a * operator token

13 Identifies a / operator token

An unidentified token is returned as type 99. The **GetToken** method is unaware of the context in which these symbols are found; it merely registers the fact that they have the characteristics of a valid token. This method is what is commonly referred to as a *lexical analyzer* in discussions of compiler and interpreter applications.

If the **GetToken** method returns a token of 1, then the entry is assumed to be a string (because its first character is a single quote). In this case, the **ParseEntry** method creates a string object by calling the **MakeStringObj** method and returns its handle.

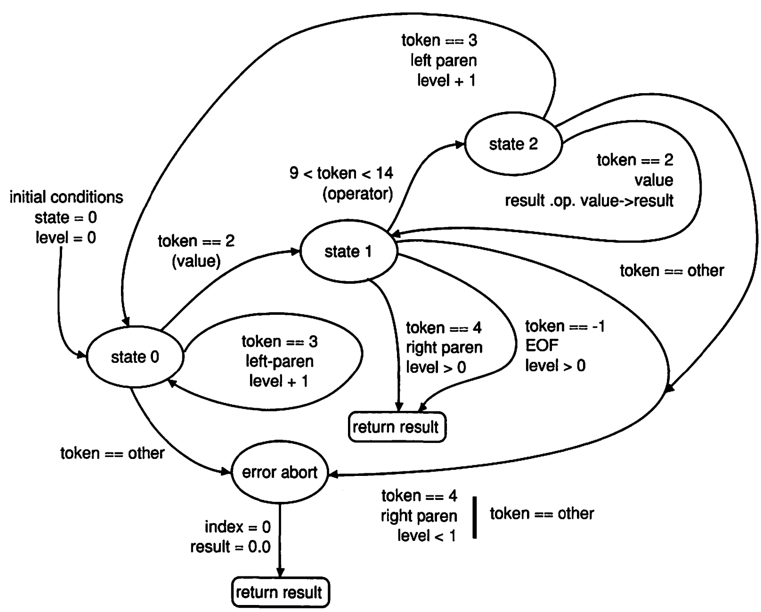
If the **GetToken** method returns a token of 2 (a value) or 3 (a left parenthesis), then the entry is assumed to be a formula, and the **GetExpression** method is called to evaluate the formula. If the evaluation is successful (i.e., the value of the **index** variable is nonzero upon return), then the **ParseEntry** method creates a value object by calling the **MakeValueObj** method and returns its handle.

If none of the foregoing are found, or if a failure occurs, a NULL object will be returned.

GetExpression Method Code

The **GetExpression** method is a miniature expression parser that uses a state transition scheme to evaluate the result of an expression. The diagram in Figure 8-1 shows approximately how the parser responds to tokens and how the entry formula is parsed. The parser begins with the index to the en-

Figure 8-1
state transitions of **GetExpression** parser



try string positioned at the first byte in the string. The **level** argument is set to zero on first entry. The following steps describe how the parser works:

1. The **value** and **result** values are set to **0.0** on each entry, and the **state** variable is set to **0**. The parser calls **GetToken** to get the next token. It then operates with this token according to the current state. If the state is **0**, then the parser expects either a numeric value or a left-parenthesis token. If neither of these is returned, the index is set to **0**, indicating that an error has occurred, and the parser returns to the **ParseEntry** method. If the token is a value, the **result** variable is set to the value and the state is advanced to **1**. If the token is a left parenthesis, the **GetExpression** method is called recursively, using the current index value, and the **level** argument is advanced by **1**.

2. In **state 1**, the parser is looking for an operator, a right parenthesis, or the end of the entry. If the next token is an operator, the parser immediately saves it and advances the **state** to **2**. If the token is a right parenthesis *and* the **level** is greater than **0**, then the parser immediately returns the current value of the **result** variable. Execution of the parser will continue in the step in which the parser called itself to processes the parenthetical group. If the token is the end-of-entry token (**-1**) *and* the **level** is greater than **0**, the parser immediately returns the **result**, resuming execution right after the point at which it called itself. Any other token or condition will result in the **index** value being set to **0** and a result of **0.0** being returned.

3. In **state 2**, the parser is looking for a left parenthesis or a value. If the next token is a left parenthesis, the parser calls itself using the current **index** value and advances the **level** by **1**. If the next token is a value, then the parser computes the result of the current value of the **result** variable and the **value**, by applying the saved **operator**. Addition, subtraction, multiplication, and division are allowed. The computed value is stored into the **result** variable, and the **state** is set to **1**. Any other token or condition will result in an error, the **index** being set to **0**, and a result of **0.0** being returned.

The code for the **GetExpression** method is as follows:

*Beginning of **GetExpression** method code*

```
double CCalcWindow::GetExpression (StringPtr anEntry, short *index, short
level)
{
   double   value  = 0.0, result = 0.0;
   short    token, state=0, operator;
   Str255   aString;

   while (TRUE)
   {
      token = GetToken (anEntry, index, &value, aString);
      switch (state)
      {
         case 0:  // initial case
         {
            if(token == 2)  // token is a value
            {
               result = value;
               state = 1;
               break;
            }
            if(token == 3)  // token is left paren '('
            {
               result = GetExpression (anEntry, index, level+1);
               if(*index == 0)
               {
                  return (0.0);
               }
               state = 1;
               break;
            }
            else
            {
               *index = 0;
               return 0.0;
            }
         }
         case 1:  // looking for an operator or EOF
         {
            if(token >= 10 && token <= 13)   // token is operator
            {
               operator = token;
               state = 2;
               break;
            }
            if(token == 4)  // token is right paren ')'
            {
```

GetExpression *method code (continued)*

```
        if(level < 1)
        {
            *index = 0;
            return (0.0);     // error
        }
        return result;
    }
    if(token == -1)
    {
        //
        // end of entry
        //
        if(level > 0)
        {
            return (result); // valid EOF
        }
        return result;
    }
    else
    {
        //
        // error
        //
        *index = 0;
        return (0.0);     // error
    }
}
case 2:  // looking for 2nd value
{
    if(token == 3)
    {
        value = GetExpression (anEntry, index, level+1);
        if(*index == 0)
        {
            return (0.0);
        }
    }
    if(token == 2 || token == 3) // token is value
    {
        switch(operator)
        {
            case 10:     // '+'
            {
                result += value;
                break;
            }
            case 11:     // '-'
            {
                result -= value;
```

GetExpression method code (concluded)

```
                    break;
                }
                case 12:    // '*'
                {
                    result *= value;
                    break;
                }
                case 13:    // '/'
                {
                    result /= value;
                    break;
                }
            }
            state = 1;
            break;
        }
        else
        {
            *index = 0;
            return (0.0);    // error
        }
      }
    }
  }
}
```

GetToken Method Code

The **GetToken** method is commonly referred to as a *lexical scanner* in the literature on compilers and interpreters. It scans the input, gets the next sequence of characters according to preset rules, and presents this to the parser as a token of a particular type and value.

The code for the **GetToken** method is as follows:

GetToken method code (beginning)

```
short CCalcWindow::GetToken (StringPtr anEntry, short *index,
                    double *value, StringPtr entry)
{
    char ch;
    Cell aCell;
    short length, numchars, type;
    Str255 cellText;

    length = anEntry[0];
    if(*index > length)
```

GetToken *method code (continued)*

```
{
   return (-1);      // end of entry
}
while (*index <= length)
{
   if(anEntry[*index] == ' ')
   {
      (*index)++;
      continue;
   }
   if(anEntry[*index] == '\"')
   {
      //
      // entry is a string
      //
      numchars = length - *index;
      BlockMove(&anEntry[*index+1], &entry[1], numchars);
      entry[0] = numchars;
      *value = 0.0;
      *index = length;
      return (1);      // token is string
   }
   if(anEntry[*index] == '(')
   {
      (*index)++;
      return (3);      // left parenthesis '('
   }
   if(anEntry[*index] == ')')
   {
      (*index)++;
      return (4);      // right parenthesis ')'
   }
   if(isConst(anEntry, index, value))
   {
      return (2);      // token is value
   }
   if(isCell(anEntry, index, &aCell))
   {
      //
      // token is a cell address
      //
      ((CList15 *)List15)->GetContents (aCell, entry, value, &type,
         cellText);
      if(type == -1)
      {
         *value = 0.0;
         *index = length;
         return (99);      // error
      }
```

***GetToken** method code (concluded)*

```
        else
        {
            //
            // token is an existing cell
            //
            return (2);  // value
        }
    }
    if(anEntry[*index] == '+')
    {
        (*index)++;
        return (10);      // addition
    }
    if(anEntry[*index] == '-')
    {
        (*index)++;
        return (11);      // subtraction
    }
    if(anEntry[*index] == '*')
    {
        (*index)++;
        return (12);      // multiplication
    }
    if(anEntry[*index] == '/')
    {
        (*index)++;
        return (13);      // division
    }
    else
    {
        *index = length;
        return (99);      // error
    }
  }
  return (-1);  // end of entry
}
```

The **GetToken** method for our worksheet is very straightforward. It contains a series of sections that test for various types of data in the entry string. The entry text is passed in as a string called **anEntry**, and the **index** variable is passed as a pointer whose associated value is updated as the method scans the string. Space characters are ignored between tokens. The steps taken by the method are as follows:

1. The method first tests for a single-quote (apostrophe) character, which signals the beginning of a string token.

If one is found, then the remainder of the entry is copied to the parameter called **entry**, the **value** parameter is set to **0.0**, and a token type of **1** is returned.

2. If the next nonblank character is a left parenthesis, a token type of **3** is returned.

3. If the next nonblank character is a right parenthesis, a token type of **4** is returned.

4. If the character is none of the preceding, then a method called **isConst** is called to determine whether the characters that follow are a numeric constant value. If so, then the **isConst** method returns TRUE, the updated **index** value, and the **value** of the constant. In this case, the **GetToken** method returns a token type of **2**. If the characters that follow are not a numeric constant, the method continues.

5. If the entry didn't contain a constant, a method called **isCell** is called, to determine whether the entry holds a valid worksheet cell designation (e.g., **B13**) as its next token. If so, **isCell** returns TRUE, the updated **index** value, and the number of the cell. The **GetContents** method is then called, and if the cell is nonempty, the value of its contents and a token type of **2** are returned. If the **isCell** method returns FALSE, the **GetToken** method continues.

6. Finally, **GetToken** tests for the operator values (+, –, *, and /). If any of these is found, the corresponding operator token type (10, 11, 12, or 13) is returned.

7. If none of the preceding tests discovers a valid token, a token type of **–1** is returned.

isConst Method Code

The **isConst** method is responsible for examining the characters in the entry string, beginning at the current position of the **index** variable and continuing until a valid constant or an invalid combination of characters is found. The code classifies the input as a valid constant if it appears in one of the following forms:

```
1234.567
 567
    .891
   0.123
```

As is apparent, constants *must* be unsigned, may have only an integral part, only a fractional part, or a combination of integral and fractional parts. The code for the **isConst** method is as follows:

***isConst** method code (beginning)*

```
Boolean CCalcWindow::isConst (StringPtr anEntry, short *index,
   double *value)
{
   short saved = *index, length, state = 0;
   unsigned char   ch, ch1, ch2, ch3;
   double result = 0.0, fraction = 0.0;
   short numFrac = 0, i;
   Boolean intVal = FALSE, fracVal = FALSE;
   length = anEntry[0];
   while (TRUE)
   {
      switch (state)
      {
         case 0:      // looking for integral value
         {
            ch = anEntry[*index];
            if(ch >= '0' && ch <= '9')
            {
               intVal = TRUE;
               result = (ch - '0');
               (*index)++;
               state = (*index > length) ? 5 : 1;
               continue;
            }
            else
            {
               state = 2;
               continue;
            }
            break;
         }
         case 1:      // get integral value
         {
            ch = anEntry[*index];
            if (ch >= '0' && ch <= '9')
            {
```

***isConst** method code (continued)*

```
            result = result * 10.0 + (ch - '0');
            (*index)++;
            if(*index > length) state = 5;
            continue;
        }
        else
        {
            state = 2;
            continue;
        }
        break;
    }
    case 2:      // check whether decimal
    {
        ch = anEntry[*index];
        if(ch == '.')
        {
            (*index)++;
            state = (*index > length) ? 5 : 3;
            continue;
        }
        else
        {
            state = 5;
            continue;
        }
        break;
    }
    case 3:      // verify digit following decimal
    {
        ch = anEntry[*index];
        if(ch >= '0' && ch <= '9')
        {
            fracVal = TRUE;
            numFrac++;
            fraction = (ch - '0');
            (*index)++;
            state = (*index > length) ? 5 : 4;
            continue;
        }
        else
        {
            state = 5;
            continue;
        }
        break;
    }
    case 4:      // collect fraction
    {
```

***isConst** method code (concluded)*

```
            ch = anEntry[*index];
            if(ch >= '0' && ch <= '9')
            {
                fraction = fraction * 10.0 + (ch - '0');
                numFrac++;
                (*index)++;
                if(*index > length) state = 5;
                continue;
            }
            else
            {
                state = 5;
                continue;
            }
            break;
        }
        case 5:     // create composite constant
        {
            if(fracVal)
            {
                for(i=0; i < numFrac; i++)
                {
                    fraction /= 10.0;
                }
            }
            if(intVal || fracVal)
            {
                *value = result + fraction;
                return TRUE;
            }
            *index = saved;
            *value = 0.0;
            return FALSE;
        }
    }
  }
}
```

The coding approach for analyzing the entry string to determine whether the next token is a numeric constant uses a state transition table design. The first character is guaranteed not to be a blank (because blanks were passed over by the **GetToken** method). The code starts with **state** set to **0**, the **result** and **fraction** variables set to **0.0**, and the **numFrac** variable set to **0**. The initial value in the **index** variable is saved. The steps for each state are as follows:

0. The next character is tested to determine whether it is numeric. If so, the result is set to the binary value of the digit, the **intVal** Boolean variable is set to TRUE, and the **index** is advanced and tested against the length of the entry. If it is greater than the length, **state** is set to **5**. Otherwise, **state** is set to **1**. If the character is not numeric, **state** is set to **2**.

1. The character at the **index** is tested to see whether it is numeric. If so, the previous **result** is multiplied by **10**, and the binary value of the digit is added to that **result**. The index is advanced, and if it exceeds the length of the entry, **state** is set to **5**. Otherwise, **state** remains the same. If the digit is not numeric, **state** is set to **2**.

2. The character at the **index** is tested to see whether it is a decimal point. If so, the **index** is advanced. If it exceeds the length of the entry, **state** is set to **5**. Otherwise, **state** is set to **3**. If the character is not a decimal point, **state** is set to **5**.

3. The character at the **index** is tested to determine whether it is numeric. If it is, the **fraction** variable is set to the binary value of the digit, the **fracVal** variable is set to TRUE, the **numFrac** variable is advanced by **1**, and the **index** is advanced and tested against the length of the entry. If it exceeds the length, **state** is set to **5**. Otherwise, **state** is set to **4**.

4. The character at the **index** is tested to determine whether it is numeric. If so, the previous **fraction** value is multiplied by **10** and the binary value of the digit is added to the **fraction** variable. Then the **numFrac** counter is advanced by **1**, and the **index** is advanced. If the **index** exceeds the length of the entry, **state** is set to **5**. Otherwise, **state** remains the same.

5. If the **fracVal** boolean variable is TRUE, the value of the **fraction** variable is successively divided by **10**, according to the count in the **numFrac** variable. If either the **intVal** or **fracVal** Boolean variable is TRUE, the final **value** is computed to be the sum of the **result** and **fraction** variable's values. In this case, the **isConst** method returns a TRUE value, along with the value of the result and the

updated index value. Otherwise, the **index** is set back to the value saved upon entry to the method, the **value** is set to **0.0**, and the method returns a FALSE value.

isCell Method Code

The **isCell** method is relatively straightforward. The intention is to determine whether the next token in the entry string is a valid cell number. Valid cells consist of a single alphabetic character (either upper- or lowercase), followed by a numeric value. Upon entry, the code saves the current value of the **index** variable, so that it can be restored if a valid cell isn't found. Next, the first character is tested to determine whether it is alphabetic. If not, the method restores the saved **index** value and returns a FALSE result. Then the column number is computed from the character value (A=0, B=1, etc.). If the next character is numeric, it and the digits following it are converted to a binary row value minus 1. If the column and row values don't exceed the **numCols** and **numRows** constants, respectively, the **(row, col)** value of the cell and a TRUE result are returned. Otherwise, the saved index value is restored and FALSE is returned. The code for the **isCell** method is as follows:

***isCell** method code (beginning)*

```
Boolean CCalcWindow::isCell (StringPtr anEntry, short *index, Cell *aCell)
{
    unsigned char ch;
    short row = 0, col = 0, saved, length;

    saved = *index;
    length = anEntry[0];
    if(*index <= length)
    {
        ch = anEntry[*index];
        if((ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z'))
        {
            col = (ch & ~0x20) - 'A';
            (*index)++;
            if(*index <= length)
            {
                ch = anEntry[*index];
                if(ch >= '0' && ch <= '9')
                {
                    for (row = 0; ch >= '0' && ch <= '9' && (*index <= length); )
                    {
                        row = row * 10 + (ch - '0');
```

***isCell** method code (concluded)*

```
                (*index)++;
                ch = anEntry[*index];
            }
            row--;
            if (row >= 0 && row <= numRows && col >= 0 &&
                 col <= numCols)
            {
                aCell->h = col;
                aCell->v = row;
                return TRUE;
            }
          }
        }
      }
    }
    *index = saved;
    return FALSE;
}
```

MakeStringObj Method Code

When the **ParseEntry** method discovers that the entry string contains a **string** value, it calls the **MakeStringObj** method to create a string object, whose handle it will store in the cluster, as shown in the following code:

```
CWSEntry * CCalcWindow::MakeStringObj (Cell aCell, StringPtr aString)
{
    CWSEntry  *aCellEntry;
    TRY
    {
        aCellEntry = new CWSEntry;
        aCellEntry->IWSEntry ();
        aCellEntry->SetWSCell (aCell);
        aCellEntry->SetWSType (1);   // string
        aCellEntry->SetWSText (aString);
        aCellEntry->SetWSEntry (aString);
        aCellEntry->SetWSValue (0.0);
    }
    CATCH
    {
        ForgetObject (aCellEntry);
    }
    ENDTRY;
    return aCellEntry;
}
```

The **MakeStringObj** method creates a new instance of the **CWSEntry** object, initializes the object, sets its instance variables to the cell number, sets the type code (1), sets both the text and entry variables to the entry string, and sets the value to 0.0. If successful, **MakeStringObj** returns the object's handle; otherwise, it calls **ForgetObject**. In case of failure, the error is propagated up to the application, where an alert informs the user of the error. The object's access methods are used to set its instance variable's values.

MakeValueObj Method Code

When the **ParseEntry** method discovers that the entry string contains a value (formula), it calls **GetExpression** to return the result of evaluating the formula. It then calls the **MakeValueObj** method to create an object whose handle will be stored in the worksheet cluster. The code for **MakeValueObj** is as follows:

```
CWSEntry * CCalcWindow::MakeValueObj (Cell aCell, double value,
                                       StringPtr aString)
{
   CWSEntry  *aCellEntry;
   Str255    dispStr;
   decform   aFormat;
   extended  temp;
   TRY
   {
      aCellEntry = new CWSEntry;
      aCellEntry->IWSEntry ();
      aCellEntry->SetWSCell (aCell);
      aCellEntry->SetWSType (2);      // type = value
      aCellEntry->SetWSValue (value);
      aCellEntry->SetWSEntry (aString);

      aFormat.style  = FIXEDDECIMAL;  // convert value to string
      aFormat.digits = 2;             // and store into cell
      x96tox80(&value, &temp);
      num2str(&aFormat, temp, dispStr);
      aCellEntry->SetWSText(dispStr);
   }
   CATCH
   {
      ForgetObject (aCellEntry);
   }
    ENDTRY;
   return aCellEntry;
}
```

Activate Method Code

The **Activate** method is an override of the same method inherited from the **CDirector** class. It calls the inherited method and then sends a **Refresh** message to the **List15** (main worksheet) pane. This forces the worksheet to be redrawn when the window is activated. In the process of redrawing the window, each of the cell values will be recalculated. The code for the **Activate** method is as follows:

```
void  CCalcWindow::Activate (void)
{
   inherited::Activate();
   if(List15 != NULL)
   {
      List15->Refresh();
   }
}
```

Adding the CWSEntryClass and Methods

In order to support the storage of information for each worksheet cell, and also provide methods to access that information, we have defined a new class, called **CWSEntry**. This class contains instance variables that define the contents of a cell's entry.

If you refer to the **MakeStringObj** (page 237) or **MakeValueObj** (page 238) method in the **CCalcWindow** class, you will see some of the access methods of the **CWSEntry** class being used to set the values of a **CWSEntry** object. In addition, the **GetCellText** and **GetContents** methods of the **CList15** class (see pages 204 and 208, respectively) refer to some of the **CWSEntry** access methods to retrieve the contents of the cell's instance variables.

By encapsulating the data and methods for a cell entry into a separate class, the definition of the class is completely hidden from the rest of the code. It can then be modified at will. As long as the current access methods remain supported, the nature and contents of the cell entry can be changed without regard to the remainder of the code. The declaration for the **CWSEntry** class is as follows:

```
class CWSEntry : public CCollaborator
{
protected:
    Cell      itsCell;
    short     itsType;
    Str255    itsString;
    Str255    itsText;
    double    itsValue;

public:
    void      IWSEntry(void);
    Cell      GetWSCell(void);
    short     GetWSType(void);
    void      GetWSEntry(StringPtr aString);
    void      GetWSText(StringPtr aString);
    double    GetWSValue(void);

    void      SetWSCell(Cell aCell);
    void      SetWSType(short aType);
    void      SetWSEntry(StringPtr aString);
    void      SetWSText(StringPtr aString);
    void      SetWSValue(double aValue);
};
```

Notice that the instance variables are all declared as **protected**. This will prevent classes other than direct descendants of **CWSEntry** from directly accessing these variables. Access methods are provided to get and set each variable, as is an initialization method to initialize an instance when it is created.

IWSEntry Method Code

This method performs initialization of a newly created **CWSEntry** instance. The code is as follows:

```
void  CWSEntry::IWSEntry (void)
{
    SetPt (&itsCell, 0, 0);
    itsType = -1;
    itsString[0] = 0;
    itsText[0] = 0;
    itsValue = 0.0;
}
```

CWSEntry Get Access Method Code

Each of the following access methods returns the corresponding instance variable value:

```
Cell  CWSEntry::GetWSCell (void)
{
   return itsCell;
}

short CWSEntry::GetWSType (void)
{
   return itsType;
}

void  CWSEntry::GetWSEntry (StringPtr aString)
{
   CopyPString (itsString, aString);
}

void  CWSEntry::GetWSText(StringPtr aString)
{
   CopyPString (itsText, aString);
}

double   CWSEntry::GetWSValue (void)
{
   return itsValue;
}
```

Although the above methods are extremely simple, it is important to stress the advantage of using them so that you will have the freedom to redesign the cell entries to incorporate new variables and features without affecting your current code.

CWSEntry Set Access Method Code

Each of the following access methods sets the value of its corresponding instance variable to the value passed as a parameter to the method:

***SetWS** access method code (beginning)*

```
void  CWSEntry::SetWSCell (Cell aCell)
{
   itsCell = aCell;
}
```

***SetWS** access method code (concluded)*

```
void  CWSEntry::SetWSType (short aType)
{
   itsType = aType;
}

void  CWSEntry::SetWSText (StringPtr aString)
{
   CopyPString (aString, itsText);
}

void  CWSEntry::SetWSEntry (StringPtr aString)
{
   CopyPString (aString, itsString);
}

void  CWSEntry::SetWSValue (double aValue)
{
   itsValue = aValue;
}
```

Viewing the Customized Results

After all the customizing has been applied to the modules, as described in this chapter, the Ensemble application can be recompiled and executed. The final result of all these efforts can be seen in Figure 8-2, which shows the Ensemble application with both windows containing appropriate sample contents. The EditText window is currently active in the figure.

Exercises

1. Explain why you think a **CCluster** object was chosen to hold the references to worksheet cell instances. Why wouldn't a **CList** or **CArray** work as well?

2. In the **DisposeData** method of the **CEnsembleData** class, the code continues to delete the first item in the cluster. Is this a typographical error? Shouldn't the code advance through the number of cells in the cluster, incrementing the item number each time?

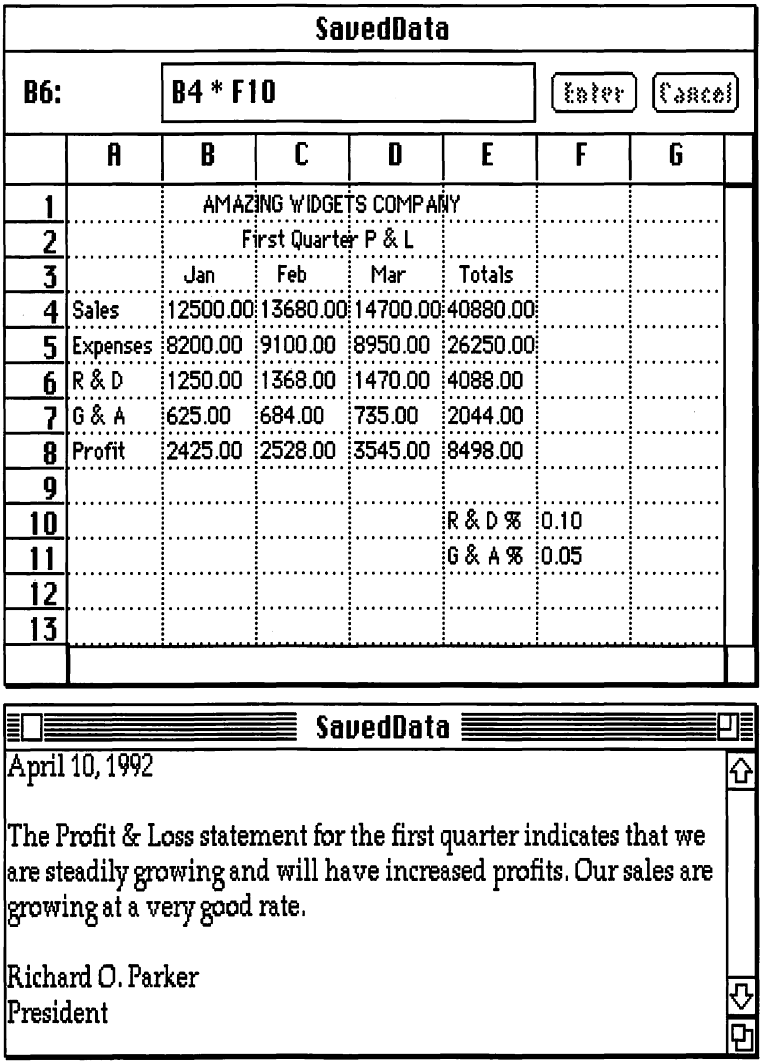


Figure 8-2 Customized Ensemble application running

3. Suggest what changes would have to be made to the various methods in the **CalcWindow** module to support more than 26 columns of data. Implement your suggestions.

4. Explain why the **CList15 GetCellText** method converts the value associated with a cell to an "extended" data type. What is gained by this approach? Explain the benefits of using the Macintosh SANE library functions.

5. How does the worksheet display keep track of changes to its cells? What fundamental mechanism of the TCL is used to implement this synchronization of the cluster and the worksheet display?

6. Describe the interaction of the column, row, and worksheet lists when the scroll bars are clicked or when either thumb is moved in the main worksheet pane. How are the entries scrolled in unison? What happens when the mouse is clicked and dragged through a set of entries in the main worksheet? How do the column and row lists reflect the range of visible cells when a drag selection is in progress?

7. Why was the **ProviderChanged** method in the **CList15** class written to override the standard behavior of that method in the **CTable** class? Why are we disregarding the **arrayInsertElement** and **arrayDeleteElement** messages? Explain what would have happened in our application if we had not overridden the method.

8. What modifications would be necessary to handle the selection of a contiguous series of cells in the worksheet, instead of the single selection now allowed? What benefits would result if this were implemented?

9. What modifications would be necessary to support the selection and operation of the worksheet with noncontiguous cell ranges? What benefit, if any, would this provide?[1]

10. Modify the worksheet code to add functions (such as **sum** and **average**) to the formula entry syntax. (*Hint*: Assign a new numeric token to represent each function, and then modify the **GetToken** lexical analyzer and the **ParseEntry** and **GetExpression** methods to parse and evaluate the new syntax.)

---

1. This question can best be answered in the context of the graphing facility, which is yet to be described; however, many spreadsheets on the market provide noncontiguous cell selections for other reasons. A very extensive extra-credit project or the subject could be undertaken either at this point or after the full application has been developed.

11. If the **CalcWindow** interface is modified to allow in-cell entry and editing of data, what methods would need to change? keeping in mind that provisions for entering, editing, canceling, and deleting an entry would be needed, how would the worksheet implementation have to be altered to provide these capabilities?[1] If selection of contiguous or noncontiguous entries, were allowed, how would these modifications affect the in-cell editing approach, if at all?

1. This could also be a very extensive extra-credit project. Bear in mind that a single pane can overlay a cell to obscure its current contents. The pane would have to be created "on the fly" and be sized to correspond exactly to the target cell size. It would also have to be placed in the correct position to appear to be applicable to the associated cell.

Chapter 9

Adding a Format Worksheet Dialog

This chapter describes the step-by-step method for adding a Format Worksheet dialog to Ensemble's user interface. In addition, after the dialog has been defined, a new command will be added to the **Format** menu, to allow the dialog to be opened.

There are several objectives to be met in the design of this dialog. The most important of these are:

- ❖ The contents of worksheet cells should be representable in any font, size, style, or alignment.
- ❖ Cell, column, and row styles should be selectable. In the case of columns and rows, the selected style should apply to every subsequent cell defined in that column or row.
- ❖ Worksheet column widths and row heights should be individually adjustable.
- ❖ The format of numeric cells must allow for the number of decimal digits specified by the user and the inclusion of commas or dollar signs, as desired.

Rather than apply the **Format Notebook** dialog to satisfy the first of these items and a completely different dialog for the remaining items, it was decided to design a single dialog that combines these functions. The following section describes the step-by-step procedure for creating the **Worksheet** dialog.

Creating the Worksheet Dialog

Creation of the **Worksheet** dialog is relatively straightforward. AppMaker's tools should be familiar by now, and given the desired appearance and the location and size of some of the important elements, it should be relatively easy to replicate the final dialog, shown in Figure 9-1.

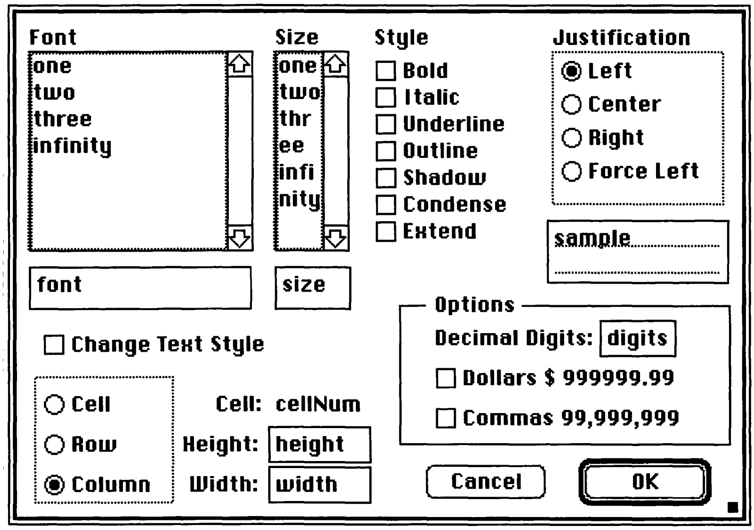


Figure 9-1
Appearance of final **Worksheet** dialog

Notice that the top portion of the dialog contains essentially the same elements as the previously defined **Notebook** dialog (although the height of the scrolling lists has been reduced to minimize the size of the dialog pane). While it might be tempting to copy and paste these elements into the current dialog, this is not recommended, as AppMaker will not provide unique identifiers for the individual elements (it will copy them exactly) which will lead to *multiply defined variable* compiler errors.

The completed dialog shown in Figure 9-1 is 296 pixels tall by 432 pixels wide. These dimensions allow it to fit on a compact Macintosh screen, although it will take up most of the 512-by-342-pixel screen area.

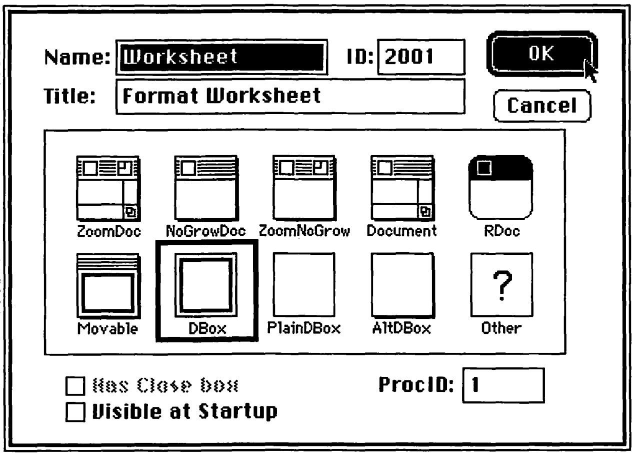
You should begin the process of creating the dialog by double-clicking on the **Ensemble.π.rsrc** resource file that was created in Chapter 6. This will serve as the basis for the new additions to the Ensemble application's user interface. By double-clicking on this file, you will launch AppMaker. The steps to create the dialog are as follows:

1. Pull down the **Select** menu and choose the **Dialogs** command.

2. Create a new dialog by pulling down AppMaker's **Edit** menu, and choose the **Create Dialog** command.

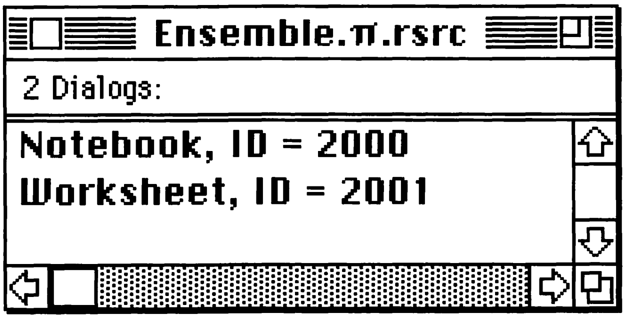
3. When the **Create Dialog** command is chosen, AppMaker will place a dialog information window on the screen. You should type the word **Worksheet** into the **Name** field, and type anything you like into the **Title** field. An example of the completed window is shown in Figure 9-2. Notice that we have not checked the **Visible at Start-up** selection. When the information has been entered, click the **OK** button, as shown.

Figure 9-2
Create Dialog information window



4. Figure 9-3 shows that the new **Worksheet** dialog, with ID=2001, appears in AppMaker's dialog selection list.

Figure 9-3
Worksheet dialog in AppMaker's selection list



Now that the dialog has been created, double-click it in AppMaker's selection list to make it active. At this point, you'll want to resize it so that it will be large enough to hold all the elements that we will be entering in the next series of steps. If you hold a ruler up to your screen and make the dialog approximately 6 inches wide by 4 inches tall, it will be just about the right size. The next series of steps covers adding some of the more complex elements to the dialog:

5. Add the **Font** table by pulling down the **View** menu and selecting **View Tools as Text**. Then, choose the **CScrollPane** tool and position the cursor (which is in the shape of a cross) at the approximate location of the top left corner of the **Font** list shown in Figure 9-1. Drag down and to the right to create a **CScrollPane** element that appears to be approximately the right size.

6. Pull down the **View** menu and choose **Item Info**, which will cause AppMaker to display a window containing the location and sizing information for the selected element. Adjust the settings for the **CScrollPane** element to match those in Figure 9-4.

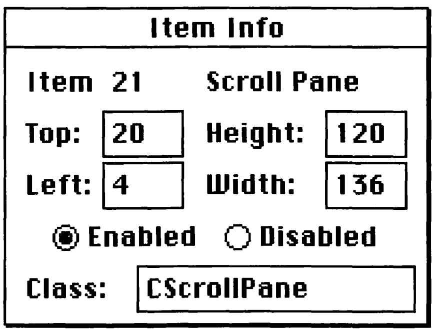


Figure 9-4
Item Info settings for **Font** CScrollPane

7. The next step involves placing a border inside the blank area of the **Font** scroll pane. Pull down the **Tools** menu and choose the **CBorder** tool. Position the center of the cross hairs of the cursor on top of the top left corner of the **Font** scroll pane, and drag down and to the right to enclose the blank portion of the scrollpane completely. The **Item Info** window should still be open. Change the settings to correspond with those shown in Figure 9-5.

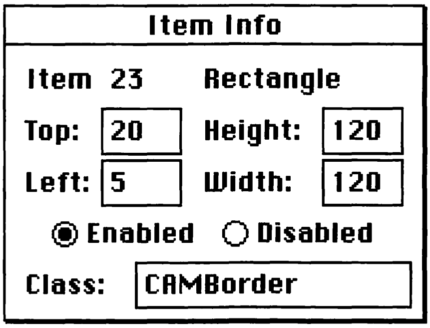


Figure 9-5
Item Info settings for **Font** CAMBorder

8. The final construction step for the **Font** table is the installation of the **CTable** element, inside the border that was just installed. Pull down the **Tools** menu and select the **CTable** tool. Click just inside the top left corner of the table's border, and drag down and to the right almost to the bottom right corner of the border. The **Item Info** settings for the **CTable** element are shown in Figure 9-6.

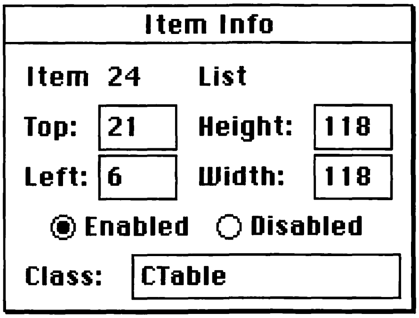
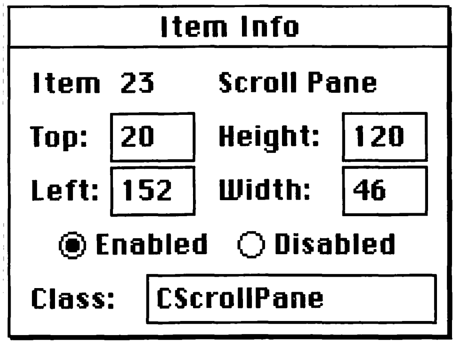


Figure 9-6
Item Info settings for **Font** CTable

9. The **Size** table is constructed in the next series of steps. The first step is to choose the **CScrollPane** tool, and click and drag the mouse so that the scroll pane is the approximate size and position shown in Figure 9-1. The final **Item Info** settings for the **Size** table's **CScrollpane** element are shown in Figure 9-8. Change the settings in AppMaker's **Item Info** window to correspond with those in the figure.

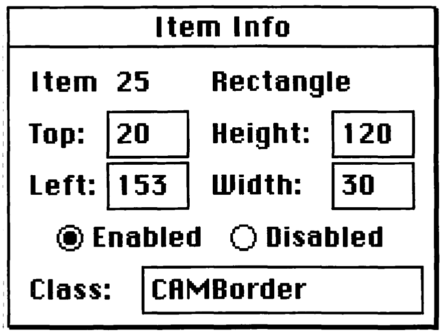
10. The next step is to place a border around the blank area of the scroll pane. Choose the **CBorder** tool, position the cursor's cross hairs at the top left corner of the **CScroll-**

Figure 9-7
Item Info settings for **Size** CScrollPane



Pane element, and drag down and to the right to enclose the blank portion of the scroll pane completely. The **Item Info** settings are shown in Figure 9-8.

Figure 9-8
Item Info settings for **Size** CAMBorder



11. The final step in the construction of the **Size** table is the placement of the **CTable** element inside the border. Choose the **CTable** tool, and then click the mouse cursor just inside the top left corner of the **CBorder** element that was installed in the previous step. Drag down and to the right, almost to the bottom right corner of the border. The **Item Info** settings for the **CTable** element are shown in Figure 9-9. Creation of the remaining elements is relatively straightforward. The **Item Info** settings for several of the less familiar elements are shown in the next few steps.

12. Create the column of checkbox items for the **style** settings by choosing the **CCheckbox** tool, clicking, and typing each checkbox's name, as shown in Figure 9-1. It

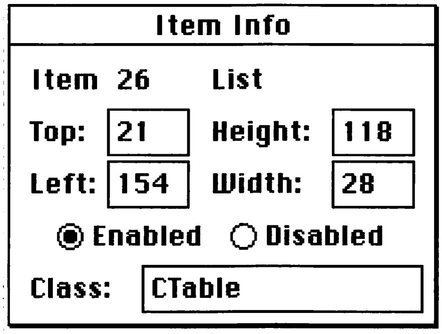


Figure 9-9
Item Info settings for **Size** CTable

should be relatively easy to approximate the appearance shown in the figure.

13. The group of radio buttons used to select the justification of worksheet cell entries is assembled by creating a **CRadioGroupPane** item, whose **Item Info** settings are shown in Figure 9-10. This pane groups all of the radio buttons

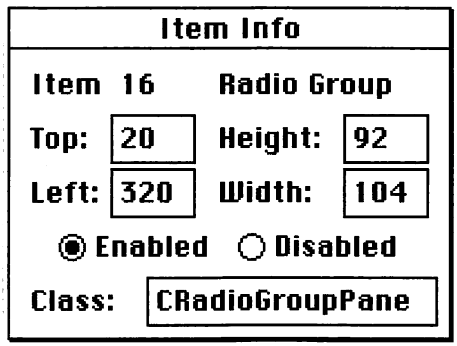


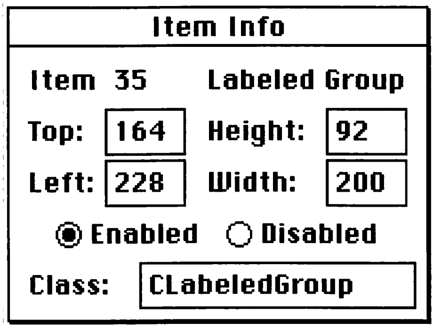
Figure 9-10
Item Info settings for **CRadioGroupPane**

into a single collection that works as a unit. The TCL ensures that one and only one button in the group is active. When the user selects an inactive button, the one that's currently active is deactivated and the new button is made active.

14. After the **CRadioGroupPane** has been installed and adjusted according to the indicated settings, choose the **CRadioControl** tool, click, and type in the names of the justification buttons, as shown in Figure 9-1. Make sure that all of the buttons are inside the **CRadioGroupPane** element.

15. The next element is a **CLabeledGroup**, so choose that tool. This creates another group of items, which are identified by a rectangular border, and an optional label. In this case, the label given to the group is **Options**. Position the cursor at the approximate top left corner of the group, and drag down and to the right until a group that is approximately the correct size has been created. Type the name **Options** into the top left (label) area of the group, and then change the **Item Info** settings to match those shown in Figure 9-11.

Figure 9-11
Item Info settings for **CLabeledGroup**



16. Choose the appropriate tools for creating the items inside the **Options** labeled group. The phrase **Decimal Digits** is a **CStaticText** item, and the text box associated with that is a **CDialogText** item. The two checkboxes that specify **Dollars** and **Commas** were created with the **CCheckbox** item.

17. Choose the **CDialogText** tool and create the **font**, **size**, and **sample** text boxes below the font list, size list, and justification group, respectively. Make these approximately the sizes shown in Figure 9-1.

18. Select the **CCheckbox** tool and create the single checkbox called **Change Text Style** below the font selection list. Just click and type the indicated name, as shown in Figure 9-1. This item provides the user the ability to bypass changing the text style and merely change a column width or row height if only that change is desired.

19. The final set of items allows the selection of the indicated cell, its associated column, or its associated row, to which the changes will be applied. The **Cell**, **Row**, and **Column** radio buttons are organized into a **CRadioGroupPane**, so choose that tool, create the pane, and then match its settings to those shown in Figure 9-12. The radio buttons are created with the **CRadioControl** tool.

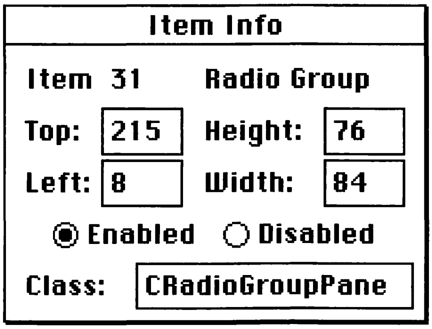


Figure 9-12
Item Info settings for **Cell, Row, Column CRadioGroupPane**

20. Create the items adjacent to the **Cell**, **Row**, and **Column** radio buttons as follows:

- The **Cell**, **Height**, and **Width** labels are created with the **CStaticText** tool, as is the **cellNum** text adjacent to the **Cell** label.
- The text boxes adjacent to the **Height** and **Width** labels are created as **CDialogText** items.

When the preceding steps are completed, the dialog should have the appearance shown in Figure 9-1. There are a lot of items in this dialog, and the contents are somewhat cramped, but it is sized to allow it to fit on the screen of a classic Macintosh and contain all of the relevant settings to modify the appearance of the worksheet.

As with the **Notebook** dialog, the text font, size, style, and justification can be changed for any worksheet cell, row, or column. If the style is changed (the **Change Text Style** checkbox is checked), the change applies only to the selected cell, row, or column of the worksheet.

The **Options** settings allow the number of digits appearing after a decimal point in numeric values to be specified. For numeric items, you can elect to have the digits grouped, with commas inserted at the appropriate places and with an optional leading dollar sign. If you enter 0 as the number of decimal digits, no decimal point will be displayed.

In the final customized code, a particular cell may have corresponding row and column styles. In this case, the column style will take precedence. If a style has been applied to an individual cell, then that style will take precedence over any existing or future row or column styles. This effect of applying the worksheet dialog will be fully realized in the code presented in Chapter 11.

Creating the Worksheet Menu Item

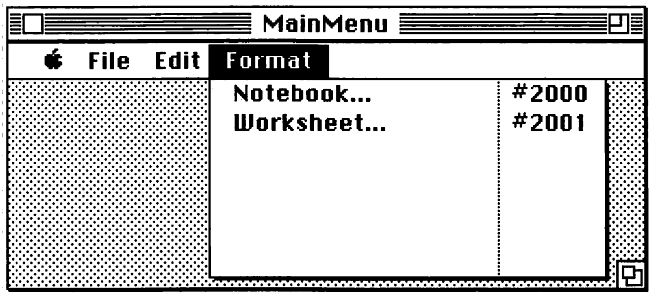
In the next series of steps, you will construct a new menu item in the **Format** menu. This will add the ability to open the dialog for formatting worksheet cells:

1. Pull down the **Select** menu and choose the **Menus** command.

2. Double-click on the **MainMenu** entry in AppMaker's selection window, and then click on the **Format** menu entry on the menu bar that is displayed.

3. Click below the **Notebook** entry and type **Worksheet...** (three periods follow the name), with a command number of **2001**, as shown in Figure 9-13. This completes the steps for adding the **Worksheet** command to the **Format** menu. When the command name and command number have been added to the menu, press the **Enter** key to indicate to AppMaker that the menu is complete.

Figure 9-13
Format Worksheet menu command added



Generating the Format Worksheet Code

After the preceding elements have been added to the user interface, you need to generate code that implements the default behavior for the elements. The next series of steps leads you through the process of generating code for the user interface changes you have just made:

1. Choose the **Generate** command from the **File** menu. AppMaker will display a dialog containing the names of all the files it intends to generate, as shown in Figure 9-14. Notice that there are four new file names in the list: **Worksheet.c**, **Worksheet.h**, **zWorksheet.c**, and **zWorksheet.h**. These files implement the default functionality of the **Worksheet** dialog. Click the **Generate** button, as shown, to generate new source code for all the listed files.

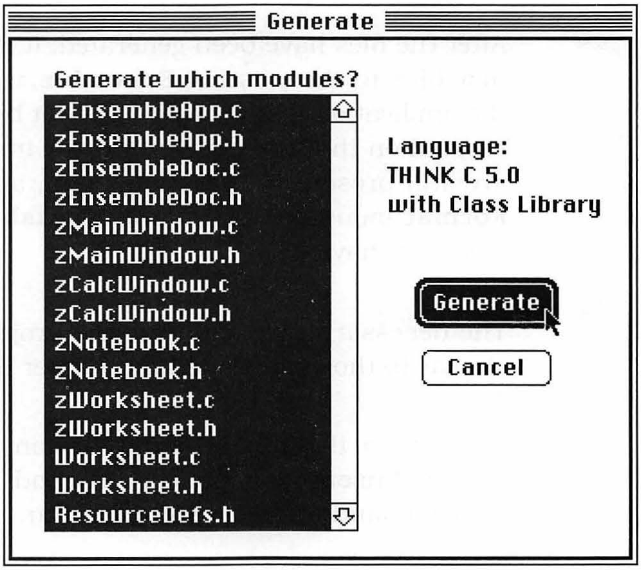


Figure 9-14 AppMaker's **Generate** dialog

2. After the files have been generated, pull down the **File** menu and choose the **Save** command, and then pull it down again and choose the **Quit** command to terminate execution of AppMaker. Notice that all new versions of the superclass files (whose names begin with the letter **z**) have been generated, in addition to the new **Worksheet.c**

and **Worksheet.h** subclass files. The complete set of files in the Ensemble folder is shown in Figure 9-15.

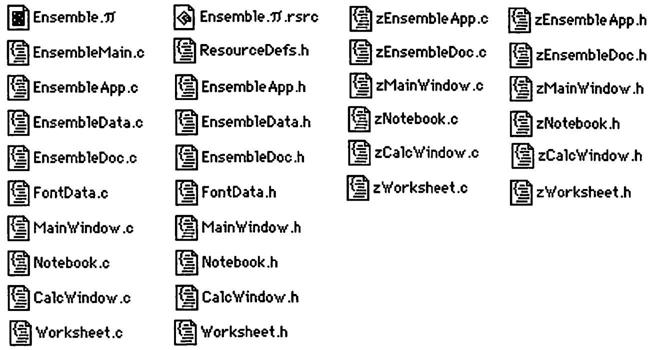


Figure 9-15
Ensemble files, as seen in the Finder

The next series of steps discusses how these files are added and recompiled in the THINK C project.

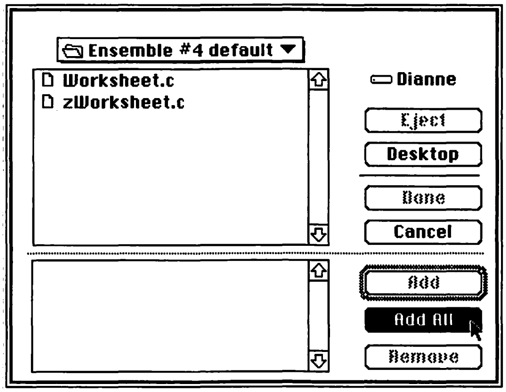
After the files have been generated, it is necessary to add any new files to the existing **Ensemble.π** project and recompile the application to see its new default behavior. It is important to mention that all of the previously implemented capabilities are still present in the application, and only the additional **Format** menu command and **Worksheet** dialog will exhibit default behavior.

The necessary steps to bring the project up to date are very similar to those presented in Chapter 6:

1. Launch the THINK C application by double-clicking on the **Ensemble.π** project file, and then choose the **Add** command from the **Source** menu.

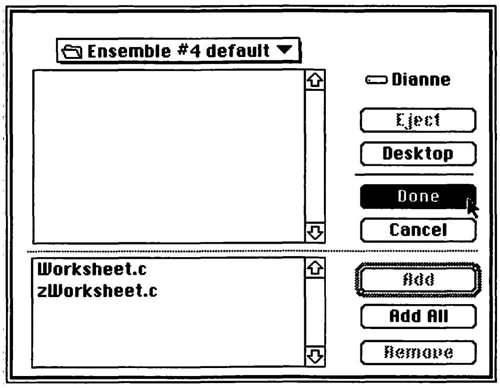
2. When the **Add** command is selected, THINK C will display a dialog in its upper portion listing any files in the current project that have not yet been added. This is shown in Figure 9-16. Notice that only the '**.c**' files are shown. Their corresponding '**.h**' header files will also be added, automatically, to the project. Click the **Add All** button, as shown in the figure.

Figure 9-16
Adding the new files to the Ensemble project



3. After all the files have been added, their names will appear in the bottom portion of the dialog. Click the **Done** button, as shown in Figure 9-17.

Figure 9-17
Selecting **Done** after adding all the new files



4. Notice that all of the files have been added to the **Ensemble.π** project file, as shown in Figure 9-18.

5. The next step is to compile the files that need recompilation. This is accomplished by pulling down the **Source** menu and choosing the **Make** command. It is necessary to use the **Make** command, rather than the **Bring Up To Date** command from the **Project** menu, because, as far as THINK C is concerned, none of the existing files has been modified.

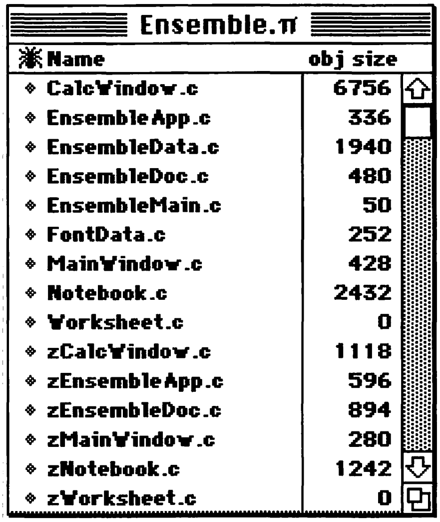


Figure 9-18
All files added to **Ensemble.π** project

6. When the **Make** command is chosen, THINK C will display the dialog shown in Figure 9-19. You should click the **Use Disk** button, as shown in the figure, to force THINK C to examine the modification dates of the files and determine which ones have really been changed since it executed the last compilation of the project. As long as you modify files while inside the THINK C environment, it will keep track of which ones need to be recompiled. When you modify files outside the environment, you have to tell THINK C explicitly to look for modified files.

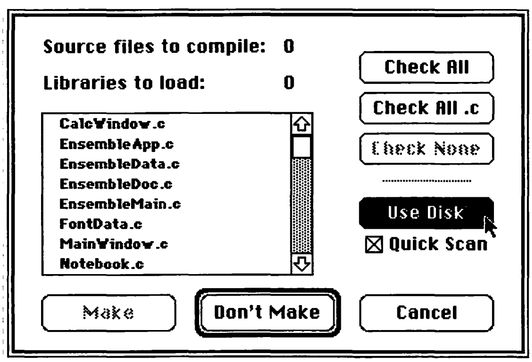


Figure 9-19
Click **Use Disk** to check for modified files

7. THINK C will scan the disk, looking for files that have been modified since its last update of the project, and will

place check marks next to their names in the dialog. It will also highlight the **Make** button at this point. You should click **Make** to instruct THINK C to compile the marked files, as shown in Figure 9-20.

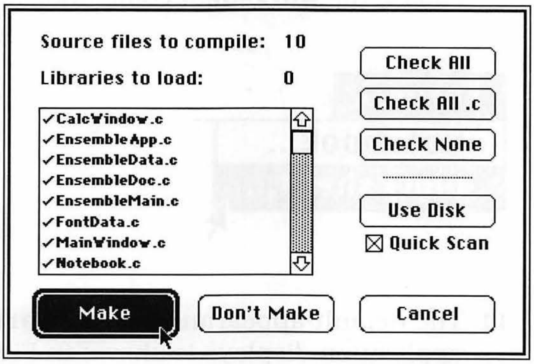


Figure 9-20
Clicking **Make** to recompile the modified files

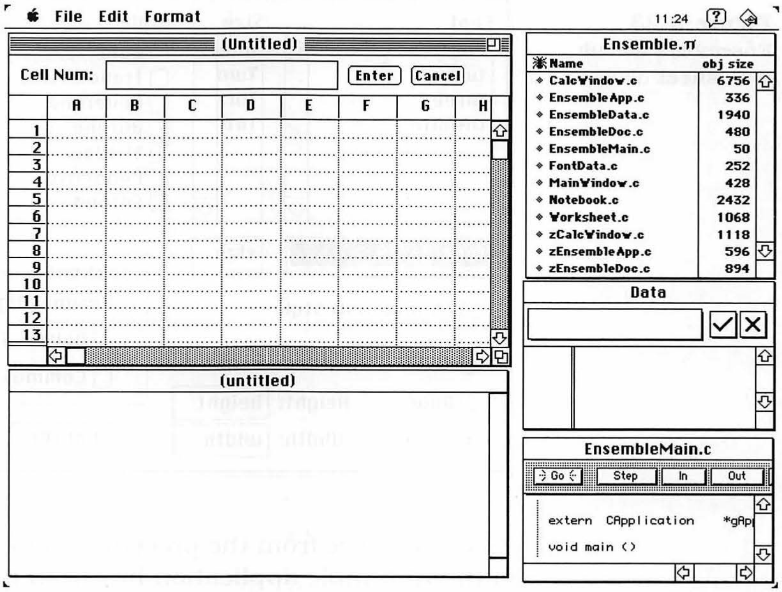


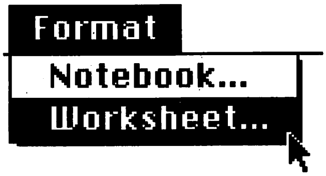
Figure 9-21
Revised Ensemble application running

8. THINK C will recompile all the modified files and will update the project with the latest code. The application is ready to run inside the THINK C environment at this point. To execute the application, pull down the **Project** menu and choose the **Run** option.

9. Figure 9-21 shows the revised application running, along with THINK C's project and debugging windows.

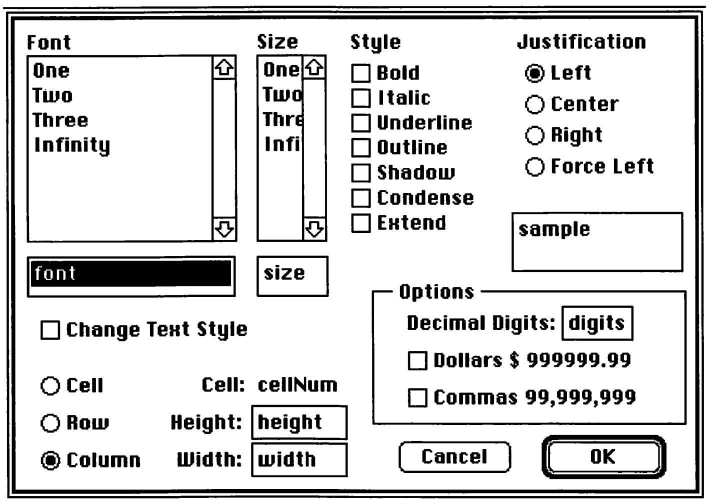
10. To verify that AppMaker has generated code to enable the **Worksheet** dialog, pull down the **Format** menu and choose the **Worksheet** command, as shown in Figure 9-22.

Figure 9-22
Choosing Ensemble's **Worksheet** command



11. The default appearance of the **Worksheet** dialog that the application displays is shown in Figure 9-23.

Figure 9-23
Ensemble's default **Worksheet** dialog



As you can see from the preceding steps, the full functionality of the Ensemble application has been retained and the **Worksheet** dialog has been added. The next chapter will discuss the newly generated code resulting from these additions, and the chapter after that will describe the custom code additions that fully implement the **Worksheet** dialog's functionality. This is a good place to stop and take stock of the major changes that have been added to the application with very little effort.

Exercises

1. Modify the style of the individual text items in the column of **Style** checkboxes to conform to the style names that they suggest. (*Hint*: Look at AppMaker's **Text Style** dialog for an example of this.)

2. Explain why the **Change Text Style** checkbox is necessary in the Worksheet dialog.

3. Explain why names such as **font** and **size** are typed into the EditText fields when, in fact, they are defined inside AppMaker. What is the result if these fields are left blank?

4. Determine what changes would be necessary to "internationalize" the Worksheet dialog.[1] Implement these changes.

---

1. Creating an international version of an application has far-reaching consequences. There are numerous features of the Ensemble application, such as its menus, that would need to be changed. Examining this topic as a standard part of a course in software development is highly recommended.

Chapter 10

Examining the Format Worksheet Code

This chapter examines the code generated by AppMaker in response to the added user interface elements described in Chapter 9: a **Worksheet** dialog and a new **Worksheet** command in the **Format** menu. These additions have resulted in AppMaker's generation of four new files, which have been added to the Ensemble application project in the steps described in that chapter. The function of each file is as follows:

- **Worksheet.c** contains the subclass methods for the **Worksheet** dialog. It is the file that we will be customizing to a great extent. The generated code, to be described shortly, provides us with an extremely good skeletal module to which our custom code will be applied.

- **Worksheet.h** contains the class declarations for the subclass definitions generated into the **Worksheet.c** file. We will be examining the generated class declarations in it and adding new declarations when we customize the code.

- **zWorksheet.c** contains the superclass methods for the **Worksheet** dialog. It contains the important code for creating and initializing each of the user interface elements in the dialog. As with all superclass files, we will not be making any changes to this code.

- **zWorksheet.h** contains the declarations for the superclass methods contained in the **zWorksheet.c** file. We will not be modifying any of these declarations.

In addition to the new code included in the foregoing files, a few methods are affected in the preexisting superclass files.

Table 10-1 shows the generated code to be described in this chapter.

Table 10-1
Customized methods to implement the I/O for EditText and spreadsheet data

Class	Method	Description
ZEnsembleDoc	UpdateMenus	Enables **Worksheet** command
ZEnsembleDoc	DoCommand	Handles **Worksheet** command
ZWorksheet	IZWorksheet	Initialize worksheet dialog items
ZWorksheet	various misc. UpdateMenus	Lists creation methods and Update Menus method
Worksheet.c global function	DoWorksheet	Subclass method to manage the **Worksheet** dialog
CList24	various	Initializes and gets cell text
CList28	various	Initializes and gets cell text
CWorksheet	IWorksheet	Initializes **Worksheet** dialog
CWorksheet	UpdateMenus	Updates related menus
CWorksheet	DoCommand	Handles worksheet-related commands
CWorksheet	ProviderChanged	Handles BroadcastChange messages in the **Worksheet** dialog

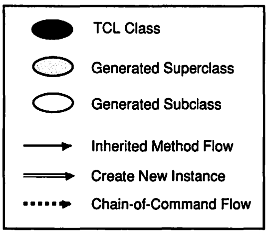
It is worthwhile to continue to emphasize that although AppMaker generates new code for all of the superclass files, it will never touch the code in any of the subclass files.

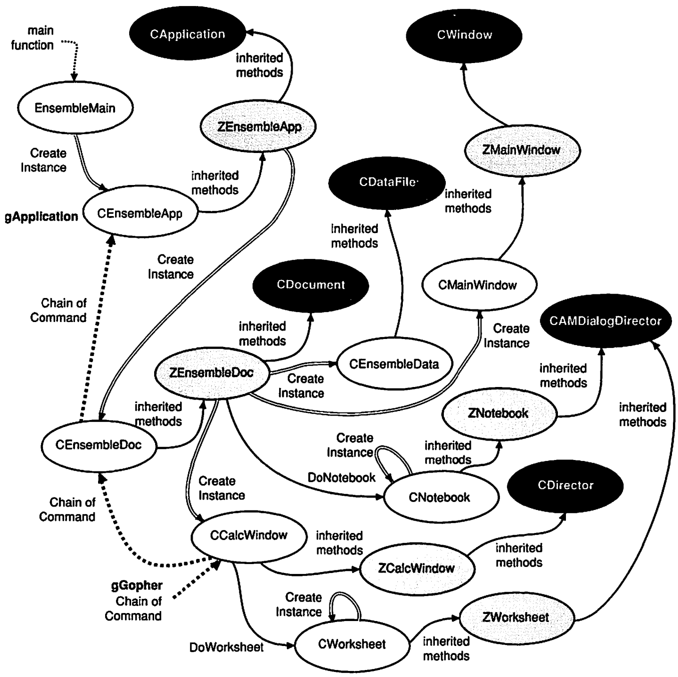
Because the **Worksheet.c** and **Worksheet.h** files did not previously exist, they were generated to contain routines that will be modified when we customize the Ensemble application to implement the worksheet styling features fully. In future code-generation sessions, however, AppMaker will refrain from modifying the subclass files.

The New Ensemble Application Structure

After the code has been generated (and with some license with regard to the point of connection of the **Worksheet** dialog), the new code structure for the Ensemble application appears as depicted in Figure 10-1.

Figure 10-1
Ensemble Application structure with **Worksheet** classes added





Although the generated code appears to attach the creation of the **CWorksheet** instance to the **ZEnsembleDoc** class, this is not how the **CWorksheet** instance will eventually be connected. The figure shows the state of the application with the **CalcWindow** active, and in that state, the **gGopher** variable points to the **CCalcWindow** instance. It is within the **DoCommand** method of the **CCalcWindow** class that the **Worksheet** dialog will be invoked.

Examining the ZEnsembleDoc Code Changes

Management of the document's windows is a responsibility of the **CEnsembleDoc** and **ZEnsembleDoc** classes. When a command is added to one of the document-related menus, the responsibility for any needed management and action associated with that command is properly vested in the **ZEnsembleDoc** class.

In most cases, AppMaker will automatically generate the appropriate code. If necessary, the generated code can be overridden in a corresponding method in the subclass file. In the case of the **Worksheet** dialog and its corresponding menu command, the generated code located in the **ZEnsembleDoc** superclass is just fine. The newly generated code for the **UpdateMenus** method is as follows:

```
void ZEnsembleDoc::UpdateMenus(void)
{
   inherited::UpdateMenus ();
   gBartender->EnableCmd (cmdNotebook);
   gBartender->EnableCmd (cmdWorksheet);
}
```

In this code, AppMaker has generated an additional message to the **gBartender** instance to enable the **Worksheet** command in the **Format** menu. This behavior will be slightly modified in the custom code, but only in the **UpdateMenus** method for the subclass.

When the **Worksheet** command is chosen by the user from the **Format** menu, the **Bartender** sends a **DoCommand** message, with the command code, to the current instance held in the **gGopher** variable. If the **CalcWindow** window is active, the message will first be passed to the **DoCommand** method in the **CCalcWindow** module. If that method does not handle the message, it will eventually be passed on until it arrives in the **DoCommand** method in the **ZEnsembleDoc** superclass. We will provide code for handling this message in the customized version of the code, presented in the next chapter. Although the generated code for this message will never be executed, it is worthwhile to examine it:

DoCommand *method code (beginning)*

```
void ZEnsembleDoc::DoCommand(long   theCommand)
{
   switch (theCommand)
   {
      case cmdNotebook:
            DoNotebook (this);
         break;
      case cmdWorksheet:
            DoWorksheet (this);
         break;
```

DoCommand
method code
(concluded)

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

AppMaker has no information to the effect that the **Worksheet** command isn't intended for use with the main document (as is the case with the **Notebook** command), so it generates code to invoke the dialog in the **ZEnsembleDoc** superclass.

It happens that we will override the **Notebook** command in the **CEnsembleDoc** subclass and will handle the **Worksheet** command in the **CCalcWindow** class, but this is just an appropriate decision for the Ensemble application.

Examining the Generated Code for ZWorksheet

Mainly through its initialization method, the superclass code for the **Worksheet** dialog is responsible for creating the worksheet window and all of its associated user interface elements (lists, checkboxes, radio buttons, etc.). In addition, the superclass contains an **UpdateMenus** method to forward the **UpdateMenus** message to its **CDirector** ancestor in the TCL.

The code for initializing the **Worksheet** dialog is quite lengthy, as there are quite a number of user interface elements to instantiate and initialize. Nevertheless, we show it in its entirety:

IZWorksheet
method code
(beginning)

```
void ZWorksheet::IZWorksheet(CDirectorOwner   *aSupervisor)
{
    CView   *enclosure;
    CBureaucrat   *supervisor;

    inherited::IAMDialogDirector (WorksheetID, aSupervisor);

    enclosure = itsWindow;
    supervisor = itsWindow;

    OKButton = new CAMButton;
    OKButton->IViewRes ('CtlP', 146, enclosure, supervisor);
    CancelButton = new CAMButton;
```

IZWorksheet
method code (continued)

```
CancelButton->IViewRes ('CtlP', 147, enclosure, supervisor);

FontLabel = new CAMStaticText;
FontLabel->IViewRes ('AETx', 136, enclosure, supervisor);

SizeLabel = new CAMStaticText;
SizeLabel->IViewRes ('AETx', 137, enclosure, supervisor);

StyleLabel = new CAMStaticText;
StyleLabel->IViewRes ('AETx', 138, enclosure, supervisor);

BoldCheck = new CAMCheckBox;
BoldCheck->IViewRes ('CtlP', 150, enclosure, supervisor);

ItalicCheck = new CAMCheckBox;
ItalicCheck->IViewRes ('CtlP', 151, enclosure, supervisor);
UnderlineCheck = new CAMCheckBox;
UnderlineCheck->IViewRes ('CtlP', 152, enclosure, supervisor);
OutlineCheck = new CAMCheckBox;
OutlineCheck->IViewRes ('CtlP', 153, enclosure, supervisor);

ShadowCheck = new CAMCheckBox;
ShadowCheck->IViewRes ('CtlP', 154, enclosure, supervisor);

CondenseCheck = new CAMCheckBox;
CondenseCheck->IViewRes ('CtlP', 155, enclosure, supervisor);

ExtendCheck = new CAMCheckBox;
ExtendCheck->IViewRes ('CtlP', 156, enclosure, supervisor);

JustificationLabel = new CAMStaticText;
JustificationLabel->IViewRes ('AETx', 139, enclosure, supervisor);

fontField = new CAMDialogText;
fontField->IViewRes ('ADTx', 131, enclosure, supervisor);

sizeField = new CAMDialogText;
sizeField->IViewRes ('ADTx', 132, enclosure, supervisor);

Group16 = new CRadioGroupPane;
Group16->IViewRes ('Pane', 132, enclosure, supervisor);
CenterRadio = new CAMRadioControl;
CenterRadio->IViewRes ('CtlP', 157, Group16, Group16);
RightRadio = new CAMRadioControl;
RightRadio->IViewRes ('CtlP', 158, Group16, Group16);
ForceLeftRadio = new CAMRadioControl;
ForceLeftRadio->IViewRes ('CtlP', 159, Group16, Group16);
LeftRadio = new CAMRadioControl;
LeftRadio->IViewRes ('CtlP', 160, Group16, Group16);
```

IZWorksheet
method code
(continued)

```
ScrollPane21 = new CScrollPane;
ScrollPane21->IViewRes ('ScPn', 137, enclosure, supervisor);

    Rect23 = new CAMBorder;
    Rect23->IViewRes ('Bord', 139, ScrollPane21, supervisor);

    List24 = NewList24 ();
    List24->IViewRes ('ATbl', 141, Rect23, supervisor);

ScrollPane21->InstallPanorama (List24);

ScrollPane25 = new CScrollPane;
ScrollPane25->IViewRes ('ScPn', 138, enclosure, supervisor);

    Rect27 = new CAMBorder;
    Rect27->IViewRes ('Bord', 140, ScrollPane25, supervisor);

    List28 = NewList28 ();
    List28->IViewRes ('ATbl', 142, Rect27, supervisor);

ScrollPane25->InstallPanorama (List28);

sampleField = new CAMDialogText;
sampleField->IViewRes ('ADTx', 133, enclosure, supervisor);

HeightLabel = new CAMStaticText;
HeightLabel->IViewRes ('AETx', 148, enclosure, supervisor);

WidthLabel = new CAMStaticText;
WidthLabel->IViewRes ('AETx', 149, enclosure, supervisor);

CellLabel = new CAMStaticText;
CellLabel->IViewRes ('AETx', 150, enclosure, supervisor);

heightField = new CAMDialogText;
heightField->IViewRes ('ADTx', 140, enclosure, supervisor);

widthField = new CAMDialogText;
widthField->IViewRes ('ADTx', 141, enclosure, supervisor);

Group35 = new CRadioGroupPane;
Group35->IViewRes ('Pane', 135, enclosure, supervisor);
RowRadio = new CAMRadioControl;
RowRadio->IViewRes ('CtlP', 171, Group35, Group35);
CellRadio = new CAMRadioControl;
CellRadio->IViewRes ('CtlP', 172, Group35, Group35);
ColumnRadio = new CAMRadioControl;
ColumnRadio->IViewRes ('CtlP', 173, Group35, Group35);
OptionsGroup = new CLabeledGroup;
```

***IZWorksheet* method code (concluded)**

```
    OptionsGroup->IViewRes ('LGrp', 128, enclosure, supervisor);

    DecimalDigitsLabel = new CAMStaticText;
    DecimalDigitsLabel->IViewRes ('AETx', 147, OptionsGroup,
        supervisor);

    digitsField = new CAMDialogText;
    digitsField->IViewRes ('ADTx', 138, OptionsGroup, supervisor);

    Dollars99999999Check = new CAMCheckBox;
    Dollars99999999Check->IViewRes ('CtlP', 169, OptionsGroup,
        supervisor);
    Commas99999999Check = new CAMCheckBox;
    Commas99999999Check->IViewRes ('CtlP', 170, OptionsGroup,
        supervisor);

    cellNumLabel = new CAMStaticText;
    cellNumLabel->IViewRes ('AETx', 151, enclosure, supervisor);

    ChangeTextStyleCheck = new CAMCheckBox;
    ChangeTextStyleCheck->IViewRes ('CtlP', 174, enclosure,
        supervisor);
}
```

The first action of the **IZWorksheet** method is to call the **IAMDialogDirector** method, which is supplied in a library provided with the AppMaker product. **IAMDialogDirector** creates a new dialog window, with the location and size specified in the **Ensemble.π.rsrc** resource file.

When the **IAMDialogDirector** method returns, a new dialog window will have been established, to which the current **IZWorksheet** method can add instances of all the user interface elements. AppMaker automatically generates code to create instances of every user interface element and then initializes each element from its corresponding resource template settings (calling the **IViewTemp**) method. Each item is named according to its stated name (e.g., Font, which results in the creation of a **CAMStaticText** field with the name **FontLabel**) or by the type of object it represents (e.g., borders are named beginning with the word **Rect** and lists with the word **List**).

We have typed names into all the **CDialogText** fields, so that AppMaker will give them those names when it generates code. For example, the font name text field, which will show the name of the currently selected font, was named **font** when we created the dialog. AppMaker's generated code for this element names it **fontField**. If you fail to type a name into these **CDialogText** fields, AppMaker will give them names like **Field1**, **Field2**, etc.

When the **IZWorksheet** method completes execution, all of the user interface elements will have been created and placed into the dialog's window. The code to display the dialog is contained in the **DoWorksheet** function, which will be discussed later in the chapter.

Two additional routines that are involved with the creation of the user interface elements are generated in the **ZWorksheet** class. These are the **NewList24** and **NewList28** methods, whose code is as follows:

```
// The only purpose of this function is so that you can override it
// to create the list as your subclass of CAMTable
CAMTable  *ZWorksheet::NewList24(void)
{
   CAMTable    *theList;

   theList = new CAMTable;
   return (theList);
}

/*----------*/
// The only purpose of this function is so that you can override it
// to create the list as your subclass of CAMTable
CAMTable  *ZWorksheet::NewList28(void)
{
   CAMTable    *theList;

   theList = new CAMTable;
   return (theList);
}
```

Note that AppMaker has also generated comments that indicate the purpose of generating these functions in the superclass module. Their only purpose is to be overridden in the subclass module. You will also find that the subclass implementation creates a subclass of these lists, in order to override their **GetCellText** method (as required by the TCL). The **IViewTemp** method is also overridden in the subclass, so that special initialization code can be added for these tables. This will all be covered later.

The remaining method, generated into the **ZWorksheet** class, is **UpdateMenus**, whose code is as follows:

```
void ZWorksheet::UpdateMenus(void)
{
   inherited::UpdateMenus ();
}
```

As is readily apparent, the **UpdateMenus** method merely calls its inherited **UpdateMenus** method. This allows the TCL to handle enabling or disabling any menu commands at the higher level.

It is important to point out that most methods call an inherited method before they perform any unique, special functions. This is to allow the TCL to perform its intended functions before more detailed, context-sensitive functions are performed. Because each call to the inherited method occurs first, the highest level of the TCL is the first to perform any actions on the specified object. When it completes its intended functions, control reverts to the next lower level, and so forth, down to the lowest subclass method in the calling hierarchy. The lowest subclass always has the last word. This is a very powerful feature of object-oriented programming.

Examining the Code for the Worksheet Subclass

All of the remaining generated code, related to the new **Worksheet** menu command and its associated dialog, is contained in the **Worksheet.c** and **Worksheet.h** files. The latter contains the class declarations for the **CWorksheet**, **CList24**, and **CList28** classes, as well as a prototype for the **DoWorksheet** global function. The declaration for the **CWorksheet** class is as follows:

***CWorksheet** class declaration (beginning)*

```
class CWorksheet : public ZWorksheet
{
public:
   virtual void IWorksheet(CDirectorOwner   *aSupervisor);
   void         UpdateMenus(void);    // is override
```

***CWorksheet** class declaration (concluded)*

```
    void        DoCommand (long   theCommand); // is override

protected:
    void        ProviderChanged(CCollaborator *aProvider,
                        long     reason,
                        void*    info);      // is override
    CAMTable  *NewList24(void);          // is override
    CAMTable  *NewList28(void);          // is override
}
```

The declaration provides three public methods, callable by any class method, and three protected methods, callable only by methods in the **CWorksheet** class or its subclasses.

In addition to the **CWorksheet** class declaration, there are declarations for the **CList24** and **CList28** classes. These will be shown later. Of immediate concern is the **DoWorksheet** global function, which is called in the **DoCommand** method of the **ZEnsembleDoc** class, as shown on page 268.

The generated code for the **DoWorksheet** function is responsible for creating the **CWorksheet** object, initializing the user interface elements contained in the dialog, and managing the execution of the dialog, indicating to the program in which manner the user chose to terminate the dialog's execution (either by clicking the **OK** or the **Cancel** button). The code is as follows:

***DoWorksheet** method code (beginning)*

```
void DoWorksheet(CDirectorOwner   *aSupervisor)
{
    CWorksheet   *dialog;
    long      dismisser;

    dialog = NULL;
    TRY
    {
        dialog = new CWorksheet;
        dialog->IWorksheet (aSupervisor);

        /* initialize dialog panes */

        dialog->BeginDialog ();
        dismisser = dialog->DoModalDialog (cmdOK);

        if (dismisser == cmdOK)
        {
```

DoWorksheet method code (concluded)

```
            /* extract values from dialog panes */
        }

        dialog->Dispose ();
    }
    CATCH
    {
        ForgetObject (dialog);
    }
    ENDTRY;
}
```

The first action of the **DoWorksheet** function is to create an instance of the **CWorksheet** object and then store this into a local variable called **dialog**. The function then calls the subclass **IWorksheet** method for that object, which in turn calls the **IZWorksheet** method that was displayed in the code listing beginning on page 269. After the initialization is complete, the generated code suggests, via a comment, that you perform any custom initialization of the dialog's user interface elements. Following this, the generated code calls the **BeginDialog** and **DoModalDialog** methods, to show and operate the dialog, respectively. The variable **dismisser** contains either the value **cmdOK** or **cmdCancel** upon return from operating the dialog. The generated code suggests, via a comment, that you place the code to extract values from the dialog panes after it has been determined that the user dismissed the dialog by clicking the **OK** button. The **DoWorksheet** function sends a **Dispose** message to the **dialog** prior to returning to the caller. In the event that an error is detected, the code in the CATCH block will be executed.

Each of the lists (font and size) in the dialog is represented by a custom class definition. The **Worksheet.h** file contains the declarations for these classes, as follows:

***CList24** and **CList28** class declarations (beginning)*

```
class CList24 : public CAMTable
{
public:
    void IViewTemp(CView      *anEnclosure,
                   CBureaucrat *aSupervisor,
                   Ptr         viewData);  // is override
    void GetCellText (Cell    aCell,
                   short      availableWidth,
                   StringPtr  itsText);  // is override
};
```

***CList24** and **CList28** class declarations (concluded)*

```
class CList28 : public CAMTable
{
public:
   void IViewTemp(CView    *anEnclosure,
                  CBureaucrat *aSupervisor,
                  Ptr      viewData);  // is override
   void GetCellText(Cell   aCell,
                  short    availableWidth,
                  StringPtr  itsText);  // is override
};
```

Note that for each of these two class declarations an **IViewTemp** and **GetCellText** method is defined. Each of these is an override of the corresponding method in the TCL. It is necessary to override the TCL to perform the appropriate initialization and return the correct cell text value.

The code for the corresponding **CList24** and **CList28** methods is identical. The code for the **IViewTemp** and **GetCellText** methods for **CList24** is as follows:

***IViewTemp** and **GetCellText** method code (beginning)*

```
void  CList24::IViewTemp (CView        *anEnclosure,
                          CBureaucrat  *aSupervisor,
                          Ptr          viewData)
{
   inherited::IViewTemp (anEnclosure, aSupervisor, viewData);

   // any additional initialization for your subclass
   AddRow (4, 0);       // e.g., add 4 rows at the beginning of the list

}

voidCList24::GetCellText (Cell         aCell,
                          short        availableWidth,
                          StringPtr    itsText)
{
   // replace with your own code which uses the cell coordinates
   // to access your private data structures,
   // then convert the cell data to a Str255
   switch (aCell.v) {
      case 0:
            CopyPString ("\pOne", itsText);
         break;
      case 1:
            CopyPString ("\pTwo", itsText);
         break;
      case 2:
```

***IViewTemp** and **GetCellText** method code (concluded)*

```
            CopyPString ("\pThree", itsText);
        break;
    default:
            CopyPString ("\pInfinity", itsText);
        break;
    };
}
```

This code is very similar to the corresponding code shown for the **Notebook** dialog's list classes, beginning on page 79. AppMaker initializes each list with four rows of cells, and the template is already initialized to create a single column of data. All of this code in both methods is intended as an example only. It will be completely replaced in our custom code that is described in Chapter 11.

The code for the **CList28** class's **IViewTemp** and **GetCell-Text** methods is identical to what has been generated for the **CList24** class, except that the class name in the method definition headers is **CList28**.

The **CWorksheet** subclass contains several methods that complete the picture of the added functionality. The first of these is the **IWorksheet** method, which was called by the **DoWorksheet** function (page 275), to perform any special initialization of the dialog's elements. The code for **IWorksheet** is as follows:

```
void CWorksheet::IWorksheet(CDirectorOwner   *aSupervisor)
{
    inherited::IZWorksheet (aSupervisor);

    // any additional initialization for your dialog
}
```

As is apparent, the generated code merely calls the inherited **IZWorksheet** code, which was shown on page 269. We will be adding further initialization code to **IWorksheet** as shown in the next chapter.

The code to override the creation of the two list objects (**CList24** and **CList28**) is identical with the exception of the class names. The code to create the **CList24** object is as follows:

```
CAMTable *CWorksheet::NewList24 (void)
{
   CList24 *theList;

   theList = new CList24;
   return (theList);
}
```

For the **CList24** and **CList28** objects, new instances are created. Both of these are subclasses of AppMaker's **CAMTable** library class.

The **CWorksheet** class also contains an **UpdateMenus** method, which merely calls the inherited method. The generated code is as follows:

```
void CWorksheet::UpdateMenus(void)
{
   inherited::UpdateMenus ();
}
```

Two particularly important methods complete the code generation for the **CWorksheet** subclass. The first of these is the **DoCommand** method, whose code is fairly lengthy:

DoCommand *method code (beginning)*

```
voidCWorksheet::DoCommand(longtheCommand)
{
   switch (theCommand) {
      case cmdBoldCheck:
            /* DoBoldCheck ();*/
         break;
      case cmdItalicCheck:
            /* DoItalicCheck ();*/
         break;
      case cmdUnderlineCheck:
            /* DoUnderlineCheck ();*/
         break;
      case cmdOutlineCheck:
            /* DoOutlineCheck ();*/
         break;
      case cmdShadowCheck:
            /* DoShadowCheck ();*/
         break;
      case cmdCondenseCheck:
```

DoCommand
method code
(concluded)

```
            /* DoCondenseCheck ();*/
        break;
    case cmdExtendCheck:
            /* DoExtendCheck ();*/
        break;
    case cmdCenterRadio:
            /* DoCenterRadio ();*/
        break;
    case cmdRightRadio:
            /* DoRightRadio ();*/
        break;
    case cmdForceLeftRadio:
            /* DoForceLeftRadio ();*/
        break;
    case cmdLeftRadio:
            /* DoLeftRadio ();*/
        break;
    case cmdRowRadio:
            /* DoRowRadio ();*/
        break;
    case cmdCellRadio:
            /* DoCellRadio ();*/
        break;
    case cmdColumnRadio:
            /* DoColumnRadio ();*/
        break;
    case cmdDollars99999999Check:
            /* DoDollars99999999Check ();*/
        break;
    case cmdCommas99999999Check:
            /* DoCommas99999999Check ();*/
        break;
    case cmdChangeTextStyleCheck:
            /* DoChangeTextStyleCheck ();*/
        break;
    default:
            inherited::DoCommand (theCommand);
        break;
  }
}
```

The default-generated code for the **DoCommand** method provides the ability to take action whenever a checkbox or radio button is clicked, while the user is operating the dialog. This gives us the opportunity to change the appearance of the dialog items when these events occur. The **DoCommand** method is called for these actions because the resource templates for the checkbox and radio button objects contain "click commands," which are transparently assigned when the object's **IViewTemp** method is called. We will be enhancing **DoCommand** in the next chapter to provide visible changes to the dialog when these commands are dispatched.

The final method, one that provides very powerful "hooks" for processing user-created events, is called **ProviderChanged**. This method is called whenever a user interface element sends a **BroadcastChange** message to the TCL. The generated code for this method is also fairly long:

ProviderChanged *method code (beginning)*

```
void CWorksheet::ProviderChanged(CCollaborator *aProvider,
                                 long     reason,
                                 void*    info)
{
    if (aProvider == fontField) {
        if (fontField->GetLength () == 0) {
            // text is empty
        } else {
            // there is some text
        }
    }
    if (aProvider == sizeField) {
        if (sizeField->GetLength () == 0) {
            // text is empty
        } else {
            // there is some text
        }
    }
    if (aProvider == List24) {
        if (List24->HasSelection ()) {
            // perhaps activate some buttons
        } else {
            // perhaps deactivate
        }
    }
    if (aProvider == List28) {
        if (List28->HasSelection ()) {
            // perhaps activate some buttons
        } else {
            // perhaps deactivate
        }
    }
    if (aProvider == sampleField) {
        if (sampleField->GetLength () == 0) {
            // text is empty
```

ProviderChanged *method code (concluded)*

```
        } else {
            // there is some text
        }
    }
    if (aProvider == heightField) {
        if (heightField->GetLength () == 0) {
            // text is empty
        } else {
            // there is some text
        }
    }
    if (aProvider == widthField) {
        if (widthField->GetLength () == 0) {
            // text is empty
        } else {
            // there is some text
        }
    }
    if (aProvider == digitsField) {
        if (digitsField->GetLength () == 0) {
            // text is empty
        } else {
            // there is some text
        }
    }
}
```

The **ProviderChanged** method is called whenever a keystroke occurs in one of the TextEdit fields or when any of the font or size list elements is selected. We will be enhancing its code to provide visual feedback when the list selections occur. The code relating to the text fields will be deleted.

Exercises

1. Examine Figure 10-1 and note the difference between the ancestors of the **CMainWindow** and **CCalcWindow** classes. Explain why these differ and in what ways they are the same. (*Hint*: Examine the class hierarchy in the THINK C Browser window.)

2. Describe the similarities in the generated code for the **Notebook** dialog and the **Worksheet** dialog. Explain in what way these similarities are beneficial to the develop-

ment of large applications that include many dialog boxes in their user interface designs.

3. Explain the need for the **NewList$_i$** methods, for each of the lists, in both the *superclass* and *subclass* files. What is the distinction between the two methods?

4. Explain the purpose of the *unusable* code generated into the **GetCellText** methods. In what way does this aid in the future customizing of these methods?

5. At this point in the Ensemble application's development, what customization would be needed for the **DoCommand** method in the **CWorksheet** class. Don't peek into the next chapter to formulate your answer.

6. What sort of customization would be needed for the **CWorksheet** class's **ProviderChanged** method? Why is code generated to handle empty text fields and fields that contain text? What purpose could handling an empty text field serve?

Chapter 11

Customizing the Format Worksheet Code

This chapter describes the additions and modifications to the source files that implement the Ensemble application, with the intent of showing how the **Worksheet** menu and dialog box functionality are fully implemented. The changes concern a limited number of (both source and header) files in the application, including **CEnsembleData**, **CalcWindow**, and **Worksheet**. In addition, we will create entirely new source and header files called **CellData.c** and **CellData.h**. These implement a new class of data describing the style characteristics of a row, column, or cell instance.

Because we wish to save the style information associated with the worksheet rows, columns, and cells, we will be modifying the **ReadData**, **WriteData**, **ReadStyles**, **WriteStyles**, **ReadWSEntries**, and **WriteWSEntries** methods in the **CEnsembleData** class. Of course, this will make any existing spreadsheet files incompatible with the new format; however, to convert the existing files, we first modified only the **WriteData** and **WriteWSEntries** methods, then added the **WriteStyles** method, and, finally, wrote out the existing file that had been read with the older **ReadData** method. We then added the changes to the **ReadData** and **ReadWSEntries** methods and then coded the **ReadStyles** method so that the new format could be read.

This book examines the development of a single application version (albeit in an evolutionary manner), and it does not behoove us to have subsequent files be compatible with files written in an earlier format. If this were necessary, we would make provisions to ease the transition from one version to another. For example, we could easily add a code at the beginning of the file to indicate the version of the application with which it was written and then add the necessary code in the I/O methods to handle the differences between the various versions.

Adding a CCellData Class

We continue our practice of encapsulating new data entities (as with the **FontData** class) by creating a new class to hold the distinctive information for the **Worksheet** styles. The header file, containing the declarations of the **CCellData** class, is as follows:

```
/* CellData.h -- CCellData class */
#define _H_CellData
#include <CObject.h>

//
// cell info structure definition
//
typedef struct
{
    short isDefault;
    short cellMetric;
    short fontNumber;
    short fontSize;
    short fontStyle;
    short fontAlign;
    short decimalDigits;
    short commas;
    short dollars;

} cellInfo;

class CCellData : public CObject
{
public:

    cellInfo  cellData;
    void      ICellData(cellInfo styleInfo);
};
```

The **CCellData** class contains a single instance variable, which is a structure holding all the important style information associated with a row, column, or cell. The fields of the structure are initialized by the **ICellData** method, and these fields are public to facilitate their modification by methods in the other classes. The **ICellData** method code is as follows:

```
void  CCellData::ICellData(cellInfo styleInfo)
{
   cellData.isDefault  = styleInfo.isDefault;
   cellData.cellMetric = styleInfo.cellMetric;
   cellData.fontNumber = styleInfo.fontNumber;
   cellData.fontSize   = styleInfo.fontSize;
   cellData.fontStyle  = styleInfo.fontStyle;
   cellData.fontAlign  = styleInfo.fontAlign;
   cellData.decimalDigits = styleInfo.decimalDigits;
   cellData.commas = styleInfo.commas;
   cellData.dollars = styleInfo.dollars;
}
```

Basically, the intention is that the method creating the **CCellData** instance will supply the initial values for the structure's fields. We will be illustrating how this is done in a number of different circumstances.

For example, if a **CCellData** instance is being created (for a newly created cell entry), and no style information currently exists for the cell, row, or column, then the style information must be obtained by querying the CTable instance to determine the default font, style, size, and alignment and also to provide default settings for the **Options** (decimal digits, commas, and dollar sign).

If the newly created **CCellData** instance is associated with a cell, column, or row for which style information is already available, then the appropriate existing style information is passed to the initialization method.

In all cases, if a cell style already exists, it takes precedence over a column style, which, in turn, takes precedence over a row style. A style can be applied to a cell without previously creating an entry for that cell. In that case, a dummy entry is created that shows up as a blank cell in the worksheet.

Customizing the CEnsembleData Code

As previously stated, the modifications to the **CEnsemble-Data** class, to implement the new *styled text* worksheet entries, are limited to the methods shown in Table 11-1.

Table 11-1
Customized methods to implement text styles for the worksheet data

Class	Method	Description
CEnsembleData	IEnsembleData	Provides new lists for row & column styles
CEnsembleData	WriteData	Writes text and worksheet document data
CEnsembleData	WriteStyles	Writes style data
CEnsembleData	WriteWSEntries	Writes worksheet entries
CEnsembleData	ReadData	Reads text and worksheet document data
CEnsembleData	ReadStyles	Reads style data
CEnsembleData	ReadWSEntries	Reads worksheet entries
CEnsembleData	**GetHList** **GetVList**	Accesses column and row style lists from other classes

Modifying the Initialization Code

The existing initialization code for the **CEnsembleData** class provides for the creation of a **CCluster** object to hold the worksheet array data. The revised **IEnsembleData** method must also make provision for storing style information pertaining to entire rows or columns of the worksheet.

We decided to create two new **CList** objects, called **itsHList** and **itsVList**, to contain the column and row style information, respectively. The handles to these lists will also be passed to the **CalcWindow** code when requested, to provide access to the style information in that class. As has been our practice from the beginning, *ownership* of all the data is vested in the **CEnsembleData** class.

IEnsembleData Method Code

The new version of the **IEnsembleData** method is as follows:

```
voidCEnsembleData::IEnsembleData(CDocument*theDocument)
{
   inherited::IDataFile ();
   hasFile = FALSE;
   itsDocument = theDocument;
   itsEditTextData = NULL;

   //
   // allocate the main worksheet cluster and the
   // H-Label & V-Label lists and initialize them
   //
   itsCluster = new CCluster;
   itsCluster->ICluster();
   itsHList = new CList;
   itsHList->IList();
   itsVList = new CList;
   itsVList->IList();
}
```

The **CList** class was chosen to contain the row and column style data because this "container class" provides ordered storage of the data. To locate the entry for the *i*th row or column, the code will be able to use a direct, random-access method. We will keep an entry in the list for *every* row and column. This requires that we maintain only 76 entries with the current dimensions of our worksheet (26 columns + 50 rows).

Modifying the Input/Output Code

Table 11-1 shows the input/output methods that require modification for reading and writing the worksheet style entries. Before presenting the modified methods, it will be useful to show the modified file format, so that the code used in the methods will make more sense.

The file format has been changed only to support the addition of the row and column style entries. The worksheet entries themselves are also larger than in the previous implementation, to include style information for individual cells. The easiest way to present the new format is to show a layout diagram for the file (Figure 11-1).

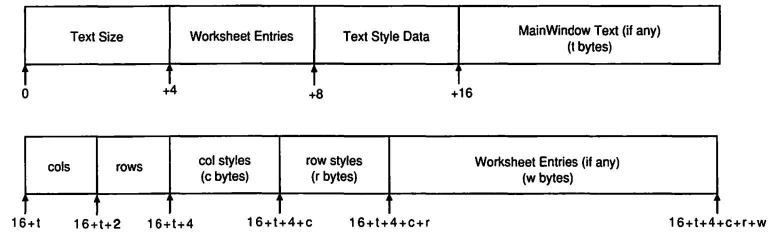


Figure 11-1
Ensemble file format

The file begins with two 4-byte integers, which define the length of the text in the **MainWindow** and the number of worksheet entries. If no text is present, then the text length will be 0, and no **Text Style Data** or **MainWindow Text** will follow. Instead, the worksheet-related data (if any) will follow. If no worksheet data are present, then the Ensemble file will contain only the two 4-byte integer values (each of which will be 0).

The **Text Style Data**, if present, consists of the font, size, style, and alignment information for the text to be displayed in the **MainWindow**. The **MainWindow Text** is the actual text to be displayed.

The worksheet-related data, if present, consists of two 2-byte integer values that indicate the number of columns and rows, respectively, in the worksheet. These values are written to the file to make allowances for different-size worksheet definitions. Following the column and row sizes are the **col styles** and **row styles** entries, which specify the styles of each of the (currently defined) 26 columns and 50 rows.

The **Worksheet Entries** are last. The initial 4-byte worksheet entry count specifies how many entries are contained in this last section of the file.

WriteData Method Code

The first section of code in the new version of the application is the main **WriteData** method. This has been modified only slightly, to allow writing the worksheet style data to the file. The code for this method is almost identical to that for the version described in Chapter 8, except for the addition of the call to the **WriteStyles** method. The new code is as follows:

```
Boolean CEnsembleData::WriteData(void)
{
   CMainWindow *theTextWindow;
   long          textLength, WSEntryCt, fileLength;
   fontInfo      theFontInfo;

   theTextWindow = ((CEnsembleDoc *)itsDocument)->GetTextWindow();
   itsEditTextData = theTextWindow->GetEditTextHandle();
   textLength = GetHandleSize(itsEditTextData);
   WSEntryCt = itsCluster->GetNumItems();
   FailOSErr (SetFPos( refNum, fsFromStart, 0L));
   WriteSome ((Ptr)&textLength, (long) sizeof(long));
   WriteSome ((Ptr)&WSEntryCt,  (long) sizeof(long));
   if(textLength > 0)
   {
      theFontInfo = ((CEnsembleDoc *) itsDocument)->theTextData
         ->GetFontData();
      WriteSome ((Ptr)&theFontInfo, sizeof (fontInfo));
      WriteSome (*itsEditTextData, textLength);
   }
   if(WSEntryCt > 0)
   {
      WriteStyles ();
      WriteWSEntries (WSEntryCt);
   }
   fileLength = GetLength();
   FailOSErr(SetEOF( refNum, fileLength));
   FailOSErr( FlushVol( NULL, volNum));
   return (TRUE);
}
```

The first action of the **WriteData** method is to send a message to the **CEnsembleDoc** class to get a handle to the text window instance (**GetTextWindow**). Once the text window handle is returned, the **WriteData** method can access the text itself, by sending the window a **GetEditTextHandle** message. The handle to the EditText data is stored into the instance variable **itsEditTextData**. Getting the size of this handle into the local 4-byte (long) **textLength** variable tells us how many bytes of text must be written.

Getting the number of worksheet entries is much easier and is accomplished by sending a **GetNumItems** message to the

itsCluster instance and storing the returned value in the 4-byte (long) **WSEntryCt** variable.

When the previous actions are complete, we know whether the file will contain text and/or worksheet data. The method proceeds by setting the file position to the beginning of the file and then writing the contents of the two 4-byte variables.

The next section of code tests whether the **textLength** is greater than 0, and if so, the text style data are retrieved from the document and written to the file, followed by the text itself. If the **textLength** is 0, then nothing is written in this section of the file.

The section that follows tests whether the **WSEntryCt** value is greater than 0, and if so, it calls the **WriteStyles** method, followed by the **WriteWSEntries** method. If no worksheet entries exist, then nothing is written in this section of the file.

The final section of the code gets the length of the file, sets the logical end-of-file marker to the point corresponding to the length, and calls the **FlushVol** method to ensure that any data remaining in the file's buffer have been written out to the disk.

WriteStyles Method Code

The **WriteStyles** method is responsible for writing the row and column style entries if worksheet entries are present in the current worksheet window. In the present implementation, we first write out the number of columns and rows in the worksheet and then write the style entries for the columns and rows. An entry is written for each column and row, for a total of 76 entries (26 columns and 50 rows). The **cols** and **rows** values allow for a different number of columns and rows, enabling files to be interchanged between two users who employ different worksheet sizes. The code for the **WriteStyles** method is as follows:

***WriteStyles** method code (beginning)*

```
void  CEnsembleData::WriteStyles(void)
{
   CCellData  *aStyle;
   short         cols, rows;
   short         index;
   cols = itsHList->GetNumItems();
```

***WriteStyles** method code (concluded)*

```
    rows = itsVList->GetNumItems();
    WriteSome((Ptr)&cols, sizeof(short));
    WriteSome((Ptr)&rows, sizeof(short));
    for(index=0; index < cols; index++)
    {
        aStyle = (CCellData *)itsHList->NthItem (index+1);
        WriteSome((Ptr)&index, sizeof(short));
        WriteSome((Ptr)&aStyle->cellData, sizeof(cellInfo));
    }
    for(index=0; index < rows; index++)
    {
        aStyle = (CCellData *)itsVList->NthItem (index+1);
        WriteSome((Ptr)&index, sizeof(short));
        WriteSome((Ptr)&aStyle->cellData, sizeof(cellInfo));
    }
}
```

The **WriteStyles** code first accesses the number of columns and rows in the spreadsheet, by sending each corresponding list a **GetNumItems** message, and then writes the returned values to the file as 2-byte (short) integers. Following this, a loop is used to write out each of the column style entries, followed by a loop to write out each of the row style entries. The entries are accessed from their corresponding lists, in order, by column or by row.

WriteWSEntries Method Code

The **WriteWSEntries** method has been modified to write out the style information for each cell, in addition to the cell's entry string, as was described on page 195. The revised code is as follows:

***WriteWSEntries** method code (beginning)*

```
void  CEnsembleData::WriteWSEntries (long entryCount)
{
    WSCellEntry   anEntry;
    short         index;
    long          WSEntryCt;
    Str255        entryData;
    CWSEntry      *aWSEntry;

    for(index = 1; index <= entryCount; index++)
    {
        itsCluster->GetItem (&aWSEntry, index);
        FailNIL (aWSEntry);
        anEntry.WSCell = aWSEntry->GetWSCell();
```

WriteWSEntries *method code (concluded)*

```
        anEntry.WSType = aWSEntry->GetWSType();
        anEntry.WSStyle = aWSEntry->GetWSStyle();
        aWSEntry->GetWSEntry(entryData);
        anEntry.WSSize = entryData[0];
        WriteSome ((Ptr)&anEntry, sizeof(WSCellEntry));
        WriteSome ((Ptr)&entryData[1], (long) entryData[0]);
    }
}
```

The difference between the new and the previous code is the call to the new **GetWSStyle** access method and the storage of the data from the style instance into the **WSCellEntry** structure. The new definition of the **WSCellEntry** structure is contained in the **EnsembleData.h** file and has been enhanced from the version shown on page 190. The new format of the **WSCellEntry** structure is as follows:

```
typedef struct
{
    Cell      WSCell;
    short     WSType;
    cellInfo  WSStyle;
    short     WSSize;

} WSCellEntry;
```

Note that the only difference between the two structure definitions is the inclusion of the **WSStyle** field in the one and its absence in the other.

The **WriteWSEntries** method accesses the style information for the cell and then stores it into the **WSCellEntry** structure, to be written out along with the other descriptive data. As before, the **WSSize** field specifies the length of the entry string that follows the structure in the file.

ReadData Method Code

The **ReadData** method is the mirror image of the **WriteData** method. It must read the data in the same format in which the data were written, so it has been modified only slightly, to account for the style information that is now contained in the file. The code for the **ReadData** method is as follows:

```
voidCEnsembleData::ReadData(void)
{
   long  textLength, WSEntryCt;
   fontInfo  theFontInfo;

   TRY
   {
      FailOSErr (SetFPos( refNum, fsFromStart, 0L));
      ReadSome((Ptr)&textLength, sizeof(long));
      ReadSome((Ptr)&WSEntryCt, sizeof(long));

      if(textLength > 0)
      {
         ReadSome((Ptr)&theFontInfo, sizeof (fontInfo));
         ((CEnsembleDoc *) itsDocument)->theTextData
            ->SetFontData (theFontInfo);

         itsEditTextData = NewHandleCanFail(textLength);
         FailNIL(itsEditTextData);
         ReadSome(*itsEditTextData, textLength);
      }

      if(WSEntryCt > 0)
      {
         ReadStyles();
         ReadWSEntries(WSEntryCt);
      }
   }
   CATCH
   {
      ForgetHandle (itsEditTextData);
   }
   ENDTRY;
}
```

The only difference between this code and the equivalent code described in Chapter 8 is the inclusion of the call to **ReadStyles** in the section following the test for a nonzero **WSEntryCt** value.

ReadStyles Method Code

The **ReadStyles** method is the mirror image of the corresponding **WriteStyles** method, shown on page 292. The code is entirely new in this version of the Ensemble application:

```
void CEnsembleData::ReadStyles (void)
{
   cellInfo      cellStyle;
   short         cols, rows;
   short         index, rowCol;
   CCellData     *aStyle;

   ReadSome ((Ptr)&cols, sizeof(short));
   ReadSome ((Ptr)&rows, sizeof(short));
   for(index=0; index < cols; index++)
   {
      ReadSome((Ptr)&rowCol, sizeof(short));
      ReadSome((Ptr)&cellStyle, sizeof(cellInfo));
      aStyle = new CCellData;
      aStyle->ICellData(cellStyle);
      itsHList->InsertAt(aStyle, rowCol+1);
   }

   for(index=0; index < rows; index++)
   {
      ReadSome((Ptr)&rowCol, sizeof(short));
      ReadSome((Ptr)&cellStyle, sizeof(cellInfo));
      aStyle = new CCellData;
      aStyle->ICellData(cellStyle);
      itsVList->InsertAt(aStyle, rowCol+1);
   }
}
```

According to the file format shown in Figure 11-1, the **ReadStyles** method must first read in the **cols** and **rows** values, which determine how many column and row style entries follow. After these values are accessed from the file, two loops that input the column style entries, followed by the row style entries, complete the method.

To store the style entries, a new instance of class **CCellData** is created, and the instance is initialized with the style data read from the file. This is the reason that the **ICellData** method (page 287) takes a **cellInfo** argument. When the **CCellData** instance has been initialized, it is inserted into the corresponding list (**itsHList** or **itsVList**), as appropriate.

ReadWSEntries Method Code

The **ReadWSEntries** method is only changed slightly, as was the **WriteWSEntries**, to make provision for the inclusion of the **WSStyle** field in the **WSCellEntry** structure shown on page 294. The revised code for the **ReadWSEntries** method is as follows:

```
void  CEnsembleData::ReadWSEntries (long entryCount)
{
   WSCellEntry   anEntry;
   short         index;
   Str255        entryData;
   CWSEntry      *aWSEntry;

   for(index = 0; index < entryCount; index++)
   {
      ReadSome((Ptr)&anEntry, sizeof(WSCellEntry));
      ReadSome((Ptr)&entryData[1], (long) anEntry.WSSize);
      entryData[0] = anEntry.WSSize;

      TRY
      {
         aWSEntry = new CWSEntry;
         aWSEntry->IWSEntry ();
         aWSEntry->SetWSCell (anEntry.WSCell);
         aWSEntry->SetWSType (anEntry.WSType);
         aWSEntry->SetWSStyle (anEntry.WSStyle);
         aWSEntry->SetWSValue (0.0);
         aWSEntry->SetWSEntry (entryData);
         if(anEntry.WSType == 1)
         {
            aWSEntry->SetWSText(entryData); // string
         }
         else
         {
            aWSEntry->SetWSText("\p0.00");   // value
         }
         itsCluster->Add(aWSEntry);
      }
      CATCH
      {
         ForgetObject (aWSEntry);
      }
      ENDTRY;
   }
}
```

The **ReadWSEntries** method is nearly the same as the method shown on page 190, except that it uses a new access method, **SetWSStyle**, to store the style information into the

CWSEntry worksheet entry instance. After an entry has been constructed, it is added to the main worksheet cluster.

DisposeData Method Code

The **DisposeData** method has been enhanced in this version of the Ensemble application by adding code to dispose of the entries in the column and row lists, in addition to the text and worksheet data. The new code is as follows:

```
void  CEnsembleData::DisposeData(void)
{
   long  WSEntryCt, index;

   if (itsEditTextData != NULL)
   {
      DisposHandle (itsEditTextData);
      itsEditTextData = NULL;
   }
   if (itsCluster != NULL)
   {
      WSEntryCt = itsCluster->GetNumItems();
      for (index = 1; index <= WSEntryCt; index++)
      {
         itsCluster->DeleteItem (1);
      }
   }
   if (itsHList != NULL)
   {
      WSEntryCt = itsHList->GetNumItems();
      for (index = 1; index <= WSEntryCt; index++)
      {
         itsHList->DeleteItem (1);
      }
   }
   if (itsVList != NULL)
   {
      WSEntryCt = itsVList->GetNumItems();
      for (index = 1; index <= WSEntryCt; index++)
      {
         itsVList->DeleteItem (1);
      }
   }
}
```

GetHList and GetVList Methods

Two new access methods have been added to the **CEnsembleData** class to provide the means for other classes to access the column and row list instances. The code for the **GetHList** and **GetVList** methods is as follows:

```
CList *CEnsembleData::GetHList (void)
{
   return itsHList;
}

CList *CEnsembleData::GetVList (void)
{
   return itsVList;
}
```

As is apparent, all that these methods do is return the value of the corresponding instance variable, which contains a handle to the list instance.

Customizing the CWorksheet Code

The newly generated code to implement the **Worksheet** dialog was described in Chapter 10. This section describes the custom additions to the code in the **CWorksheet** subclass that implements the full functionality of the dialog.

The subclass is always the code that is modified when you implement the full functionality of a new user interface feature. In the case of the **Worksheet** dialog, although AppMaker generates the entire code to create the dialog and respond to the user's actions when a button or checkbox is clicked, we must add the code that makes each of these actions functional.

The code modifications in the **CWorksheet** class are fairly comprehensive. We need to take deliberate actions to show visual feedback when the user selects a font or size from the associated list, or when a font style or justification selection is made. In addition, when a **row** is selected for modification, the **column** information is irrelevant and should not be shown. Conversely, when a **column** is selected, the **row** information is of no value. When an individual **cell** is selected, neither the **row** nor **column** settings are shown. The methods that have been modified or added are listed in Table 11-2.

Table 11-2
Customized methods to implement the **Worksheet** dialog

Class	Method	Description
global	DoWorksheet	Main Worksheet dialog function
CWorksheet	IWorksheet	Initializes the CWorksheet
CWorksheet	DoCommand	Handles click commands
CWorksheet	ProviderChanged	Handles list events
CWorksheet	**DrawSample**	Draws sample text
CWorksheet	**CellToString**	Converts cell # to string
CWorksheet	**GetSettings**	Gets style data for selected row, column, or cell
CWorksheet	**ComparePStrings**	Compares two Pascal strings
CList24	IViewTemp	Initializes font list instance
CList24	GetCellText	Returns font name string
CList28	IViewTemp	Initializes font size list instance
CList28	GetCellText	Returns font size string

DoWorksheet Function Code

The **DoWorksheet** code is a global function that can be called by any method in the application. As you will see in a later section, the function is called by the **DoCommand** method in the **CCalcWindow** class. The existing call contained in the **ZEnsembleDoc** generated code is not disturbed (as the rule for never modifying the superclass code dictates), but is never executed because the menu command is first sent to the method in the **CCalcWindow** class, as is shown in Figure 10-1.

The **DoWorksheet** function is responsible for creating the **Worksheet** dialog, initializing its user interface elements, running the dialog so that the user can make different selections, and collecting the results after the user has dismissed the dialog by clicking the **OK** button. If the **Cancel** button is clicked, the previous settings must be left intact. The code for the **DoWorksheet** function is as follows:

DoWorksheet
function code
(beginning)

```
void  DoWorksheet(CDirectorOwner  *aSupervisor)
{
   CWorksheet   *dialog;
   long         dismisser, value;
   Str255       aString;
   short        aChoice;

   dialog = NULL;

   TRY
   {
      dialog = new CWorksheet;
      dialog->IWorksheet (aSupervisor);

      //
      // get the settings and initialize the dialog pane
      //
      dialog->GetSettings(cCellRadioViewID);

      //
      // "Cell" is initially selected,
      //  so disable the row and column fields.
      //
      dialog->HeightLabel->Hide();
      dialog->WidthLabel->Hide();
      dialog->heightField->Hide();
      dialog->widthField->Hide();

      //
      // now, show the dialog
      //
      dialog->BeginDialog ();
      //
      // start running the dialog event loop
      //
      dismisser = dialog->DoModalDialog (cmdOK);
      if (dismisser == cmdOK)
      {
         //
         // save the style and measurement values,
         // as well as the new prospective cell,
         // so that the caller can alter the
         // worksheet.
         //
         dialog->theStatus.modified = TRUE;
         dialog->theCellData->cellData = dialog->theInfo;
         ((CCalcWindow *)aSupervisor)->SetCellData(dialog->theCellData);
```

***DoWorksheet** function code (concluded)*

```
        ((CCalcWindow *)aSupervisor)->SetCellStatus(dialog->theStatus);
      }
      dialog->Dispose ();
   }
   CATCH
   {
      ForgetObject (dialog);
   }
    ENDTRY;
}
```

The code for the **DoWorksheet** function is divided into three sections. The first section creates and initializes the dialog, the second section "runs" the dialog, and the third section saves the settings so that the **DoCommand** method in the **CCalcWindow** class can make the necessary modifications to the worksheet's appearance.

The first section of the code begins as generated by AppMaker. The **CWorksheet** instance is created, and the **IWorksheet** method is called to initialize the instance. As the code in Chapter 10 shows (see page 278), the first action of the **IWorksheet** method is to call its inherited **IZWorksheet** method in the **ZWorksheet** superclass. This is also true of the revised code, though we will perform some additional initialization in the **IWorksheet** method, after the **Worksheet** dialog elements have been created. When the **IWorksheet** method returns, the **DoWorksheet** function calls a new method (**GetSettings**) to access the settings associated with the current cell, row, or column selection. Because the **Cell** radio button is initially selected, the **DoWorksheet** function calls the **GetSettings** method with an identifier of **cCellRadioViewID**, which is the resource ID of the **Cell** radio button, as defined in our code. After the settings for the current cell have been accessed and placed into the appropriate user interface elements, the row height and column width fields are hidden by calling the TCL to hide them. When this is done, the **BeginDialog** method is called to show the dialog. This is the end of the first section of the function.

The second section merely calls the **DoModalDialog** method. The TCL takes care of interacting with the user, accepting the key and mouse events, and sending messages to the appropriate **DoCommand** or **ProviderChanged** methods in the **CWorksheet** class when an event affects one of the user interface elements. Although taking part in the execution of the second section, the **DoCommand** and **ProviderChanged** methods will be discussed later. When the user selects a font name, a message is sent to the **ProviderChanged** method to handle the selection. Similarly, when a font style or justification selection is made, the **DoCommand** method is called. The second section of the **DoWorksheet** function terminates when the user dismisses the dialog, by clicking either the **OK** or **Cancel** button.

The third section of the **DoWorksheet** function will execute only if the **OK** button was clicked. This section is responsible for saving the user's selections. When the dialog is dismissed by clicking on the **OK** button, the function uses two access methods in the **CCalcWindow** class (**SetCellData** and **SetCellStatus**) to save the style and status data associated with the user's actions. (These access methods will be described later in the chapter.) After the style data and status have been saved, if the user cancels the dialog, the dialog is disposed of, which also disposes all of its user interface elements.

The **DoWorksheet** function is also organized into TRY and CATCH blocks, which provide a failure recovery mechanism in case an error occurs when the dialog is being created or operated. As with other THINK C code, if the CATCH block is executed, the failure will propagate to the TCL, which will display a dialog to the user, indicating the nature of the failure.

IWorksheet Method Code

As previously indicated, the **IWorksheet** method is called by the **DoWorksheet** function after the **CWorksheet** instance is created. The purpose of the **IWorksheet** method is to create the **Worksheet** dialog and instantiate all of its user interface elements. In addition, any special initialization not included in the generated code is added to the method. The code for **IWorksheet** is as follows:

***IWorksheet** method code (beginning)*

```
void  CWorksheet::IWorksheet(CDirectorOwner*aSupervisor)
{
   inherited::IZWorksheet (aSupervisor);

   // any additional initialization for your dialog
   CenterRadio->ID = cCenterRadioViewID;
   RightRadio->ID = cRightRadioViewID;
```

***IWorksheet* method code (concluded)**

```
    ForceLeftRadio->ID = cForceLeftRadioViewID;
    LeftRadio->ID = cLeftRadioViewID;

    RowRadio->ID = cRowRadioViewID;
    CellRadio->ID = cCellRadioViewID;
    ColumnRadio->ID = cColumnRadioViewID;

    fontField->SetTextString ("\pSystem");
    sizeField->SetTextString ("\p12);
}
```

The generated code calls the inherited **IZWorksheet** method to create the dialog and its elements. This code is generated into the **ZWorksheet** superclass and is not modified. The custom initialization code added to this method consists of assigning "view IDs" to the radio buttons in the dialog. These IDs are arbitrary, but are stored in an instance variable associated with the view, providing a method of identifying a particular element to use for enabling a specific button in a group. We have added definitions for these buttons in our **Worksheet.h** header file to allow references to the IDs to be symbolic. The definitions are as follows:

Worksheet dialog radio button viewID definitions

```
enum
{
    cCenterRadioViewID= 157,
    cRightRadioViewID,
    cForceLeftRadioViewID,
    cLeftRadioViewID
};

enum
{
    cRowRadioViewID= 171,
    cCellRadioViewID,
    cColumnRadioViewID
};
```

Although these definitions are arbitrary (as far as the TCL is concerned), we have chosen to use the **'CtlP'** resource ID numbers for them, as shown in the **IZWorksheet** method code beginning on page 269. In addition to initializing the view IDs, the code writes an initial valid font name and size into the corresponding EditText fields.

DoCommand Method Code

Each of the checkbox and radio button elements in the dialog is assigned an associated click command in the associated resource template created by AppMaker. When the element is created and its **IViewTemp** method is executed, its click command is sent to the TCL. Whenever the user clicks the mouse in one of these controls, the appropriate click command is sent to the **DoCommand** method for the element's supervisor, which, in this case, is the **CWorksheet** class.

The **DoCommand** method is passed a long integer that specifies the command that is to be handled. This could just as easily be a menu command as a click command; when it is received by the **DoCommand** method, there is no difference at that point. If it is ever necessary to install a menu when a dialog is invoked, selection of a menu command will also generate a message to the dialog's **DoCommand** method. In our case, only the checkbox and radio buttons have associated commands. The code for the **DoCommand** method is quite lengthy:

DoCommand *method code (beginning)*

```
void  CWorksheet::DoCommand (long theCommand)
{
   short style = -100;
   short align = -100;

   switch (theCommand)
   {
      case cmdBoldCheck:
      {
         style = bold;
         break;
      }
      case cmdItalicCheck:
      {
         style = italic;
         break;
      }
      case cmdUnderlineCheck:
      {
         style = underline;
         break;
      }
      case cmdOutlineCheck:
      {
```

DoCommand
method code
(continued)

```
        style = outline;
        break;
    }
    case cmdShadowCheck:
    {
        style = shadow;
        break;
    }
    case cmdCondenseCheck:
    {
        style = condense;
        break;
    }
    case cmdExtendCheck:
    {
        style = extend;
        break;
    }
    case cmdCenterRadio:
    {
        align = teCenter;
        break;
    }
    case cmdRightRadio:
    {
        align = teFlushRight;
        break;
    }
    case cmdForceLeftRadio:
    {
        align = teFlushDefault;
        break;
    }
    case cmdLeftRadio:
    {
        align = teFlushLeft;
        break;
    }
    case cmdRowRadio:
    {
        GetSettings(cRowRadioViewID);
        WidthLabel->Hide();
        widthField->Hide();
        HeightLabel->Show();
        heightField->Show();
        theStatus.rowColCell = 0; // row
        break;
    }
    case cmdCellRadio:
```

DoCommand* *method code (continued)

```
    {
        GetSettings(cCellRadioViewID);
        HeightLabel->Hide();
        WidthLabel->Hide();
        heightField->Hide();
        widthField->Hide();
        theStatus.rowColCell = 2; // cell
        break;
    }
    case cmdColumnRadio:
    {
        GetSettings(cColumnRadioViewID);
        HeightLabel->Hide();
        heightField->Hide();
        WidthLabel->Show();
        widthField->Show();
        theStatus.rowColCell = 1;
        break;
    }
    case cmdDollars99999999Check:
    {
        theInfo.dollars = Dollars99999999Check->GetValue();
        break;
    }
    case cmdCommas99999999Check:
    {
        theInfo.commas  = Commas99999999Check->GetValue();
        break;
    }
    case cmdChangeTextStyleCheck:
    {
        theStatus.changeStyle = ChangeTextStyleCheck->GetValue();
        break;
    }
    default:
    {
        inherited::DoCommand (theCommand);
        break;
    }
}
if(style != -100)
{
    sampleField->SetFontStyle(style);
    theInfo.fontStyle ^= style;
    DrawSample();
}
if(align != -100)
{
    sampleField->SetAlignment(align);
```

***DoCommand** method code (concluded)*

```
            theInfo.fontAlign = align;
            DrawSample();
    }
}
```

The first portion of the **DoCommand** method mimics the behavior of the corresponding method in the **CNotebook** class, as shown beginning on page 119. Initial values are assigned to the **style** and **align** variables, and if one of the buttons associated with the style or justification control was clicked, the associated variable will be updated with a new value. When the method reaches its end, the **style** and **align** variables are tested to determine whether they hold something other than the default values. If so, the appropriate style or alignment changes are made.

Following the sections of code that handle style or alignment events is a section that handles events when the **Row**, **Column**, or **Cell** button is clicked. The function of this code is to get the style settings for the chosen row, column, or cell and modify the dialog's displayed values to correspond with these settings. The **GetSettings** method is used to perform the appropriate changes. Incidentally, if the **Row** button is selected, then the column width is hidden, if the **Column** button is selected, the row height is hidden, and if the **Cell** button is selected, both the row height and column width are hidden.

Finally, before the code that changes the text style or justification is executed are sections of code that handle clicks on the **Dollars**, **Commas**, and **Change Text Style** checkboxes. These "cases" in the method merely save the current value of the associated control, so that its status can affect the worksheet text in an appropriate manner after the dialog has been dismissed.

ProviderChanged Method Code

The **ProviderChanged** method is invoked whenever one of the **Worksheet** dialog's lists or EditText fields has been changed. The classes that manipulate those elements in the TCL send **BroadcastChange** messages up the hierarchy, these messages are intercepted by the **CBureaucrat** class, and the **ProviderChanged** method is called for the "owner" of the element (i.e., the **CWorksheet** class).

The default-generated code for the **ProviderChanged** method was shown beginning on page 281. This code provided a framework for the custom code that we have added, to process the messages sent by the list and text field elements. In customizing this method, we have changed the code to eliminate both outcomes of the conditional tests; however, the code is essentially the same:

ProviderChanged *method code (beginning)*

```
void  CWorksheet::ProviderChanged(CCollaborator   *aProvider,
                                                long     reason,
                                                void*    info)
{
   short    index;
   Str255   theText;
   long     value;
   Cell     aCell;

   if (aProvider == fontField)
   {
      if (fontField->GetLength () != 0)
      {
         DrawSample();
      }
   }
   if (aProvider == sizeField)
   {
      if (sizeField->GetLength () != 0)
      {
         DrawSample();
      }
   }
   if (aProvider == List24) {
      if (List24->HasSelection ())
      {
         // store selection in EditText field
         if(List24->GetChoice(&index))
         {
            GetItem(((CList24 *)List24)->fontMenu, index+1, theText);
            fontField->SetTextString(theText);
            DrawSample();
         }
      }
   }
   if (aProvider == List28)
   {
      if (List28->HasSelection())
      {
```

***ProviderChanged** method code (concluded)*

```
            // store selection in EditText field
            if(List28->GetChoice(&index))
            {
                index = (index << 1) +1;
                theText[0] = 2;
                theText[1] = ((CList28 *) List28)->typeSizes[index++];
                theText[2] = ((CList28 *) List28)->typeSizes[index++];
                sizeField->SetTextString(theText);
                DrawSample();
            }
        }
    }
    if (aProvider == heightField)
    {
        if (heightField->GetLength () != 0)
        {
            heightField->GetTextString(theText);
            StringToNum(theText, &value);
            theStatus.cellHeight = value;
        }
    }
    if (aProvider == widthField)
    {
        if (widthField->GetLength () != 0)
        {
            widthField->GetTextString(theText);
            StringToNum(theText, &value);
            theStatus.cellWidth = value;
        }
    }
    if (aProvider == digitsField)
    {
        if (digitsField->GetLength () != 0)
        {
            digitsField->GetTextString(theText);
            StringToNum(theText, &value);
            theInfo.decimalDigits = value;
        }
    }
}
```

The **ProviderChanged** method deals only with events that occur with regard to the EditText and list user interface elements. For example, when a different font is selected from the font list, the method copies its name into the EditText field below the list and also causes the **Sample** field to be redrawn using the new font. A new size selection causes a similar action to be taken. If the contents of the **heightField**, **widthField**, or **digitsField** are changed by the user, then the associated value is saved into the appropriate structure, for subsequent access by the **DoCommand** method in the **CCalcWindow** class (from where the **Worksheet** dialog was invoked).

DrawSample Method Code

Several of the sections of code in the **ProviderChanged** method make use of a method called **DrawSample**, whose job it is to draw the sample text in the specified font, size, style, and justification. The code for **DrawSample** is as follows:

DrawSample *method code (beginning)*

```
void  CWorksheet::DrawSample(void)
{
   short       fontNum;
   long        fontSize, strLength;
   Str255      theFontText, theSizeText, theSampleText;
   strLength = fontField->GetLength();
   if(strLength > 0)
   {
      fontField->GetTextString(theFontText);
      if(EqualString(theFontText, "\pSystem"))
      {
         fontNum = systemFont;
      }
      else if(EqualString(theFontText, "\pApplication"))
      {
         fontNum = applFont;
      }
      else
      {
         GetFNum(theFontText, &fontNum);
      }
   }
   else
   {
      fontNum = systemFont;
   }
   strLength = sizeField->GetLength();
   if(strLength > 0)
   {
      sizeField->GetTextString(theSizeText);
      StringToNum(theSizeText, &fontSize);
   }
   else
```

***DrawSample* method code (concluded)**

```
        fontSize = 12;
    CopyPString("\pSample", theSampleText);
    sampleField->SetTextString(theSampleText);
    sampleField->SetFontNumber(fontNum);
    sampleField->SetFontSize(fontSize);
    theInfo.fontNumber = fontNum;
    theInfo.fontSize = fontSize;
}
```

The purpose of the **DrawSample** method has already been explained. The method takes a few steps to avoid trying to draw in a font that doesn't exist, but will use any font size you specify. If the font is too large or too small, the sample text will simply not be readable. There's no harm in this practice: It may be important for the user to key in a font size that isn't supported in the list of sizes. In this case, the method dutifully uses the size specified by the user. If the font isn't the **System** or **Application** font, then the toolbox **GetFNum** routine is used to return the font number of a specified font. If the font does not exist, **GetFNum** will return 0, the **System** font number.

CellToString Method Code

The **CellToString** method is used to convert a binary cell value to its string equivalent, for display in the **Worksheet** dialog box. The code is "hard wired" to the notion that there is a maximum of 26 columns in our worksheet; however, this could be modified if the worksheet size is expanded. The code for **CellToString** is as follows:

```
void  CWorksheet::CellToString (Cell aCell, StringPtr aString)
{
    Str15  Col, Row;
    Col[0] = 1;
    Col[1] = (aCell.h + 'A');
    NumToString (aCell.v + 1, Row);
    CopyPString(Col, aString);
    ConcatPStrings(aString, Row);
}
```

The preceding code merely adds the character 'A' to the column value and then converts the row+1 value to a string and concatenates the two.

GetSettings Method Code

The **GetSettings** method is a major addition to the **Worksheet** module. The method is responsible for changing the settings when the user chooses to modify the **Row**, **Column**, or **Cell** style. Because the existing font, size, style, alignment, and options can be totally different for each of these choices, the **GetSettings** method must maintain the current values of each of these possibilities. This provides the user with instant feedback on the current settings for each choice. The worksheet cell objects contain these settings for individual cells, and the corresponding row and column lists maintain the settings for the row and column corresponding to the selected cell. The function of the **GetSettings** method is to use the incoming parameter (which specifies one of the three radio buttons selecting a **Row**, **Column**, or **Cell**) and reset the dialog's parameters to match the corresponding settings for the selected choice. The code to implement this feature is as follows:

GetSettings *method code (beginning)*

```
void  CWorksheet::GetSettings (short viewID)
{
   short    sizeSelect;
   short    fontIndex;
   short    sizeIndex;
   short    radioID;
   Str255   aString;
   Str255   listName;

   switch (viewID)
   {
      case cCellRadioViewID:
      {
         theCellData = ((CCalcWindow *)itsSupervisor)->GetCellData();
         break;
      }
      case cColumnRadioViewID:
      {
         theCellData = ((CCalcWindow *)itsSupervisor)->GetColData();
         break;
      }
      case cRowRadioViewID:
      {
         theCellData = ((CCalcWindow *)itsSupervisor)->GetRowData();
         break;
      }
   }
```

GetSettings *method code (continued)*

```
Group35->SetStationID(viewID);
theInfo = theCellData->cellData;
theStatus = ((CCalcWindow *)itsSupervisor)->GetCellStatus();
CellToString(theStatus.itsCell, aString);
cellNumLabel->SetTextString(aString);
NumToString(theStatus.cellHeight, aString);
heightField->SetTextString(aString);
NumToString(theStatus.cellWidth, aString);
widthField->SetTextString(aString);
NumToString(theInfo.decimalDigits, aString);
digitsField->SetTextString(aString);
Dollars99999999Check->SetValue(theInfo.dollars);
Commas99999999Check->SetValue(theInfo.commas);

GetFontName (theInfo.fontNumber, aString);
if (aString[0] == 0)
{
   CopyPString ("\pSystem", aString);
}
for (fontIndex = 1; fontIndex <= ((CList24 *)List24)->numFonts;
   fontIndex++)
{
   GetItem(((CList24 *)List24)->fontMenu, fontIndex, listName);
   if (EqualString (listName, aString))
   {
      fontIndex--;
      break;
   }
}
if (fontIndex > ((CList24 *)List24)->numFonts)
{
   fontIndex = 0;
}
fontField->SetTextString (aString);

NumToString (theInfo.fontSize, aString);
if (aString[0] == 1)
{
   aString[2] = aString[1];
   aString[1] = ' ';
   aString[0] = 2;
}

for (sizeIndex = 1; sizeIndex <= 24; sizeIndex+=2)
{
   if (((CList28 *)List28)->typeSizes[sizeIndex+0] == aString[1]
   &&  ((CList28 *)List28)->typeSizes[sizeIndex+1] == aString[2])
   {
      sizeIndex >>= 1;
```

GetSettings *method code (concluded)*

```
            break;
        }
    }
    if (sizeIndex > 12)
    {
        sizeIndex = 3;
        CopyPString ("\p12", aString);
    }
    sizeField->SetTextString (aString);
    List24->SetChoice (fontIndex);
    List28->SetChoice (sizeIndex);

    BoldCheck->SetValue(theInfo.fontStyle & bold);
    ItalicCheck->SetValue(theInfo.fontStyle & italic);
    UnderlineCheck->SetValue(theInfo.fontStyle & underline);
    OutlineCheck->SetValue(theInfo.fontStyle & outline);
    ShadowCheck->SetValue(theInfo.fontStyle & shadow);
    CondenseCheck->SetValue(theInfo.fontStyle & condense);
    ExtendCheck->SetValue(theInfo.fontStyle & extend);

    if(theInfo.fontAlign == teFlushLeft)
    {
        radioID = cLeftRadioViewID;
    }
    else if(theInfo.fontAlign == teCenter)
    {
        radioID = cCenterRadioViewID;
    }
    else if(theInfo.fontAlign == teFlushRight)
    {
        radioID = cRightRadioViewID
    }
    else
        radioID = cForceLeftRadioViewID;
    Group16->SetStationID(radioID);
    sampleField->SetTextString("\pSample");
    sampleField->SetFontNumber(theInfo.fontNumber);
    sampleField->SetFontSize(theInfo.fontSize);
    sampleField->SetFontStyle(theInfo.fontStyle);
    sampleField->SetAlignment(theInfo.fontAlign);
    ChangeTextStyleCheck->SetValue(0);
}
```

As is apparent, the code for **GetSettings** is quite lengthy. The method begins by determining which **viewID** was selected and then calls the appropriate access method in the **CCalcWindow** class to acquire either the row, column, or cell style data. Once armed with the data, it reconstructs the dialog's fields, controls, lists, and selections to correspond with the previously specified values.

Because only the simple binary values of the font, size, style, and alignment, along with the decimal digits, commas, and dollars settings are stored, the method must create the appropriate settings from these basic data. The process, though somewhat laborious, saves us from having to store a great deal of data for each defined worksheet cell, not to mention the corresponding settings for each row and column in the worksheet.

It is believed that the trade-off of some extra code for a substantial savings in worksheet storage is justified. In addition, the file size is minimized when the data are saved to disk.

CList24 IViewTemp and GetCellText Methods

The **CList24** class implements the font name list. There are two methods generated in this class. The initialization method, called **IViewTemp**, specifies the size of the list, based on the number of fonts in the user's open resource files. The **GetCellText** method returns the contents of a font name string to the TCL's **CTable** class when the entry must be redrawn. The code for the **IViewTemp** and **GetCellText** methods is as follows:

```
void  CList24::IViewTemp(CView        *anEnclosure,
                         CBureaucrat  *aSupervisor, Ptr  viewData)
{
   inherited::IViewTemp (anEnclosure, aSupervisor, viewData);
   // get the font menu handle and initialize the table
   fontMenu = GetMHandle(FontID);
   numFonts = CountMItems(fontMenu);
   AddRow (numFonts, 0);
}

voidCList24::GetCellText(Cell       aCell,
                         short      availableWidth, StringPtritsText)
{
   short index;
   // get font names from "fontMenu" and insert in list
   index = aCell.v;
   GetItem(fontMenu, index+1, itsText);
}
```

The **IViewTemp** and **GetCellText** methods depend upon the **fontMenu** that was built in the **SetUpMenus** method of the **CEnsembleApp** class, shown on page 92. The **FontID** resource number used to retrieve the menu is defined in the **ResourceDefs.h** header file.

The **IViewTemp** method sets up the list to contain the number of rows corresponding to the number of available fonts, and the **GetCellText** method, when called with a cell number, uses the vertical component (the row) to determine which name string to return in the **itsText** argument.

CList28 IViewTemp and GetCellText Methods

As with the **CList24** class, the **CList28** class implements a list, which in this case is the font size list. The **IViewTemp** method creates a list with 12 rows, corresponding to the 12 fixed font sizes that we have defined. The **GetCellText** method returns the string value of the selected size when called by the TCL's **CTable** class. The code for the **CList28** class's **IViewTemp** and **GetCellText** methods is as follows:

```
void  CList28::IViewTemp(CView          *anEnclosure,
                          CBureaucrat   *aSupervisor,
                          Ptr           viewData)
{
   inherited::IViewTemp (anEnclosure, aSupervisor, viewData);
   CopyPString("\p 8 9101214161820242832 36", typeSizes);
   AddRow (12, 0);
}

void  CList28::GetCellText(Cell         aCell,
                            short       availableWidth,
                            StringPtr   itsText)
{
   short     strIndex = (aCell.v << 1) + 1;
   *itsText++ = 2;
   *itsText++ = typeSizes[strIndex++];
   *itsText  = typeSizes[strIndex];
}
```

Customizing the CCalcWindow Code

The **CCalcWindow** module provides support for all of the worksheet-oriented operations. Because support for row, column, and cell styles has been added, several of the classes and their methods have been further customized, based on the operational worksheet model presented in Chapter 8. Many of the classes and methods described in that chapter remain unchanged. This section covers only the classes and methods that have been modified to support styled worksheet cells.

Customizing the Lists

As indicated in Chapter 8, the worksheet uses three visible lists that contain the column labels, row labels, and main body of the worksheet. The modified and new methods are listed in Table 11-2. The boldfaced names specify newly created methods.

Table 11-3
List class custom code modifications

Class	Method	Description
CList10	DrawCell	Draws row label cell
CList15	GetCellText	Returns main worksheet cell text
CList15	**DrawCell**	Draws worksheet cell contents
CList15	**GetCellStyle**	Accesses cell style information
CList15	**DrawWSCell**	Draws styled text cell entry
CList15	**SetStyleLists**	Saves instance handle for row and column style lists

The table indicates that only two of the existing list methods were modified, and that four additional methods have been added. This speaks well for the modularity of the object-oriented design of the Ensemble application. When new functionality is provided, it is always a good sign if the existing code doesn't have to be scrapped to make way for features of the new implementation.

CList10 DrawCell Method Code

The first list method to be changed is the **DrawCell** method for the **CList10** instance. This list holds the row labels. Because it wasn't anticipated at the outset that the user would need to change the row height, the row label was being written at a standard vertical offset in the cell. Now that the **Worksheet** dialog offers the ability to change the height of any row, the **DrawCell** method for the row labels must center the row label vertically in the row. The modified code is as follows:

```
void  CList10::DrawCell (Cell theCell, Rect *cellRect)
{
   Str255   cellText;
   short    availWidth, textWidth, availHeight;

   availWidth = cellRect->right - cellRect->left;
   availHeight = cellRect->bottom - cellRect->top;
   GetCellText(theCell, availWidth, cellText);
   textWidth = StringWidth(cellText);
   indent.h = availWidth - textWidth - vertLabMargin;
   indent.v = ((availHeight - 12) >> 1) + 12;
   if (cellText[0] > 0)
   {
      MoveTo( cellRect->left + indent.h, cellRect->top + indent.v);
      DrawString( cellText);
   }
}
```

The modified **DrawCell** code takes advantage of the available cell height to center the row label vertically. If the cell is too short for the label to fit, its baseline is set 12 pixels from the top of the cell. It is possible that the label will not entirely show. The label font and size are not adjustable, and the label's height above the baseline is a constant 12 pixels.

CList15 GetCellText Method Code

The **CList15** class implements the main worksheet list. It contains methods to manipulate the individual cells of the worksheet. The **GetCellText** method has been modified to provide for a variable number of decimals and for commas in numeric values. The modified code is as follows:

CList15 GetCellText *method code (beginning)*

```
void  CList15::GetCellText(Cell        aCell,
                           short       availableWidth,
                           StringPtr   itsText)
{
    double      itsValue, newValue;
    short       itsType, index, num, dec;
    long        aParam;
    Str255      itsEntry, itsCellText;
    decform     aFormat;
    extended    temp;
    CWSEntry    *anObj;
    cellInfo    cellStyle;

    if((CWSEntry *)itsCluster == NULL)
    {
        CopyPString("\p", itsText);
        return;
    }
    aParam = *(long*) &aCell;
    anObj = (CWSEntry *)itsCluster->FindItem1 (FindWSCell, aParam);
    if(anObj)
    {
        if((itsType = anObj->GetWSType()) == 2)
        {
            index = 1;
            anObj->GetWSEntry(itsEntry);
            newValue = ((CCalcWindow *)itsSupervisor)
                ->GetExpression (itsEntry, &index, 0);
            itsValue = anObj->GetWSValue();
            cellStyle = anObj->GetWSStyle();
            GetCellStyle (aCell, &cellStyle);
            aFormat.style  = FIXEDDECIMAL;
            aFormat.digits = cellStyle.decimalDigits;
            x96tox80(&newValue, &temp);
            num2str(&aFormat, temp, itsCellText);
            if(cellStyle.commas)
            {
                num = itsCellText[0];
                dec = cellStyle.decimalDigits;
                dec = (dec > 0) ? dec : -1;
                index = num - dec - 3;
                while (index > 1)
                {
                    BlockMove(&itsCellText[index], itsEntry, num-index+1);
                    itsCellText[index] = ',';
                    BlockMove(itsEntry, &itsCellText[index+1], num-index+1);
                    index -= 3;
```

***CList15 GetCellText* method code (concluded)**

```
                num++;
            }
            itsCellText[0] = num;
        }
        if(cellStyle.dollars)
        {
            CopyPString ("\p$ ", itsEntry);
            ConcatPStrings (itsEntry, itsCellText);
            CopyPString (itsEntry, itsCellText);
        }
        anObj->SetWSText(itsCellText);
        anObj->SetWSValue(newValue);
    }
    anObj->GetWSText(itsCellText);
    CopyPString(itsCellText, itsText);
  }
  else
  {
    CopyPString("\p", itsText);
  }
}
```

The modifications shown to the **GetCellText** method are concerned with the number of decimal digits in formatting numeric values and the inclusion of commas at the appropriate places if indicated. The method produces a string that represents the value to be drawn for the cell. The string is formatted and returned to the **DrawCell** method, which is located in the **CTable** class, but which we have overridden in this version of the application to provide for styled text in any cell. The override version of the **DrawCell** method is presented next.

CList15 DrawCell Method Code

The **CTable** class in the TCL contains a generic **DrawCell** method, which we have been using up to this point. It provides for writing the string returned by the **GetCellText** method in a standard font, size, and style. Our new override of this method provides complete control over the appearance of a cell. In this regard, a cell's font, size, style, and justification can be modified via the **Worksheet** dialog. The code for the new method is as follows:

```
void  CList15::DrawCell (Cell theCell, Rect *cellRect)
{
   CWSEntry      *anEntry;
   cellInfo       cellStyle;
   long           aParam;

   aParam = *(long *) &theCell;
   if((CWSEntry *) itsCluster)
   {
      if((anEntry = (CWSEntry *)itsCluster->FindItem1 (FindWSCell,
         aParam)) == NULL)
      {
         return;
      }
      cellStyle = anEntry->GetWSStyle();
      GetCellStyle (theCell, &cellStyle);
      DrawWSCell (theCell, cellRect, cellStyle);
   }
}
```

The code attempts to optimize the drawing of cells by handling only the first portion of the task. It determines whether the cell has been defined, and if not, it simply returns, drawing nothing in the process. If the cell *does* exist, the **DrawCell** method accesses the style data for the cell and then calls the **GetCellStyle** method to determine whether the individual cell style or that of the corresponding row or column should be used. It then calls the **DrawWSCell** method to perform the task of setting the style attributes and to draw the cell's contents.

CList15 GetCellStyle Method Code

The **GetCellStyle** method is responsible for determining whether the input cell style information should be used for drawing a given cell or wither the cell's corresponding column or row style data should be used. The code for this method is as follows:

CList15 GetCellStyle *method code (beginning)*

```
voidCList15::GetCellStyle (Cell theCell, cellInfo *cellStyle)
{
   CCellData*aColStyle, *aRowStyle;

   if(cellStyle->isDefault)
   {
      aColStyle = (CCellData *)itsHStyle->NthItem((long)theCell.h+1);
```

CList15 GetCellStyle *method code (concluded)*

```
        if(aColStyle->cellData.isDefault)
        {
            aRowStyle = (CCellData *)itsVStyle->NthItem((long)theCell.v+1);
            if(aRowStyle->cellData.isDefault)
            {
                return;
            }
            else
            {
                *cellStyle = aRowStyle->cellData;
            }
        }
        else
        {
            *cellStyle = aColStyle->cellData;
        }
    }
}
```

The **GetCellStyle** method immediately returns if the cell format is not the default value (i.e., if it has been changed with the **Worksheet** dialog). If the individual cell still reflects its default format, then the method checks whether the column style reflects its default style. If not, then the column style is used. If the column style still contains the default settings, then the row style is checked. If this still contains its default settings, then the method simply returns, using the cell's default style; otherwise, the row style is used. This method reflects a priority that has been established for all worksheet styles: The individual cell style takes precedence, followed by the column style, and, finally, by the row style. The method is called by both the **GetCellText** and the **DrawCell** methods.

CList15 DrawWSCell Method Code

The code that performs the final font, size, style, and justification settings and draws the cell text is encapsulated in this method. The code is as follows:

CList15 DrawWSCell *method code (beginning)*

```
void  CList15::DrawWSCell (Cell theCell, Rect *cellRect,
                                        cellInfo cellStyle)
{
    short       curFont;
    short       curSize;
    Style       curStyle;
    Str255      cellText;
```

CList15 DrawWSCell *method code (continued)*

```
FontInfo     fInfo;
short        availWidth, availHeight;
short        textWidth, textHeight;
short        hIndent, vIndent;

availWidth  = cellRect->right - cellRect->left;
availHeight = cellRect->bottom - cellRect->top;
GetCellText( theCell, availWidth, cellText);
if (cellText[0] > 0)
{
   curFont  = macPort->txFont;
   curSize  = macPort->txSize;
   curStyle = macPort->txFace;

   TextFont (cellStyle.fontNumber);
   TextSize (cellStyle.fontSize);
   TextFace (cellStyle.fontStyle);

   GetFontInfo (&fInfo);
   textWidth  = StringWidth(cellText);
   textHeight = fInfo.ascent + fInfo.descent;
   switch (cellStyle.fontAlign)
   {
      case teCenter:
      {

         indent.h = (availWidth - textWidth) >> 1;
         break;
      }
      case teFlushRight:
      {
         indent.h = availWidth - textWidth;
         break;
      }
      case teFlushLeft:
      {
         indent.h = 0;
         break;
      }
      case teFlushDefault:
      {
         indent.h = 0;
         break;
      }
      default:
      {
         indent.h = 0;
         break;
      }
```

***CList15 DrawWSCell** method code (concluded)*

```
            }
            if (textHeight > availHeight)
            {
                indent.v = availHeight - 2;
            }
            else
            {
                indent.v = ((availHeight - textHeight) >> 1) + fInfo.ascent;
            }

            MoveTo( cellRect->left + indent.h, cellRect->top + indent.v);
            DrawString( cellText);

            TextFont (curFont);
            TextSize (curSize);
            TextFace (curStyle);
        }
    }
```

The **DrawWSCell** method begins by computing the cell's available width and height and then calls the **GetCellText** method to get the string it must draw. The method is also called with a **cellStyle** argument that specifies the font, size, style, and justification settings for the cell text to be written. If the length of the cell text is greater than 0, the method continues; otherwise, it simply returns, drawing nothing.

The current settings for the macPort's **txFont**, **txSize**, and **txFace** are saved, and then the new values for this cell are installed by calling the toolbox **TextFont**, **TextSize**, and **TextFace** routines, with the **cellStyle** settings.

The **GetFontInfo** toolbox routine is called to get the metric settings for the new font. We are primarily interested in the ascent (distance above the baseline of the tallest character) and the descent (distance below the baseline for the character with the lowest descender) measurements. Following this, we also get the width of the character string that we intend to draw, using the toolbox **StringWidth** routine and placing this measurement into our **textWidth** variable. The **textHeight** is computed as the sum of the ascent and descent measurements of the font.

Armed with this information, we can commence the task of computing the horizontal position, **indent.h**, of the first character of the text. Different methods are used, depending on whether the selected setting is for **left-justified**, **centered**, or **right-justified** text. Following this, the vertical offset, **indent.v**, from the top of the cell is computed to center the text vertically within the cell.The final section of code moves the pen to the appropriate horizontal and vertical positions within the cell and then draws the string. When the operation is complete, the previous font, size, and style (face) are restored from the saved values.

CList15 SetStyleLists Method Code

This method is called by the **ICalcWindow** method, described later in the chapter. The entire purpose of the **SetStyleLists** method is to pass the handles to the row and column style lists from the main **CCalcWindow** module to the **CList15** class, where they need to be accessed. The code for the method is as follows:

```
void  CList15::SetStyleLists(CList *aHList, CList *aVList)
{
   itsHStyle = aHList;
   itsVStyle = aVList;
}
```

Customizing the CCalcWindow Methods

The **CCalcWindow** class implements the main behavior of the worksheet. It contains all the algorithms for accepting new entries, making changes to existing entries, and invoking the **Worksheet** dialog when that command is chosen.The new and modified methods for the class are shown in Table 11-2.

As is our custom, the entirely new methods are shown in boldface type, while existing methods are in plain type. It is important to point out that in no case was an existing method entirely scrapped and completely rewritten. In most cases, the modifications to existing methods are rather slight, and the entirely new methods consist of one or two lines of code, to provide access to the new style data.

Table 11-4
CCalcWindow custom code modifications

Class	Method	Description
CCalcWindow	ICalcWindow	Initializes worksheet window
CCalcWindow	MakeStringObj	Creates string object
CCalcWindow	MakeValueObj	Creates value object
CCalcWindow	UpdateMenus	Enables/disables menu commands
CCalcWindow	DoCommand	Handles menu commands
CCalcWindow	**GetCellData** **SetCellData**	Gets and sets cell style data-access methods
CCalcWindow	**GetCellStatus** **SetCellStatus**	Gets and sets cell status data-access methods
CCalcWindow	**GetColData** **GetRowData**	Gets column and row style data-access methods
CCalcWindow	**InitCellStyle**	Initializes new cell style

ICalcWindow Method Code

The **ICalcWindow** method has been modified to create default entries in the row and column style lists if these lists are empty. If the window is opened as a result of opening a file and reading prewritten worksheet data, then the lists are not disturbed. The code for the **ICalcWindow** method is as follows:

ICalcWindow *method code (beginning)*

```
void  CCalcWindow::ICalcWindow(CDirector        *aSupervisor,
                               CEnsembleData *theData)
{
   Rect         aRect;
   Str255       theFilename;
   long         index;
   CCellData    *aStyle;
   cellInfo     cellStyle;

   itsData = theData;
   inherited::IZCalcWindow (aSupervisor);
   gDecorator->StaggerWindow (itsWindow);

   //
   // put the file name into the CalcWindow's title
   // and set the min and max window sizes
   //
```

ICalcWindow
method code
(continued)

```
SetRect(&aRect, minWinHSize, minWinVSize, maxWinHSize,
    maxWinVSize);
if(((CEnsembleDoc *) aSupervisor)->itsFile != NULL)
{
    ((CEnsembleDoc *) aSupervisor)->itsFile->GetName(theFilename);
    itsWindow->SetTitle(theFilename);
}
itsWindow->SetSizeRect(&aRect);
itsWindow->ChangeSize(minWinHSize, minWinVSize);
((CAMEditText *)EntryField)->SetTextString("\p");

//
// turn on TEAutoView to enable scrolling the Entry field
//
TEAutoView (TRUE, ((CAMEditText *) EntryField)->macTE);

//
// send handles to the various lists to the "main worksheet"
// table (List15), so that it can access their contents
//
((CList15 *)List15)->SetLists (List5, List10);
wsCluster = theData->GetCluster();
itsHList = theData->GetHList();
itsVList = theData->GetVList();
((CList15 *)List15)->SetStyleLists (itsHList, itsVList);
((CList15 *)List15)->SetCluster (wsCluster);
((CList15 *)List15)->SetArray(wsCluster, FALSE);

//
// if the column and row label lists are empty,
// then fill them with default list entries.
// Otherwise, set the appropriate column widths
// and row heights.
//
InitCellStyle(&cellStyle);
TRY
{
    if(itsHList->GetNumItems() <= 0)
    {
        for(index=0; index < numCols; index++)
        {
            cellStyle.cellMetric = List15->GetColWidth (index);
            aStyle = new CCellData;
            aStyle->ICellData(cellStyle);
            itsHList->InsertAt(aStyle, index+1);
        }
    }
    else
    {
```

ICalcWindow
method cod
(concluded)

```
            for(index=0; index < numCols; index++)
            {
                aStyle = (CCellData *)itsHList->NthItem (index+1);
                cellStyle = aStyle->cellData;
                List15->SetColWidth (index, cellStyle.cellMetric);
                List5->SetColWidth (index, cellStyle.cellMetric);
            }
        }
        if(itsVList->GetNumItems() <= 0)
        {
            for(index=0; index < numRows; index++)
            {
                cellStyle.cellMetric = List15->GetRowHeight (index);
                aStyle = new CCellData;
                aStyle->ICellData(cellStyle);
                itsVList->InsertAt(aStyle, index+1);
            }
        }
        else
        {
            for(index=0; index < numRows; index++)
            {
                aStyle = (CCellData *)itsVList->NthItem (index+1);
                cellStyle = aStyle->cellData;
                List15->SetRowHeight (index, cellStyle.cellMetric);
                List10->SetRowHeight (index, cellStyle.cellMetric);
            }
        }
    }
    CATCH
    {
        ForgetObject (aStyle);
    }
    ENDTRY;
    ((CList15 *)List15)->Refresh();
}
```

Comparing this code with the version shown of **ICalcWindow** on page 215, we see that the major changes consist of sending the style list handles to the **CList15** class and creating default entries for the row and column style lists if they are empty. The creation of the default list entry code is placed inside a TRY block, just in case there isn't enough memory to create all of the list entries. The CATCH block will receive control if an error is detected, and then will pass the error on to the TCL to display a dialog indicating the source of the problem. Although the new initialization method is fairly long, the additional code is quite straightforward.

MakeStringObj Method Code

The code for the **MakeStringObj** method has been changed merely to provide for the additional style information that is stored in the workeheet cells. The code for this method is as follows:

```
CWSEntry *CCalcWindow::MakeStringObj (Cell aCell, StringPtr aString)
{
   CWSEntry  *aCellEntry;
   cellInfo     cellStyle;

   TRY
   {
      InitCellStyle (&cellStyle);
      aCellEntry = new CWSEntry;
      aCellEntry->IWSEntry ();
      aCellEntry->SetWSCell (aCell);
      aCellEntry->SetWSType (1);// string
      aCellEntry->SetWSText (aString);
      aCellEntry->SetWSEntry (aString);
      aCellEntry->SetWSValue (0.0);
      aCellEntry->SetWSStyle (cellStyle);
      return aCellEntry;
   }
   CATCH
   {
      ForgetObject (aCellEntry);
      return NULL;
   }
    ENDTRY;
}
```

The only change to the method from what was presented on page 237 is the calls to **InitCellStyle** and **SetWSStyle** for the cell entry. Each cell is initialized with a default style, which can be changed by the user through the application of the **Worksheet** dialog.

MakeValueObj Method Code

The **MakeValueObj** method has also been modified from the version presented on page 238, to include cell style information. The modified version of this code is as follows:

```
CWSEntry  *CCalcWindow::MakeValueObj (Cell aCell, double value,
                                       StringPtr  aString)
{
   CWSEntry     *aCellEntry;
   Str255       dispStr;
   decform      aFormat;
   extended     temp;
   cellInfo     cellStyle;

   TRY
   {
      InitCellStyle (&cellStyle);
      aCellEntry = new CWSEntry;
      aCellEntry->IWSEntry ();
      aCellEntry->SetWSCell (aCell);
      aCellEntry->SetWSType (2);// value
      aCellEntry->SetWSValue (value);
      aCellEntry->SetWSEntry (aString);
      aCellEntry->SetWSStyle (cellStyle);
      aFormat.style  = FIXEDDECIMAL;
      aFormat.digits = cellStyle.decimalDigits;
      x96tox80(&value, &temp);
      num2str(&aFormat, temp, dispStr);
      aCellEntry->SetWSText(dispStr);
      return aCellEntry;
   }
   CATCH
   {
      ForgetObject (aCellEntry);
      return NULL;
   }
    ENDTRY;
}
```

As with the **MakeStringObj** method, **MakeValueObj** deviates from its previous implementation only by the addition of calls to the **InitCellStyle** and **SetWSStyle** methods, with the number of decimal digits changed to the default value in the **cellStyle** structure.

UpdateMenus Method Code

The **UpdateMenus** method has been modified to handle disabling the **Notebook** command when the **CalcWindow** is frontmost, and enabling or disabling the **Worksheet** command, depending on whether a cell is selected or not, respectively. The modified code is as follows:

```
void  CCalcWindow::UpdateMenus(void)
{
   inherited::UpdateMenus ();

   gBartender->DisableCmd (cmdClose);
   gBartender->DisableCmd (cmdNotebook);
   if(List15->HasSelection())
   {
      gBartender->EnableCmd  (cmdWorksheet);
   }
   else
   {
      gBartender->DisableCmd (cmdWorksheet);
   }
}
```

DoCommand Method Code

Although the code for the **DoCommand** method wasn't shown in Chapter 8, it was mentioned there that it wasn't necessary to make any changes to the generated code for that version of the Ensemble application. In the new version, we must implement the invocation of the **Worksheet** command and the subsequent actions, based on the user's actions when interacting with the dialog. Therefore, quite a bit of code has been added to the **DoCommand** method. Because this code is not used in any other place, it made sense to have it in-line, for ease of reference. The new **DoCommand** method is as follows:

DoCommand *method code (beginning)*

```
void  CCalcWindow::DoCommand (long theCommand)
{
   Cell        aCell;
   short       height;
   short       width;
   short       changeStyle;
   cellInfo    styleInfo;
   cellInfo    oldStyle;
   CWSEntry    *anEntry;
   long        param;

   switch (theCommand)
```

DoCommand
method code
(continued)

```
{
   case cmdEnterButton:
   {
      DoEnterButton ();
      break;
   }
   case cmdCancelButton:
   {
      DoCancelButton ();
      break;
   }
   //
   // added case
   //
   case cmdWorksheet:
   {
      if(List15->HasSelection())
      {
         SetPt (&aCell, 0, 0);
         List15->GetSelect (TRUE, &aCell);
         itsSelectedCell = aCell;
         param = *(long *)&aCell;
         InitCellStyle(&styleInfo);
         theRowStyle = (CCellData *)itsVList->NthItem((long)aCell.v+1);
         theColStyle = (CCellData *)itsHList->NthItem((long)aCell.h+1);
         if((anEntry = (CWSEntry *)wsCluster->FindItem1
            (FindWSCell, param)) != NULL)

         {
            styleInfo = anEntry->GetWSStyle();
         }
         itsCellData = new CCellData;
         itsCellData->ICellData(styleInfo);
         itsCellStatus.itsCell = aCell;
         itsCellStatus.cellHeight = List15->GetRowHeight(aCell.v);
         itsCellStatus.cellWidth  = List15->GetColWidth (aCell.h);
         itsCellStatus.rowColCell = 2;  // cell
         itsCellStatus.changeStyle = 0;
         itsCellStatus.modified = FALSE;
         DoWorksheet (this);
         if(itsCellStatus.modified)
         {
            ((CEnsembleData *) itsData)->SetDirty (TRUE);
            //
            // parameters that affect this cell
            // may have been modified. We have
            // to check it out.
            //
            height = itsCellStatus.cellHeight;
```

DoCommand *method code (continued)*

```
width = itsCellStatus.cellWidth;
changeStyle = itsCellStatus.changeStyle;
switch (itsCellStatus.rowColCell)
{
  case 0: // row
  {
    List15->SetRowHeight(aCell.v, height);
    List10->SetRowHeight(aCell.v, height);
    oldStyle = theRowStyle->cellData;
    oldStyle.cellMetric = height;
    theRowStyle->cellData = oldStyle;
    if(changeStyle)
    {
      itsCellData->cellData.isDefault = 0;
      styleInfo = itsCellData->cellData;
      styleInfo.cellMetric = height;
      theRowStyle->cellData = styleInfo;
    }
    break;
  }
  case 1: // col
  {
    List15->SetColWidth(aCell.h, width);
    List5->SetColWidth(aCell.h, width);
    oldStyle = theColStyle->cellData;
    oldStyle.cellMetric = width;
    theColStyle->cellData = oldStyle;
    if(changeStyle)
    {
      itsCellData->cellData.isDefault = 0;
      styleInfo = itsCellData->cellData;
      styleInfo.cellMetric = width;
      theColStyle->cellData = styleInfo;
    }
    break;
  }
  case 2: // cell
  {
    if(changeStyle)
    {
      itsCellData->cellData.isDefault = 0;
      styleInfo = itsCellData->cellData;
      param = *(long *)&aCell;
      if((anEntry = (CWSEntry *)wsCluster
        ->FindItem1 (FindWSCell, param)) != NULL)
      {
        anEntry->SetWSStyle(styleInfo);
      }
      else
```

DoCommand
method code
(concluded)

```
                        {
                            anEntry = MakeStringObj (aCell, "\p'");
                            anEntry->SetWSStyle(styleInfo);
                            wsCluster->Add(anEntry);
                        }
                    }
                    break;
                }
            }
        }
        itsCellData->Dispose();
    }
    break;
}
default:
{
    inherited::DoCommand (theCommand);
    break;
}
}
}
```

There is quite a bit of code in the modified **DoCommand** method. It is necessary to set things up prior to calling the **DoWorksheet** method: We must access the row, column, and cell styles, as well as the row height and column width settings for the selected cell. The dialog is initialized to provide the user with the settings for the selected cell; however, as the description of the **GetSettings** method (page 313) indicates, the user has the privilege of modifying the corresponding row and column settings, instead of the selected cell settings.

After the **Worksheet** dialog has been dismissed, the code must determine whether the settings were modified. This is reflected in the **modified** field of the **itsCellStatus** structure, which is updated at the conclusion of running the dialog.

If the user clicked the **OK** button to dismiss the dialog, then the **modified** field will be set to TRUE. If the **Cancel** button was used to dismiss the dialog, then the code in the **DoCommand** method will dispose of the new cell entry it created and exit.

If the settings were modified, then the **DoCommand** method determines whether a row, column, or cell style was affected.

If it was a row style, then the **cellMetric** field will hold the new row height value, which is changed for both the main worksheet and the row label lists.

In addition, if the **changeStyle** flag is set, the row style is updated to reflect the change. Similar code is used to handle changes to the column style. In this case, the **cellMetric** field holds the column width, which is installed in both the main worksheet and the column label lists. If the **changeStyle** flag is set, then the column style is updated to reflect the associated change.

Finally, if the specific selected cell was modified, then it is determined whether the cell has already been defined. If not, then a dummy, empty string object is constructed, the cell style is applied to that, and the instance is entered into the main worksheet list. If the cell already exists, then the **SetWSStyle** method is called to change its style. In any case, whether or not changes were made, the created **CCellData** object is disposed of.

GetCellData and SetCellData Methods

The **GetCellData** and **SetCellData** methods provide access to the font, size, style, and justification data kept for the current cell, for use by the **Worksheet** dialog. The code for both methods is as follows:

```
CCellData*CCalcWindow::GetCellData ()
{
   return itsCellData;
}

void   CCalcWindow::SetCellData (CCellData *aCellData)
{
   itsCellData->cellData = aCellData->cellData;
}
```

GetCellStatus and SetCellStatus Methods

These methods are also used only by the **Worksheet** dialog, to access the status data associated with the currently selected cell. The code for them is as follows:

```
cellStatus   CCalcWindow::GetCellStatus (void)
{
   return itsCellStatus;
}

void   CCalcWindow::SetCellStatus (cellStatus aStatus)
{
   itsCellStatus = aStatus;
}
```

GetColData and GetRowData Methods

The **Worksheet** dialog also needs access to the column and row style settings. **GetColData** and **GetRowData** provide the capability for the dialog to acquire the column and row style data through the use of these access methods.

Note that there are no corresponding **SetColData** or **SetRowData** methods, because the selected row or column style data are stored into the **itsCellData** instance variable using the **SetCellData** method when the dialog is dismissed. The code for the **GetColData** and **GetRowData** methods is as follows:

```
CCellData *CCalcWindow::GetColData()
{
   return theColStyle;
}

CCellData *CCalcWindow::GetRowData()
{
   return theRowStyle;
}
```

InitCellStyle Method Code

Whenever a new cell is created, it is given a set of default settings that reflect the style of the main worksheet as a whole. The **InitCellStyle** method is called from a number of other methods in the **CCalcWindow** class. The code for **InitCellStyle** is as follows:

```
void  CCalcWindow::InitCellStyle (cellInfo *cellStyle)
{
   TextInfoRec    textInfo;
   ((CTextEnvirons *)List15->itsEnvironment)->GetTextInfo (&textInfo);
   cellStyle->isDefault = 1;
   cellStyle->cellMetric = 0;
   cellStyle->fontNumber = textInfo.fontNumber;
   cellStyle->fontSize = textInfo.theSize;
   cellStyle->fontStyle = textInfo.theStyle;
   cellStyle->fontAlign = teFlushLeft;
   cellStyle->decimalDigits = 2;
   cellStyle->commas = 0;
   cellStyle->dollars = 0;
}
```

Adding New CWSEntry Methods

In order to support the additional style data that are stored in worksheet cell entries, two new methods have been added. These methods support the retrieval and setting of the data in the **CWSEntry** class. Both methods are relatively trivial, but reflect our desire to insulate the programmer from having to access these fields directly.

GetWSStyle & SetWSStyle Method Code

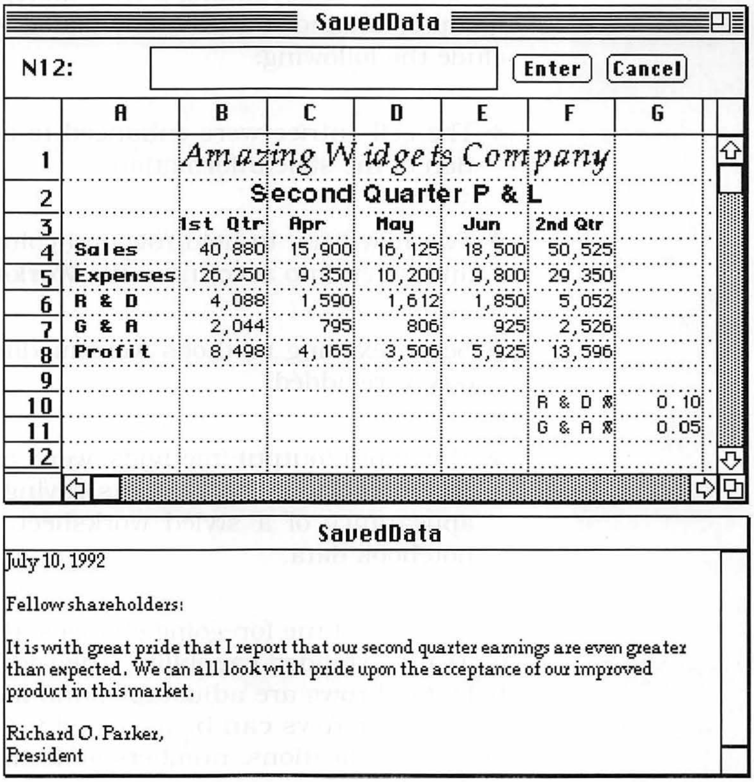
These methods provide access to the **cellInfo** style structure for an individual worksheet cell entry, as shown on page 286, pertaining to the **CCellData** header file declarations. The code for the new **GetWSStyle** and **SetWSStyle** methods is as follows:

```
cellInfo   CWSEntry::GetWSStyle (void)
{
   return cellData;
}
void       CWSEntry::SetWSStyle (cellInfo theCellData)
{
   cellData = theCellData;
}
```

This concludes the description of the customized code implemented for the new **Worksheet** dialog. Although there were a considerable number of modified and added methods, we were able to save almost every line of code that was previously written. This is an important consideration when designing a fairly substantial application, as this is. When the number of statements that you have to discard and rewrite becomes large, it is an indication that the initial design was not amenable to evolutionary development.

As a final salute to the power of the new worksheet capabilities, we offer the following screen shot, showing the Ensemble application, in all its glory, running with a modified set of worksheet entries. The new appearance of the Macintosh screen is shown in Figure 11-2.

Figure 11-2
Appearance of the Ensemble application with styles applied to the worksheet rows, columns, and cells



Notice that the main title, *Amazing Widgets Company*, has been set in 18-point Palatino italic type, that the subhead is set in 12-point Helvetica type, and that the row and column headings inside the worksheet have been set in Helvetica type and have also been centered. The dollar figures are shown with no decimals, contain commas, and are right justified. The row headings are left justified and the column has been widened to accommodate the longer headings. The illustration you see was read in, just as pictured, from a file containing the worksheet and notebook data, after the data had been styled and saved.

Summary of the Changes to Ensemble

This and the preceding two chapters have focused on the addition of styling information to the main worksheet. This was accomplished by adding a new dialog and a corresponding menu command to invoke the dialog. Additional changes include the following:

- ❖ The cell entries were enhanced to accommodate the addition of the style information.
- ❖ Two new lists, to hold row and column styles, were added and taken into account in the **Worksheet** dialog.
- ❖ Seven existing methods were modified and 12 new methods were added.
- ❖ The input/output methods were enhanced to provide a new file format that permits saving and restoring the full appearance of a styled worksheet, as well as the styled notebook data.

As a result of the foregoing changes, the main worksheet has a greatly extended capability. The widths of columns and the heights of rows are adjustable, and individual cells or entire columns or rows can be assigned independent fonts, sizes, styles, justifications, numbers of decimal digits, and commas within them.

The next three chapters will discuss the addition of a graphing capability to the Ensemble application. This may be a good time to step back and reflect on the incredible evolution of the Ensemble application up to now.

Exercises

1. Explain why a new **CCellData** class was added to the application. What is the rationale for defining a separate class to encapsulate data pertaining to a worksheet cell, especially if there is only a single method in the associated class declaration?

2. Describe the purpose of the **itsHList** and **itsVList** lists that are defined in the **IEnsembleData** method of the **CEnsembleData** class. Why are instances of **CList** used in this case, rather than **CCluster** or **CArray**?

3. What information is being written for the worksheet entries in the new file format? (*Hint*: Look in the **WriteWS-Entries** method in the **CEnsembleData** class.) Why is this information sufficient to reconstruct each of the entries?

4. Examine the code for the Worksheet dialog. Describe in what way the dialog reacts to the selection of **Row**, **Column**, or **Cell** radio buttons. What information is pertinent to each of these selections?

5. Describe the purpose of the **GetSettings** method in the **CWorksheet** class. What function does this method serve? What other methods call this method, and for what reason?

6. Describe the dynamics in the operation of the **DoCommand** and **ProviderChanged** methods for the Worksheet dialog. In what way do these methods modify the dialog's appearance?

7. Explain the purpose of the new **DrawCell** method in the **CList10** class. Why was it necessary to modify this method from the implementation shown in Chapter 8?

8. In the main worksheet list class, **CList15**, what was added to the **GetCellText** method to implement the features of the Worksheet dialog?

9. Why has a **DrawCell** method been added to the **CList15** class? What primary feature of object-oriented programming does it illustrate?

10. Describe the operation of the **DrawWSCell** method in the **CList15** class. What features of the Worksheet dialog does this method implement?

11. Describe the operation of the **ICalcWindow** method in the **CCalcWindow** class. What purpose does each section of the code serve?

12. Describe the operation of the **DoCommand** method in the **CCalcWindow** class. How does this method react to changes made in the Worksheet dialog?

13. If the Ensemble application is modified to provide for multiple contiguous or discontiguous cell selections, how will the changes need to be reflected in the Worksheet dialog?[1] Also, in what way does this dialog relate to multiple selections?

14. If the Ensemble application is modified to support in-cell entry and editing, how would these features be implemented for multiple cell selections? Is it appropriate to do so? How does in-cell editing relate to the Worksheet dialog's design and implementation.[2]

---

1. The proposed modifications to the Worksheet are described in the exercises for Chapter 8. Modifying the worksheet to provide useful functionality if multiple cells are selected is a very complex task. It could be assigned as an extra-credit project.
2. In-cell entry and editing were first introduced as a user interface concept in the exercises for Chapter 6. Carrying through the design and implementation of these features to include their impact on the design of the Worksheet dialog is an ongoing and complex task that could be assigned as an extra-credit project.

Chapter 12

Adding a Graph Window to Ensemble

This chapter describes how a third simple window is added to the Ensemble application to support graphs that we will create using data from the worksheet window.

The window design for displaying graphs is quite simple. It consists merely of a blank window—much like the worksheet window—that contains a scroll pane, with both horizontal and vertical scroll bars, and a panorama to contain the graphic displays.

In addition to the window, we will also present the design of a dialog box for selecting the graph type, its labels (if any), and its scaling settings.

The addition of the dialog will also require that a third command be added to the **Format** menu. We will be call this command **Chart**, which when chosen will cause a dialog box containing parameters for the graph window to be shown.

The remainder of this chapter discusses the step-by-step technique for creating the window, dialog, and menu items within AppMaker. In addition, default code will be generated, added to the THINK C project, and compiled. The two chapters that follow this will, in turn, describe the generated code for the graph window additions and explain all the custom additions that make the window fully functional.

Creating the GraphWindow with AppMaker

Adding the graph window, which we will call **GraphWindow**, to the Ensemble application is a simple matter. To give you an idea of how the window will look, it is shown in Figure 12-1.

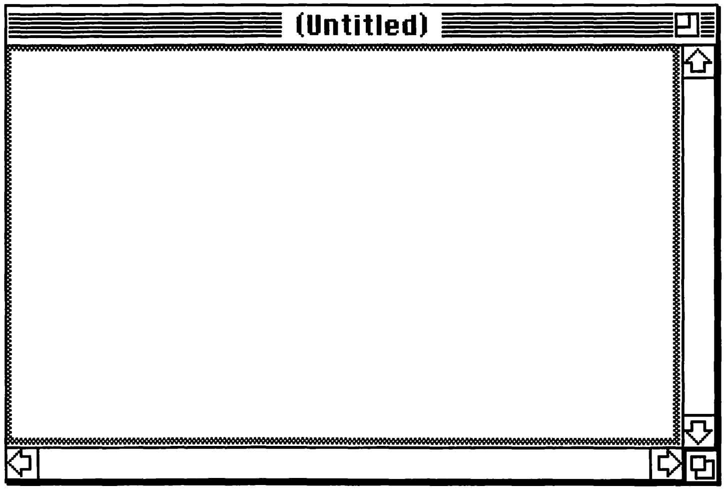


Figure 12-1
Completed GraphWindow appearance

Notice that the basic window does not have a close box, as was the case with the worksheet window. It does have both horizontal and vertical scroll bars, and a panorama in which the graphs are drawn. The step-by-step approach for creating this window is as follows:

1. Launch AppMaker by double-clicking on the Ensemble resource file.

2. Select the **Tools as Text** option from the **View** menu, and then pull down the **Select** menu and choose the **Windows** command.

3. Pull down the **Edit** menu and select the **Create Window** command. This will create a new window. Its initial size is not too important, as it can be readily resized by the user, and will be staggered with respect to the other windows when it is created at run time. The window will have the characteristics shown in Figure 12-2.

4. Select the **ZoomDoc** window type, but uncheck the "Has Close box" option and leave the "Visible at Startup" option checked. Name the window **GraphWindow**, and give it a title of **(Untitled)**, as shown. Then, click OK to dismiss the new window dialog.

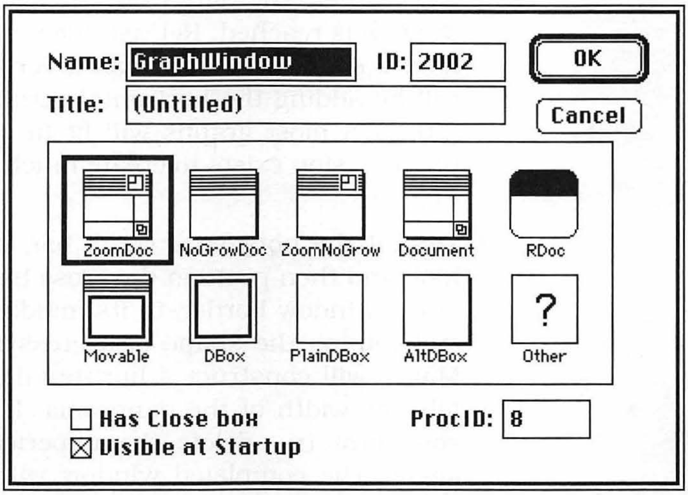


Figure 12-2
GraphWindow features

5. When the window's features have been defined, it will have the appearance shown in Figure 12-3. Make the window large enough so that you can add the new elements in the following steps.

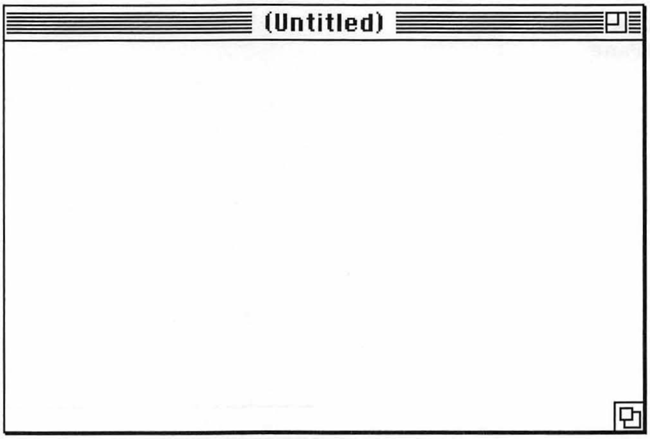


Figure 12-3
Plain GraphWindow appearance

6. Pull down the **Tools** menu and select the **CScrollPane** tool.

7. Position the cross hairs at the upper left corner of the blank portion of the window (below the title bar), and drag down and to the right until the bottom right corner of the window is reached. Release the mouse. You will have created a scroll pane that has a vertical scrollbar only. We will be adding the horizontal scroll bar in the next step. Although most graphs will fit in a modest size window, the provision exists to create much larger graphs.

8. To add the horizontal scroll bar, choose the **CScrollbar** tool, and then position the cross hairs right at the bottom of the window border, in its middle, but while the cursor still retains the shape of a cross hair. Click once. AppMaker will construct a horizontal scroll bar that exactly fills the width of the panorama. If you don't succeed on your first try, delete the imperfect scroll bar and try again. The completed window with the scroll pane and both scroll bars installed has the appearance shown in Figure 12-4.

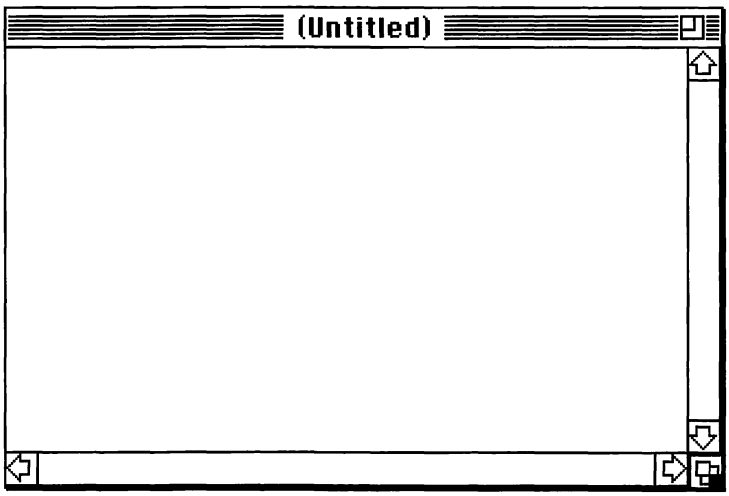


Figure 12-4 **GraphWindow** with the **CScrollPane** installed

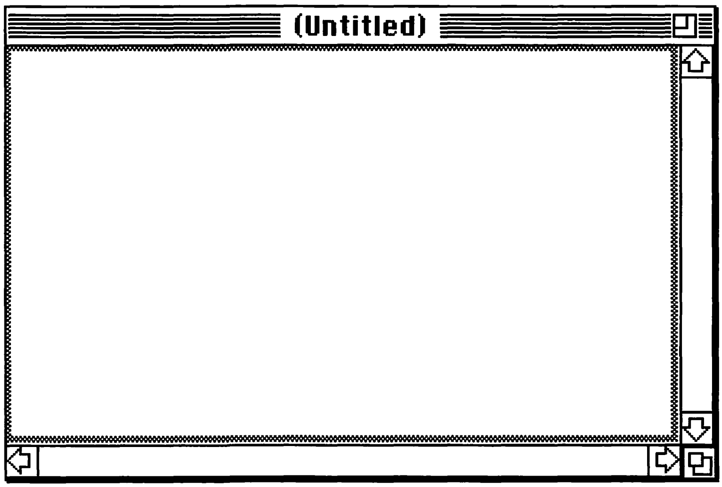
9. After the scroll pane is installed, we need to add a panorama, which is a pane that can be arbitrarily large, to hold very large images or data. In our case, we will be customizing this pane to be the size of a letter-sized page of paper; however, that step will be undertaken in Chapter 14 when we discuss the custom code additions to the **GraphWindow** class. We don't have to specify the eventual size of the panorama at the time it is designed

within AppMaker. To create the panorama pane, pull down the **Tools** menu and choose the **CPanorama** command.

10. Once again, to create the panorama, you position the cursor's cross hairs at the upper left of the blank portion of the window, within the scroll pane, and drag down and to the right until you reach the intersection of the left edge of the vertical scrollbar and the top edge of the horizontal scroll bar. The result of installing the panorama is shown in Figure 12-5.

The panorama appears with a gray border within AppMaker; however, the border will not show in the running application. Figure 12-5 is identical in appearance to the completed window shown in Figure 12-1.

Figure 12-5
GraphWindow with **CPanorama** installed



Adding the Format Chart Menu Command

The next series of steps shows how the new menu item is added to the Ensemble application. This procedure is identical to the steps that you have already followed in constructing **Format** menu items in Chapters 5 and 9.

1. Pull down the **Select** menu and choose **Menus**.

2. Double-click on the **MainMenu** entry in the list of menu bars at the right of your screen. This will show the main menu bar for the Ensemble application.

3. Click on the **Format** menu to drop down the menu's entries, and create a new entry by clicking below the **Worksheet** command. Enter the information for the **Chart** command, as shown in Figure 12-6.

Note that the **Chart** command has been given a command number of 2002, which is one larger than the previous entry. When you use the TCL, you must choose command numbers that are larger than 1024. Starting with 2000 is a completely safe approach.

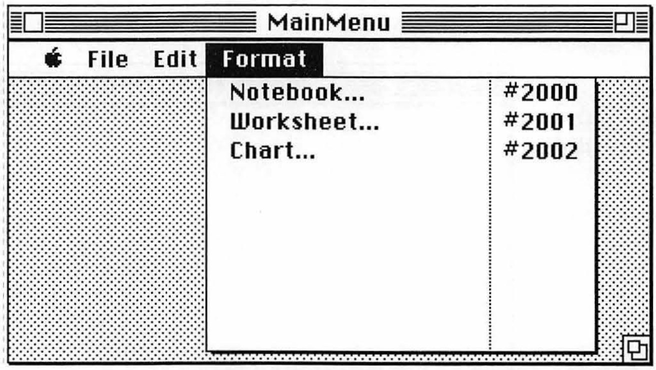


Figure 12-6
Format Chart menu command added

Adding the Format Chart Dialog

The **Format Chart** dialog is a fairly complex addition to the application. The individual dialog elements are quite simple; however, there are more than a few that need to be created.

At this point in the development of the application, you have doubtless had quite a bit of experience working with AppMaker's tools, so we will present only the completed appearance of this dialog and describe a few of its features in more detail. The final dialog is shown in Figure 12-7.

Notice that the upper left quadrant of the dialog contains a labeled group (**CLabeledGroup**) with the label **Chart Type**. This group includes three radio buttons (**CRadioControl**)

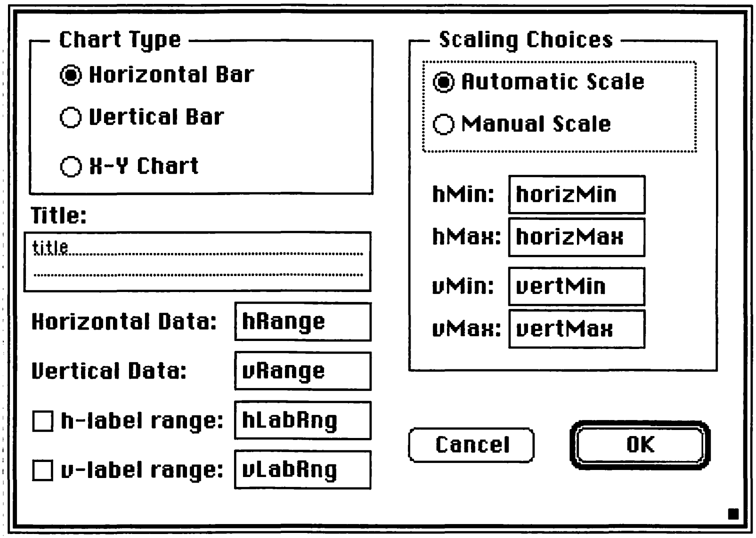


Figure 12-7
Completed **Chart** dialog

that enable the user to select the type of chart to be produced. The choices include both horizontal and vertical bar charts and an X-Y Chart (one that plots points at the intersection of horizontal and vertical coordinate values) option. The **Item Info** settings for the **Chart Type** group are shown in Figure 12-8. The radio buttons can be installed inside the group, visually, by inspection of the figure.

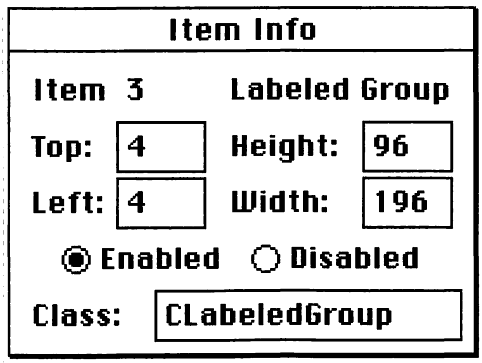


Figure 12-8
Chart Type labeled group Item Info settings

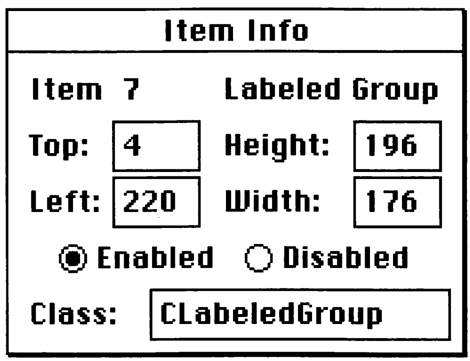
Immediately below the **Chart Type** group is an EditText field (**CEditText**), to allow the user to type in a title for the chart. Immediately above that field is the static text (**CStaticText**) label called **Title**.

Because the dialog must allow for both X and Y data ranges (in the case of the X-Y Chart option), we provide static text labels of **Horizontal Data** and **Vertical Data**, with appropriate dialog text fields to their right (**CDialogText**).

In addition, depending on the type of chart, the user will have the option of supplying a cell or data range holding the title for the horizontal or vertical title axis of the chart. To implement the optional fields, checkbox items (**CCheckBox**) labeled **h-label range** and **v-label range** have been added. To their immediate right, we have also added corresponding dialog text (**CDialogText**) fields to hold the label ranges.

Almost the entire right half of the dialog is taken up with another labeled group that specifies the chart's scaling options. The **Item Info** settings for this interface element are shown in Figure 12-9. Note that the two radio buttons that select **Automatic Scaling** or **Manual Scaling** are placed inside their own radio group (**CRadioGroup**) pane.

Figure 12-9
Scaling Choices labeled group Item Info settings



The remaining fields in the **Scaling Choices** group are the static text labels **hMin**, **hMax**, **vMin**, and **vMax**, with dialog text fields to their right in which the scale values are entered.

The automatically provided **OK** and **Cancel** buttons complete the dialog. It is important for you to note that we have typed names into each of the EditText and DialogText fields. These names will be used by AppMaker in the generated code to refer to the corresponding user interface elements. If you fail to type in the names, then AppMaker will invent names that will not correspond to the code that is presented in subsequent chapters.

When you create the **Title CEditText** field, take a moment to click inside the field and type **title** as its name. In a similar fashion, type in the names for the horizontal and vertical data range fields, as well as the label range fields. These are shown in Figure 12-7 as **hRange**, **vRange**, **hLabRng**, and **vLabRng**, respectively.

On the right side of the dialog, the scale fields are similarly named by typing the field names inside the dialog text elements after they have been created. The names **horizMin**, **horizMax**, **vertMin** and **vertMax** will be used for these elements in the generated code.

Generating the New Code

After the **GraphWindow**, the **Format Chart** command, and the **Chart** dialog have been created, you will need to generate the code to implement these user interface features. To do so, pull down the **File** menu and choose the **Generate** command. You will see the dialog shown in Figure 12-10. Click the **Generate** button, as shown in the figure.

Figure 12-10 shows eight new files, in addition to the files for the various superclass modules. Four of the new files are **ZGraphWindow.h**, **ZGraphWindow.c**, **GraphWindow.h**, and **GraphWindow.c**. These files implement the **GraphWindow** class and its methods. The other four files, **ZChart.h**, **ZChart.c**, **Chart.h**, and **Chart.c**, implement the **Chart** dialog's functionality. After the code has been generated, quit AppMaker and proceed to the next step.

Compiling the Generated Code

After the new versions of the existing superclass files, as well as the newly generated files, have been written, you need to add the new files to your THINK C project and compile them. The following steps will lead you through this process:

1. Launch THINK C by double-clicking on the **Ensemble.π** project file, and then pull down the **Source** menu and

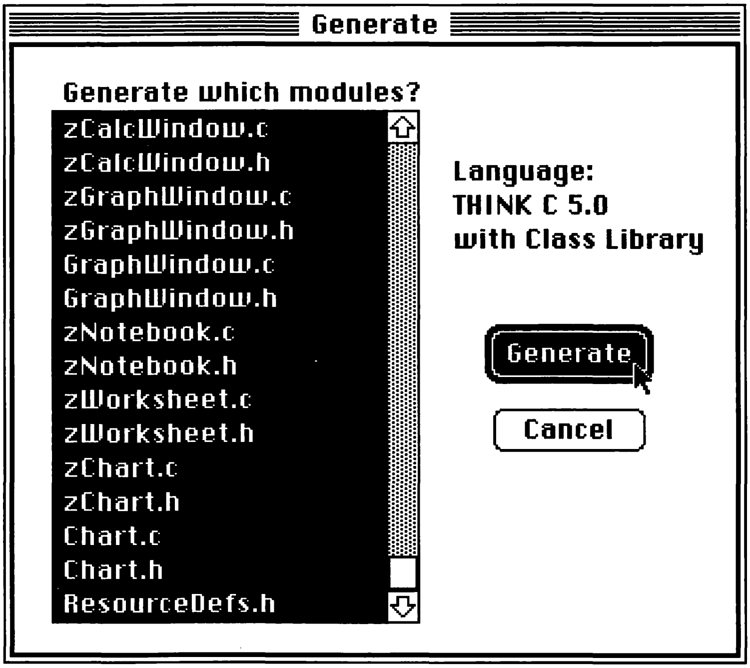


Figure 12-10
AppMaker's
Generate dialog

select the **Add** command. THINK C will present a dialog box that lists the new files that aren't currently part of the project, as shown in Figure 12-11. Click the **Add All** button, as shown in the figure.

2. After clicking the **Add All** button, the four source files will move to the lower pane in the dialog. The next step is to click the **Done** button, as shown in Figure 12-12. After this button is clicked, all four files will be added to the first segment of the existing project file.

3. The completed project file will list all the files for the Ensemble project in the first segment, as shown in Figure 12-13. Because code segments are limited in size to 32 kilobytes, we are going to have to move some files, which we will do in the next few steps.

4. If we tried to compile the project at this point, the segment would not exceed the 32 kilobyte limit; however, because it will exceed that limit when we add the custom code that is described in Chapter 14, it is useful to move

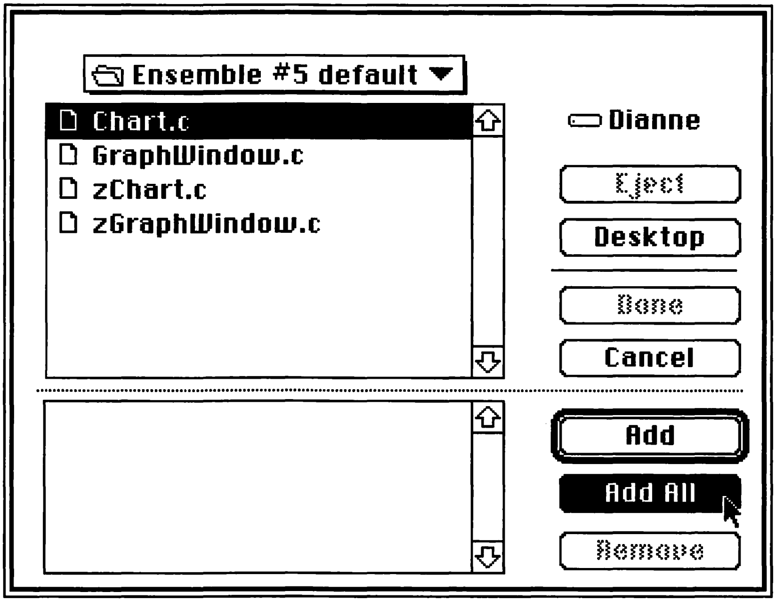


Figure 12-11
Selecting **Add All** in the THINK C **Add** dialog

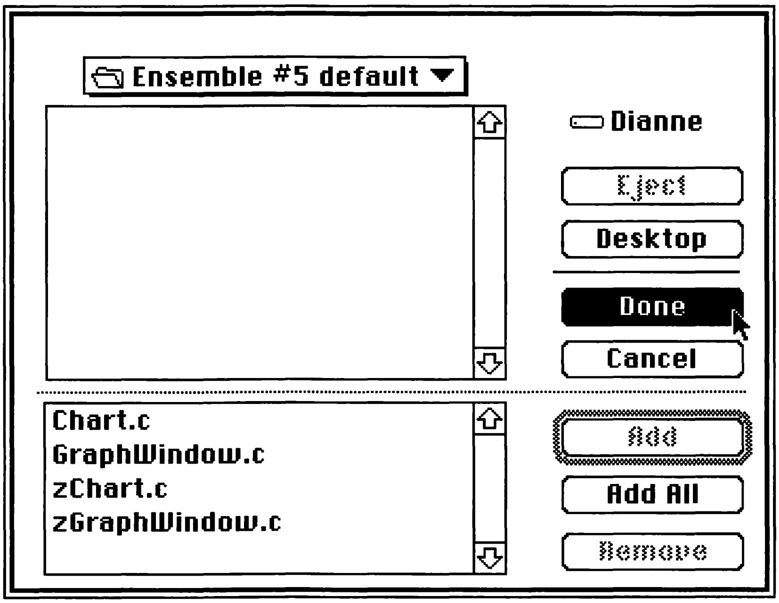
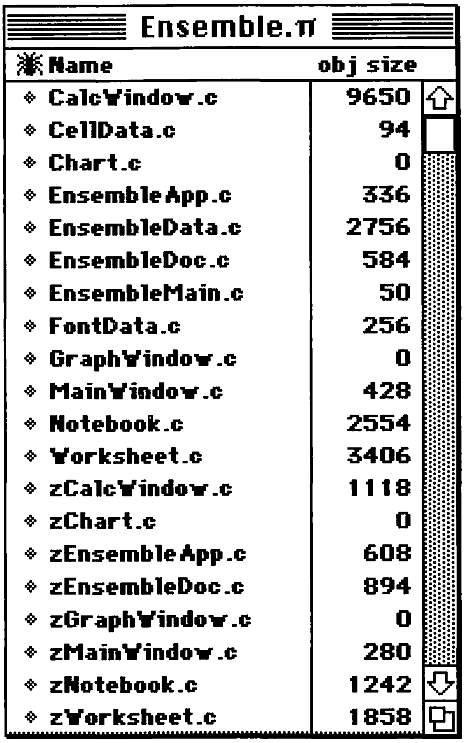


Figure 12-12
Clicking the **Done** button in the THINK C **Add** dialog

some of the files to a new segment. This process is a bit awkward, but straightforward. It requires four steps:

Figure 12-13
All files added to the first Ensemble.π project segment



a. We have decided to move the code for the three subclass files that implement the functionality of the **MainWindow**, **CalcWindow**, and **GraphWindow** to their own segment. We commence this procedure by clicking on the **CCalcWindow.c** file and dragging it below the bottom of the list. By holding down the mouse button, the list will scroll downward. Continue holding down the button until the bottom of the list is exposed. There will be a gray dividing line that shows the bottom of the last segment. Release the button when the outline of the **CCalcWindow.c** file's rectangle is just below that line, as shown in Figure 12-14. The **CalcWindow.c** file will be in a new segment, by itself, at this point.

b. If you hold down the Option key, and then click and drag a file in the project window, the entire segment containing the file will be moved. In this step, press and hold down the Option key, and then click the

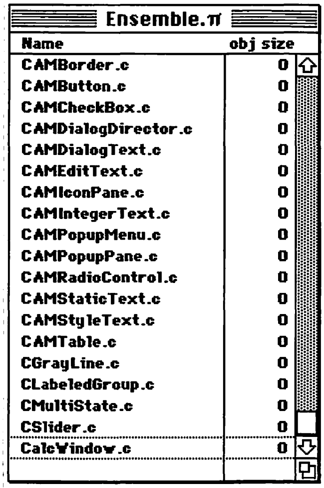


Figure 12-14
CalcWindow.c moved into a new segment

mouse button with the cursor on the **CCalcWindow.c** file and drag the file back up above the top of the list. The list will scroll back toward the top. Continue to hold down the Option key and position the mouse above the list, so that it will continue to scroll. When the list reaches the top, and the rectangle that represents the **CalcWindow.c** file exactly straddles the dividing line between the first and second segment, as shown in Figure 12-15, release the mouse button and then the Option key. A segment containing just the **CCalcWindow.c** file will be moved to the position indicated by the outline.

c. The new segment will appear in the project list as shown in Figure 12-16. We could have just used the new segment at the bottom of the list, as shown in Figure 12-15; however, it is easier to keep track of the files that are unique to the current project if they are together.

d. The last step involves moving the **MainWindow.c** and **GraphWindow.c** files to the new segment by clicking on each of them and dragging them, one by one, so that they overlap the **CCalcWindow.c** file in the project window. This will cause them to be placed in that segment when the mouse button is released. *Do Not* hold down the Option key for this step, as the

Figure 12-15
Dragging the file until the rectangle straddles the first and second segments

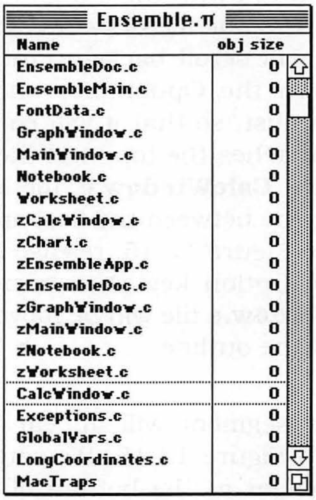


Figure 12-16
New segment containing just the **CCalcWindow.c** file

entire first segment would be moved into the new second segment, and the first would automatically be deleted, leaving us back where we started. After moving the other two files to the new segment, the project list will have the appearance shown in Figure 12-17.

5. Now that the files are rearranged, you can compile the project to update the object files to correspond to the newly generated versions and also compile the new files

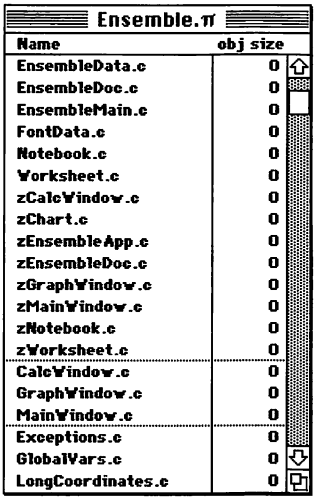


Figure 12-17
All three window files moved into the second segment

for which no object files currently exist. This is accomplished by pulling down the **Source** menu and choosing the **Make** option. When you do so, THINK C will display the dialog shown in Figure 12-18. Click the **Use Disk** button, as shown.

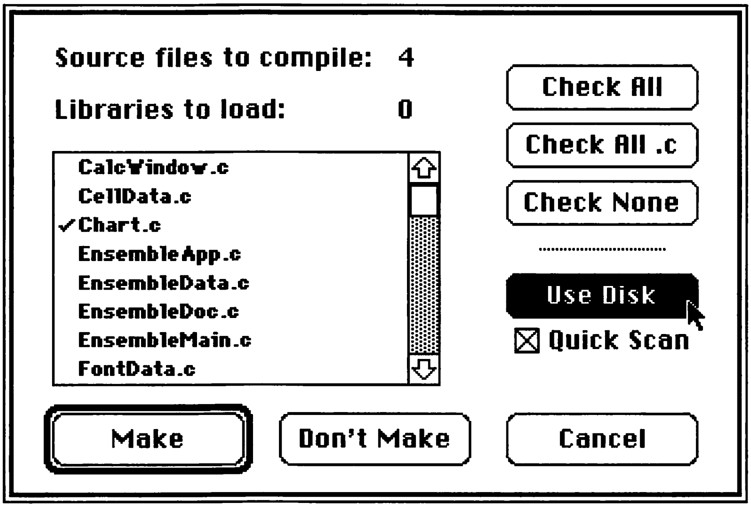


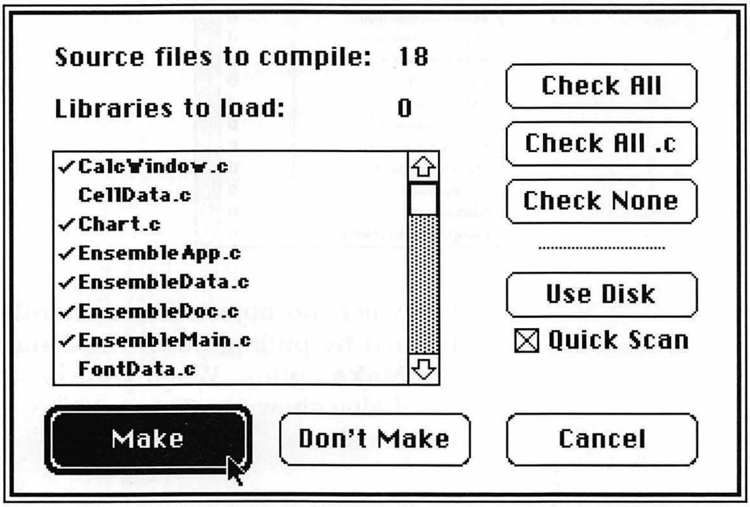
Figure 12-18
THINK C **Make** dialog, **Use Disk** selected

6. Clicking the **Use Disk** button forces THINK C to reexamine the modification dates of the project's files. In the process of doing this for the **Ensemble.π** project, it will find

that most of the files unique to the project will need to be recompiled.

7. When one or more files are found to require recompilation, the **Make** button is enabled. Click this button, as shown in Figure 12-19.

Figure 12-19
Clicking the **Make** button to recompile the project's files



After the **Make** button has been clicked, THINK C will begin compiling all the files that have changed since the last time the project was built. When the compilation is complete, you can run the application to see what it looks like with the addition of the new window. Pull down the **Project** menu and choose the **Run** command. THINK C will display the debugger windows at this point. Click the **Go** button in the debugger's source window, and the application will begin execution. The new windows will appear.

You can move and resize these windows, as shown in Figure 12-20. In this figure, the worksheet window is on top, the text window is on the bottom, and the chart window is at the right. Both the text and chart windows have a similar appearance; however, the text window lacks a horizontal scroll bar, while the graph window has both horizontal and vertical scroll bars. To see the new dialog that we have just constructed, pull down the **Format** menu and choose the **Chart** command. The default version of the dialog is shown in Figure 12-21.

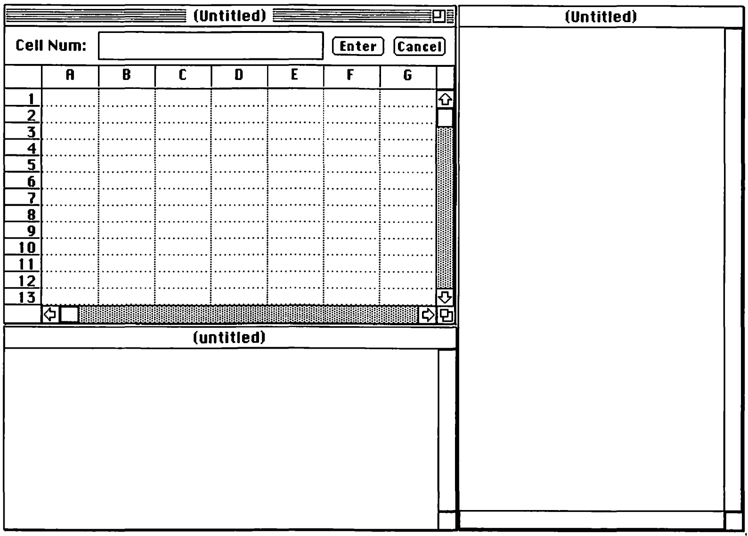


Figure 12-20 Default version of Ensemble running with 3 windows

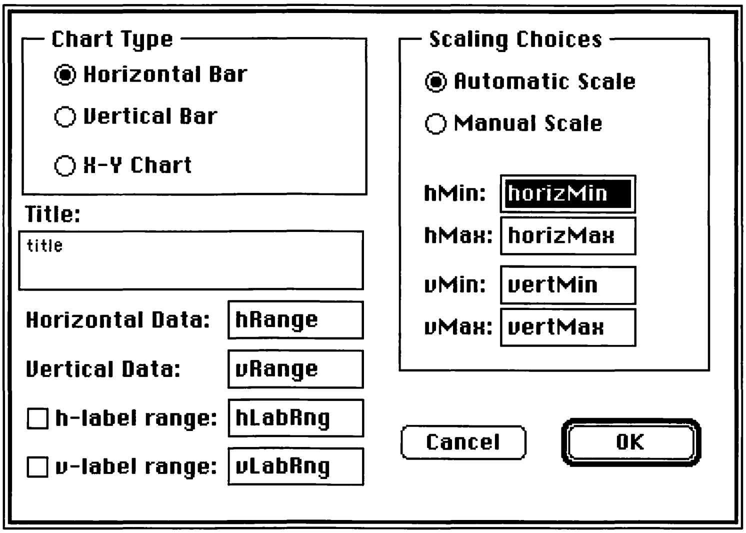


Figure 12-21 Default appearance of **Format Chart** dialog

The default version of the dialog shows all the fields and controls enabled. This complicates the appearance of the dialog. The custom code that is described in Chapter 14 will disable some of the controls and hide various fields, depending on what type of chart and whether automatic or manual scaling has been chosen. The set of files that comprise the Ensemble application is shown in Figure 12-22, as these files appear in the Finder.

Figure 12-22 Ensemble application's files as seen in the Finder

The next chapter will describe the newly generated code for the **GraphWindow** and its accompanying **Chart** dialog.

Exercises

1. Explain why an instance of the **CPanorama** class was chosen for inclusion in the ScrollPane in the new **GraphWindow**?

2. What kind of information would be required if the Chart dialog made provision for pie charts?

3. What other types of charts might be useful? Describe the applications where these charts would be used and describe the user interface modifications that would be necessary to support them.

4. Redefine the Chart dialog's user interface to support the implementation of pie charts.[1]

5. Describe why it was necessary to create a new segment to hold the "window" classes. Would it make any difference if some other classes were placed in this segment instead?

1. Implementing pie charts is a fairly complex task. Making appropriate provisions for just the labeling of the pie slices is a very involved problem. Therefore, this exercise should be treated as an extra-credit project.

Chapter 13

Examining the GraphWindow Code

This chapter describes the newly generated code that adds a graph window and associated dialog box to the Ensemble application.

As has been the case in all additions to the Ensemble application, none of the existing custom code has been modified by AppMaker when it generates new code. Only the superclass modules, whose names begin with the letter **z**, are regenerated. In addition, new modules are generated to implement the added functions. The modules that we will examine in this chapter are as follows:

- **GraphWindow.c** contains the subclass methods to override and supplement the methods generated in its superclass. This will be the file in which the majority of the custom additions to support drawing the various graphs will be implemented.
- **GraphWindow.h** contains the class declarations for the subclasses defined in the **GraphWindow.c** file.
- **zGraphWindow.c** contains the superclass methods that initialize the default behavior of the graph window.
- **zGraphWindow.h** contains the declarations for the classes and methods defined in the **zGraphWindow.c** module. Many of the methods declared in **zGraphWindow.h** are intended to be overridden by corresponding subclass methods declared in **GraphWindow.h**.
- **Chart.c** contains the subclass methods to override and supplement the methods generated in the **zChart.c** superclass.
- **Chart.h** contains the declarations for the methods defined in the **Chart.c** source file.

- **zChart.c** contains the superclass methods that establish the default behavior of the **Chart** dialog.

- **zChart.h** contains the superclass declarations for the methods contained in the **zChart.c** file.

The preceding modules contain all of the default-generated code to implement both the **GraphWindow** and its associated **Format Chart** dialog box.

The following section describes the classes and methods that provide the basis for the graphing functions to be added to the Ensemble application.

The Final Structure of the Ensemble Application

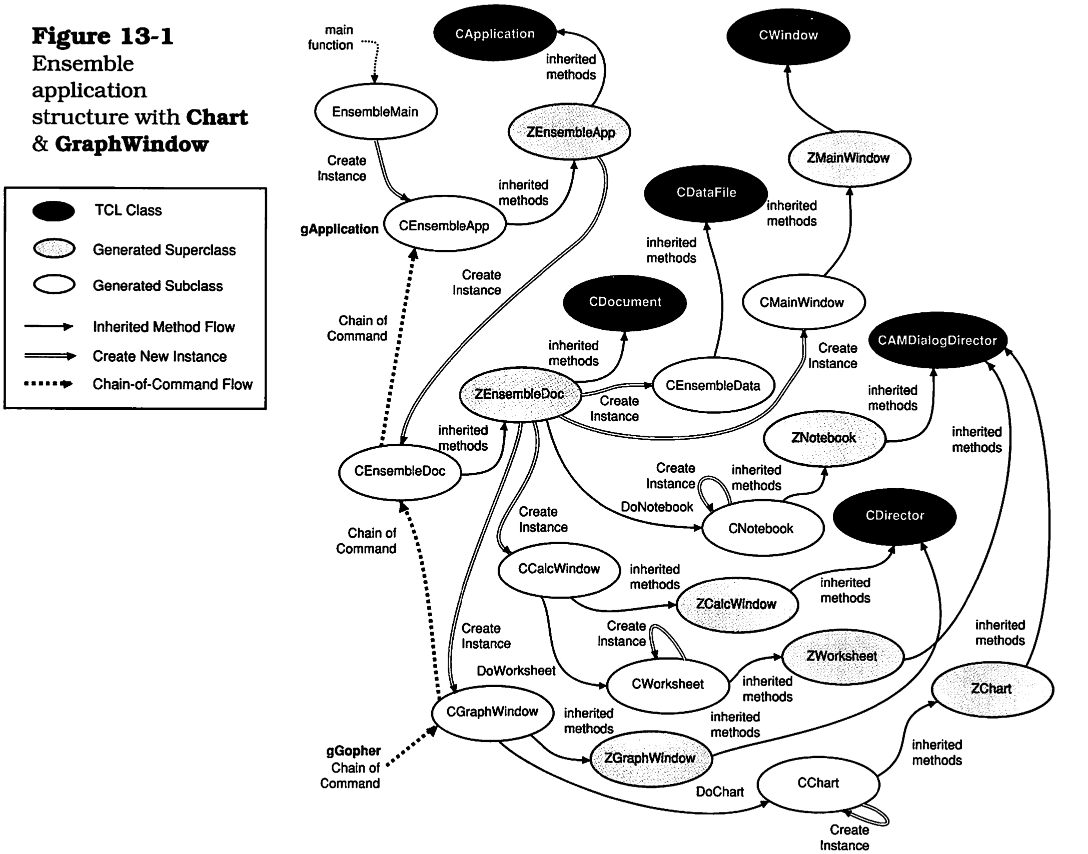
When the new files were generated for the GraphWindow features, all of the superclass source and header files were regenerated as well. The new structure of the application, with the addition of the new window and dialog, is shown in Figure 13-1. This is the final structure of the Ensemble application.

You can see from looking at the figure that all of the windows are created from within the **ZEnsembleDoc** module—specifically, from within the **BuildWindows** method. This method has been modified in the newly generated code to create the **CGraphWindow** instance, which inherits its default appearance from the methods in the **ZGraphWindow** class.

The new version of the **ZEnsembleDoc** module also contains code in its **DoCommand** method to create an instance of the **CChart** dialog. We will leave this code alone, as we did for the code to create the **Worksheet** dialog.

Instead of invoking the dialog, as indicated in its superclass method, we will add code to do so in the **CGraphWindow** module's **DoCommand** method. This method will completely override the other. The diagram in Figure 13-1 shows the **DoChart** function being called from within the **CGraphWindow** module. This is the intended structure of the application, so the figure depicts *what will be*, rather than *what is* at the time AppMaker generates the code for the new additions to the application.

Figure 13-1
Ensemble application structure with **Chart** & **GraphWindow**



Although the application's structure is complete as shown in Figure 13-1, the code is not complete. We will discuss the custom additions to make the charting function fully operational in the next chapter. In a subsequent chapter, we will discuss the addition of code to enable the contents of all three windows to be printed. That feature will round out the functionality of the Ensemble application.

In the sections that follow, we will describe the newly generated code, what features it provides, and what will need to be modified to make it fully functional. The classes and methods that comprise the important functions of this code are shown in Table 13-1. We will be describing each of these in the remainder of the chapter.

Table 13-1
Generated code changes for the **GraphWindow** and **Chart** interface

Class	Method	Description
ZEnsembleDoc	BuildWindows	Creates all window instances
ZGraphWindow	IZGraphWindow	Creates and initializes the **GraphWindow** instance
ZGraphWindow	NewUser4	Method to create a CPanorama
ZGraphWindow	DoCommand	Method meant to be overridden in the subclass
CGraphWindow	NewUser4	Override to create a **User4** instance
CGraphWindow	IGraphWindow	Further window initialization
CGraphWindow	UpdateMenus	Override method in which to place code to enable and disable menu commands
CGraphWindow	DoCommand	Method in which a call to the **DoChart** method will be created
CGraphWindow	ProviderChanged	Override of ancestor method
CUser4	IViewTemp	Initializes **User4** Panorama
CUser4	Draw	Draws graph method
GraphWindow.c global function	DoChart	Creates the **CChart** instance, initializes and runs the **Chart** dialog
ZChart	IZChart	Initializes the **Chart** dialog's user interface elements
ZChart	UpdateMenus	Method to be overridden in **CChart**
CChart	IChart	Additional dialog initialization
CChart	UpdateMenus	Overrides method in **ZChart**
CChart	DoCommand	Handles click commands for dialog items
CChart	ProviderChanged	Handles BroadcastChange messages for text fields in dialog

Newly Generated Code in ZEnsembleDoc

The principal change in the **ZEnsembleDoc** module is the addition of code to create the new **CGraphWindow** instance in the **BuildWindows** method.

BuildWindows Method Code

This method is responsible for creating all of the windows in the application. Each is managed by the **CEnsembleDoc** class, and each has that class as its supervisor. When we want to send a command to the **CEnsembleDoc** instance from within any of the windows, we can do so by referring to **itsSupervisor**, which is an instance variable for all window instances. The new code for the **BuildWindows** method is as follows:

```
void  ZEnsembleDoc::BuildWindows(void)
{
   CWindow*mainWindow;
   CDirector*subWindow;

   mainWindow = new CMainWindow;
   itsWindow = mainWindow;
   ((CMainWindow *)mainWindow)->IMainWindow (this, itsData);
   itsMainPane = ((CMainWindow *)mainWindow)->itsMainPane;
   subWindow = new CCalcWindow;
   ((CCalcWindow *)subWindow)->ICalcWindow (this, itsData);
   subWindow = new CGraphWindow;
   ((CGraphWindow *)subWindow)->IGraphWindow (this, itsData);
}
```

Notice that the new **BuildWindows** method creates instances of all three windows. This is in contrast to the code shown on page 170, in which only the **CMainWindow** and **CCalcWindow** instances are created. The **CCalcWindow** and **CGraphWindow** instances are known as *directors*. They, along with the TCL's **CDocument** class, are immediate subclasses of the **CDirector** class, which is an abstract class that implements a window that can handle commands. A director manages the communication between the application and the window.

Figure 13-1 shows that when the **GraphWindow** is active, the **gGopher** variable points to the **CGraphWindow** class. This will be the first class to receive commands in its **DoCommand** method. If the command is not handled in that method, it is passed up the *chain of command* to the first director able to perform the desired action.

Newly Generated Code in ZGraphWindow

The generated code for the **ZGraphWindow** class consists of three major methods: **IZGraphWindow**, **NewUser4**, and **DoCommand**.

IZGraphWindow Method Code

When the **BuildWindows** method executes, it creates an instance of the **CGraphWindow** class, whose **IGraphWindow** method is called. The first action of the **IGraphWindow** method is to call the inherited **IZGraphWindow** method to create the window. The code for **IZGraphWindow** is as follows:

```
void  ZGraphWindow::IZGraphWindow(CDirectorOwner *aSupervisor)
{
   CView          *enclosure;
   CBureaucrat   *supervisor;
   CSizeBox       *aSizeBox;

   inherited::IDirector (aSupervisor);

   itsWindow = new CWindow;
   itsWindow->IWindow (GraphWindowID, FALSE, gDesktop, this);

   enclosure = itsWindow;
   supervisor = this;
   ScrollPane1 = new CScrollPane;
   ScrollPane1->IViewRes ('ScPn', 139, enclosure, supervisor);
      User4 = NewUser4 ();
      User4->IViewRes ('Pano', 128, ScrollPane1, supervisor);
   ScrollPane1->InstallPanorama (User4);

   aSizeBox = new CSizeBox;
   aSizeBox->ISizeBox (enclosure, supervisor);
}
```

The method first calls its inherited **IDirector** method and then creates a new instance of the **CWindow** class and initializes it. The **itsWindow** instance variable for the **CGraphWindow** class will contain the handle to the new window. Following the creation of the window, the **IZGraphWindow** method installs an instance of the **CScrollPane** class and a new instance of **User4**, which is AppMaker's autonaming convention for items of class **CPanorama**.

NewUser4 Method Code

To create the **CPanorama**, AppMaker generates code to call a method named **NewUser4**, which is as follows:

```
CPanorama*ZGraphWindow::NewUser4(void)
{
   CPanorama    *pane;
   pane = new CPanorama;
   return (pane);
}
```

The **NewUser4** method will be completely overridden by the subclass method of the same name. It is generated into the superclass file so that the version in the subclass can override its functions. Following the creation of the **User4** instance, it is initialized and installed into the scroll pane as the panorama for the **ScrollPane1** instance.

DoCommand Method Code

One other method is generated into the superclass and meant to be overridden. This is the **DoCommand** method, whose code is as follows:

```
void  ZGraphWindow::DoCommand(long theCommand)
{
   switch (theCommand)
   {
      default:
            inherited::DoCommand (theCommand);
         break;
   }
}
```

Newly Generated Code in CGraphWindow

The **CGraphWindow** class is the director for the graph window and contains methods that override and provide additional functionality for other methods in the Ensemble application.

NewUser4 Method Code

The first new method is **NewUser4**, which is an override of the method of the same name in the **ZGraphWindow** class. The code for **NewUser4** is as follows:

```
CPanorama*CGraphWindow::NewUser4(void)
{
   CUser4  *pane;

   pane = new CUser4;
   return (pane);
}
```

The method returns a pane that is a subclass of **CPanorama**. Creating a subclass of **CPanorama** is necessary to override its methods—in particular, the **IViewTemp** and **Draw** methods (discussed later).

IGraphWindow Method Code

In the discussion of the **BuildWindows** method of the **ZEnsembleDoc** class, we mentioned that when the **CGraphWindow** instance was created, its **IGraphWindow** method was called. The code for that method is as follows:

```
void  CGraphWindow::IGraphWindow( CDirector *aSupervisor,
                                  CEnsembleData *theData)
{
   itsData = theData;
   inherited::IZGraphWindow (aSupervisor);
   gDecorator->StaggerWindow (itsWindow);

   // any additional initialization for your window
}
```

The **IGraphWindow** method saves the reference to the **CEnsembleData** instance into its **itsData** instance variable and then calls the superclass **IZGraphWindow** method, which we have previously examined. When that method returns, it will have created the window and all its contents. The **IGraphWindow** method can call the **gDecorator** to size and position the window, staggering it with respect to the other windows on the screen. We will be customizing **IGraphWindow** to perform some additional initialization in the next chapter.

UpdateMenus Method Code

The **CGraphWindow** class also contains an **UpdateMenus** method, which, in its generated form, merely calls the inherited method. We will be modifying that method to enable and disable menu commands when we discuss the custom code changes in the next chapter. The generated code for the **UpdateMenus** method is as follows:

```
void  CGraphWindow::UpdateMenus(void)
{
   inherited::UpdateMenus ();
}
```

DoCommand Method Code

AppMaker also generates a **DoCommand** method to handle any commands sent to the **GraphWindow**, when it is active. The code for this method is as follows:

```
void  CGraphWindow::DoCommand(long theCommand)
{
   switch (theCommand)
   {
      default:
            inherited::DoCommand (theCommand);
         break;
   }
}
```

As generated, the **DoCommand** method merely passes on each command to its inherited method. We will be customizing **DoCommand** to call the **DoChart** function when the **Format Chart** command is selected.

ProviderChanged Method Code

The last method generated into the **CGraphWindow** class is called **ProviderChanged**. The intention of this method is to handle the **ProviderChanged** messages created by the **CBureaucrat** class, in response to a window item that sends a

BroadcastChange message. The **ProviderChanged** method is essentially empty, and we will not be modifying it for the Ensemble application:

```
void  CGraphWindow::ProviderChanged(CCollaborator *aProvider,
                                    long    reason,
                                    void*   info)
{
   // empty method
}
```

Newly Generated Code for CUser4

The **GraphWindow** module also contains two important methods that pertain to the **User4** class. Recall that the **NewUser4** method of the **CGraphWindow** class created an instance of the **User4** class to represent the panorama for the window's scroll pane. The **User4** instance is initialized with a **'Pano'** resource; however, we will be modifying the **IViewTemp** method to set the bounds of the panorama when we discuss the custom graph code modifications in the next chapter.

IViewTemp Method Code

The generated code for the **IViewTemp** method is as follows:

```
void  CUser4::IViewTemp(CView *anEnclosure,
                        CBureaucrat *aSupervisor,
                        Ptr viewData)
{
   inherited::IViewTemp (anEnclosure, aSupervisor, viewData);

   // any additional initialization for your subclass
}
```

Note that the generated code merely calls the inherited **IViewTemp** method.

Draw Method Code

The most important of the **User4** class methods is the **Draw** method. As generated, the method does nothing visible; however, we will be adding a great deal of code to draw the selected graph when the method is called. The generated code for the **Draw** method is as follows:

```
void  CUser4::Draw(Rect  *area)
{
   // replace with your own code, which draws the pane.
   // Note that 'area' is usually ignored; it has no relationship
   // to the size of the pane; it merely indicates what portion
   // (in QuickDraw coordinates) of the pane needs to be drawn

   Rect        theFrame;
   PenState    savePen;

   GetPenState (&savePen);
   PenNormal ();
   FrameToQDR (&frame, &theFrame);

   // place any drawing commands here

   SetPenState (&savePen);
}
```

Notice that AppMaker has included comments that provide instructions regarding this method. The generated code saves the current pen characteristics by calling the toolbox **Get-PenState** routine, resets the pen's characteristics using the **PenNormal** routine, and then converts the bounds of the window's **frame** (which is an instance variable for every pane) from Frame to QuickDraw coordinates, saving the result into the local Rect variable **theFrame**.

Any code to perform drawing functions within the panorama would immediately follow the **FrameToQDR** call. When all drawing operations are complete, the pen state is restored to the initial value by calling the **SetPenState** toolbox routine with the contents of the **savePen** variable.

Newly Generated Code for DoChart

The **DoChart** function is global and accessible to any method in the application. The AppMaker-generated code calls the **DoChart** function from within the **ZEnsembleDoc** class's **Do-Command** method when the **Format Chart** command has been selected. We intend for this command to be recognized only when the **GraphWindow** is active, so we have shown in

Figure 13-1 that the **DoChart** function is called from the **CGraphWindow** module (which will be the case after the custom code modifications are complete). The generated code for the **DoChart** function is as follows:

```
void  DoChart(CDirectorOwner  *aSupervisor)
{
   CChart  *dialog;
   long      dismisser;

   dialog = NULL;
   TRY
   {
      dialog = new CChart;
      dialog->IChart (aSupervisor);

      /* initialize dialog panes */

      dialog->BeginDialog ();
      dismisser = dialog->DoModalDialog (cmdOK);

      if (dismisser == cmdOK)
      {
         /* extract values from dialog panes */
      }

      dialog->Dispose ();
   }
   CATCH
   {
      ForgetObject (dialog);
   }
   ENDTRY;
}
```

The **DoChart** function contains comments that indicate where code should be added to initialize the dialog panes and also where code should be added to extract the final settings as a result of the user's interaction with the dialog.

The first act of the **DoChart** function is to create an instance of **CChart** and then send the **IChart** message to it. The **IChart** method immediately sends the **IZChart** message to perform the creation and initialization of the chart dialog, within the superclass method. The superclass methods are described in the next section.

After the dialog has been initialized, the **DoChart** function calls the **BeginDialog** and **DoModalDialog** methods to display the dialog and cause it to begin accepting user events. When the user clicks one of the *dismisser* buttons, the **DoChart** function tests whether what was clicked was the **OK** button and suggests that code be added to extract the dialog's settings. In any case, the dialog is disposed of and the function returns to its caller.

Newly Generated Code for ZChart

Table 13-1 shows two methods that are generated into the **ZChart** class. Of these, the more important one is the **IZChart** method, which initializes the dialog's fields and controls. In effect, the method *creates* the appearance of the dialog.

IZChart Method Code

The code for this method is as follows:

***IZChart** method (beginning)*

```
void  ZChart::IZChart(CDirectorOwner *aSupervisor)
{
   CView          *enclosure;
   CBureaucrat   *supervisor;
   inherited::IAMDialogDirector (ChartID, aSupervisor);
   enclosure = itsWindow;
   supervisor = itsWindow;

   OKButton = new CAMButton;
   OKButton->IViewRes ('CtlP', 175, enclosure, supervisor);
   CancelButton = new CAMButton;
   CancelButton->IViewRes ('CtlP', 176, enclosure, supervisor);

   ChartTypeGroup = new CLabeledGroup;
   ChartTypeGroup->IViewRes ('LGrp', 129, enclosure, supervisor);

   HorizontalBarRadio = new CAMRadioControl;
   HorizontalBarRadio->IViewRes ('CtlP', 177, ChartTypeGroup,
      ChartTypeGroup);
   VerticalBarRadio = new CAMRadioControl;
   VerticalBarRadio->IViewRes ('CtlP', 178, ChartTypeGroup,
      ChartTypeGroup);
   XYChartRadio = new CAMRadioControl;
   XYChartRadio->IViewRes ('CtlP', 179, ChartTypeGroup,
      ChartTypeGroup);
   ScalingChoicesGroup = new CLabeledGroup;
```

***IZChart** method code (continued)*

```
ScalingChoicesGroup->IViewRes ('LGrp', 130, enclosure,
   supervisor);

hMinLabel = new CAMStaticText;
hMinLabel->IViewRes ('AETx', 152, ScalingChoicesGroup,
   supervisor);
hMaxLabel = new CAMStaticText;
hMaxLabel->IViewRes ('AETx', 153, ScalingChoicesGroup,
   supervisor);
horizMinField = new CAMDialogText;
horizMinField->IViewRes ('ADTx', 142, ScalingChoicesGroup,
   supervisor);
horizMaxField = new CAMDialogText;
horizMaxField->IViewRes ('ADTx', 143, ScalingChoicesGroup,
   supervisor);
vMinLabel = new CAMStaticText;
vMinLabel->IViewRes ('AETx', 154, ScalingChoicesGroup,
   supervisor);
vMaxLabel = new CAMStaticText;
vMaxLabel->IViewRes ('AETx', 155, ScalingChoicesGroup,
supervisor);

vertMinField = new CAMDialogText;
vertMinField->IViewRes ('ADTx', 144, ScalingChoicesGroup,
   supervisor);
vertMaxField = new CAMDialogText;
vertMaxField->IViewRes ('ADTx', 145, ScalingChoicesGroup,
   supervisor);

Group16 = new CRadioGroupPane;
Group16->IViewRes ('Pane', 136, ScalingChoicesGroup, supervisor);
AutomaticScaleRadio = new CAMRadioControl;
AutomaticScaleRadio->IViewRes ('CtlP', 180, Group16, Group16);
ManualScaleRadio = new CAMRadioControl;
ManualScaleRadio->IViewRes ('CtlP', 185, Group16, Group16);

TitleLabel = new CAMStaticText;
TitleLabel->IViewRes ('AETx', 156, enclosure, supervisor);
titleField = new CAMDialogText;
titleField->IViewRes ('ADTx', 146, enclosure, supervisor);

vlabelRangeCheck = new CAMCheckBox;
vlabelRangeCheck->IViewRes ('CtlP', 184, enclosure, supervisor);
hRangeField = new CAMDialogText;
hRangeField->IViewRes ('ADTx', 147, enclosure, supervisor);
hlabelRangeCheck = new CAMCheckBox;
hlabelRangeCheck->IViewRes ('CtlP', 183, enclosure, supervisor);
vRangeField = new CAMDialogText;
vRangeField->IViewRes ('ADTx', 148, enclosure, supervisor);
```

***IZChart** method code (concluded)*

```
    hLabRngField = new CAMDialogText;
    hLabRngField->IViewRes ('ADTx', 149, enclosure, supervisor);
    vLabRngField = new CAMDialogText;
    vLabRngField->IViewRes ('ADTx', 150, enclosure, supervisor);

    HorizontalDataLabel = new CAMStaticText;
    HorizontalDataLabel->IViewRes ('AETx', 157, enclosure, supervisor);
    VerticalDataLabel = new CAMStaticText;
    VerticalDataLabel->IViewRes ('AETx', 158, enclosure, supervisor);
}
```

Each of the dialog elements is created as an instance of an appropriate class and then is initialized from the specified resource generated by AppMaker (e.g., in the last line of the code, the **VerticalDataLabel** field is initialized with an **'AETx'** resource number 158). The field and control names are taken from the labels assigned to them when we designed the dialog box.

When the **IZChart** method is complete, all of the user interface elements have been created and placed into the dialog window. The final step of showing the window and operating the dialog is performed by the **DoChart** function.

UpdateMenus Method Code

The **ZChart** module also contains an **UpdateMenus** method that is intended to be overridden during the course of the application's execution. The generated code for this method is as follows:

```
void  ZChart::UpdateMenus(void)
{
    inherited::UpdateMenus ();
}
```

As is apparent, the **UpdateMenus** method calls its inherited method, thereby passing on the responsibility for handling any menu updates.

Newly Generated Code for CChart

The generated code for the **CChart** class consists of four major methods: **IChart**, **UpdateMenus**, **DoCommand**, and **ProviderChanged**. We will be customizing all of these methods in the next chapter. For now, we show the generated code to which the custom additions will be applied.

IChart Method Code

The **IChart** method overrides and supplements the initialization of the **IZChart** method shown beginning on page 373. The generated code for **IChart** is as follows:

```
void  CChart::IChart (CDirectorOwner *aSupervisor)
{
   inherited::IZChart (aSupervisor);
   // any additional initialization for your dialog
}
```

Although the generated code simply calls the inherited method, we will be supplementing the code with some additional initialization steps.

UpdateMenus Method Code

The generated code for the **UpdateMenus** method needs no modification. It simply passes on the message to its superclass to handle. The code is as follows:

```
void  CChart::UpdateMenus(void)
{
   inherited::UpdateMenus ();
}
```

DoCommand Method Code

The **DoCommand** method is generated to handle the click commands assigned to all the controls in the dialog. We will be modifying the method heavily as generated by AppMaker; however it is as follows:

DoCommand *method code (beginning)*

```
void  CChart::DoCommand(long theCommand)
{
   switch (theCommand) {
      case cmdHorizontalBarRadio:
            /* DoHorizontalBarRadio ();*/
         break;
      case cmdVerticalBarRadio:
```

***DoCommand** method code (concluded)*

```
            /* DoVerticalBarRadio ();*/
        break;
    case cmdXYChartRadio:
            /* DoXYChartRadio ();*/
        break;
    case cmdAutomaticScaleRadio:
            /* DoAutomaticScaleRadio ();*/
        break;
    case cmdManualScaleRadio:
            /* DoManualScaleRadio ();*/
        break;
    case cmdvlabelRangeCheck:
            /* DovlabelRangeCheck ();*/
        break;
    case cmdhlabelRangeCheck:
            /* DohlabelRangeCheck ();*/
        break;
    default:
            inherited::DoCommand (theCommand);
        break;
  }
}
```

The code anticipates that you will be writing methods or functions to handle each of the command cases, so it suggests the naming conventions to use when any of these commands occurs. The **DoCommand** method is invoked when a *click command* (assigned from the resource for each of the controls when it is initialized) is generated. Clicking on a checkbox, for example, will generate a click command that invokes the **DoCommand** method. The TCL will take care of checking and unchecking the checkbox; however, you will have to write the code that performs any special functions that are required as a result of the click. We will show a greatly enhanced version of **DoCommand** in the next chapter.

ProviderChanged Method Code

The final method in the **CChart** module is called **ProviderChanged**. This method is invoked as a result of entries into any of the text fields in the dialog.

The **CAMDialogText** classes send a **BroadcastChange** message when a keystroke changes the contents of a text field. As previously described, the **CBureaucrat** class overrides this method and sends a **ProviderChanged** message to the supervisor of the element making the broadcast. For our charts, the supervisor is the **CChart** class. For any of the dialog's fields, AppMaker generates code to respond to the **Provider-Changed** messages. The generated code is as follows:

ProviderChanged *method code (beginning)*

```
void  CChart::ProviderChanged(CCollaborator  *aProvider,
                              long      reason,
                              void*     info)

{
    if (aProvider == horizMinField) {
        if (horizMinField->GetLength () == 0) {
            // text is empty
        } else {
            // there is some text
        }
    }
    if (aProvider == horizMaxField) {
        if (horizMaxField->GetLength () == 0) {
            // text is empty
        } else {
            // there is some text
        }
    }
    if (aProvider == vertMinField) {
        if (vertMinField->GetLength () == 0) {
            // text is empty
        } else {
            // there is some text
        }
    }
    if (aProvider == vertMaxField) {
        if (vertMaxField->GetLength () == 0) {
            // text is empty
        } else {
            // there is some text
        }
    }
    if (aProvider == titleField) {
        if (titleField->GetLength () == 0) {
            // text is empty
        } else {
            // there is some text
        }
    }
    if (aProvider == hRangeField) {
        if (hRangeField->GetLength () == 0) {
            // text is empty
```

***ProviderChanged* method code (concluded)**

```
        } else {
            // there is some text
        }
    }
    if (aProvider == vRangeField) {
        if (vRangeField->GetLength () == 0) {
            // text is empty
        } else {
            // there is some text
        }
    }
    if (aProvider == hLabRngField) {
        if (hLabRngField->GetLength () == 0) {
            // text is empty
        } else {
            // there is some text
        }
    }
    if (aProvider == vLabRngField) {
        if (vLabRngField->GetLength () == 0) {
            // text is empty
        }
        else
        {
            // there is some text
        }
    }
}
```

AppMaker's generated code tests the provider handle passed to the method against each of the field handles to determine to which field the message pertains. Once the field that has changed is found, the code tests if the field is empty or whether it contains one or more characters of data. We will make significant changes to this method, as described in the custom code modifications in the next chapter.

Exercises

1. Assuming that Figure 13-1 shows the universe of objects that have been created, how many actual object instances exist in the completed application? (Bear in mind that only one dialog object can exist at a given instant.)

2. Examine the **IZChart** method in the **ZChart** class and describe its similarity to the generated code in the **IZNotebook** and **IZWorksheet** methods for their corresponding dialogs. Explain how these similarities are useful in the development of large applications.

3. Describe in what way the **DoCommand** method of the **CChart** class should be customized. Explain your reasoning for each expected custom code addition.

4. Describe the customization that will need to be applied to the **ProviderChanged** method in the **CChart** class. Is it possible to eliminate any of the code blocks generated for the various providers? If so, which ones, and why?

5. Develop a methodology for validating the contents of each of the text fields in the **Chart** dialog. What measures must be taken to ensure that when the user dismisses the dialog, each of the fields contains a valid entry? Describe how you would go about verifying the data ranges entered into the fields that support those entries.

6. Assuming that you have defined a methodology for validating the input data in the **Chart** dialog, how would this be implemented so that it would prevent the dialog from being dismissed when one or more of the fields contain inconsistent or invalid data? (*Hint*: Look into the class hierarchy for the dialog, and see if there is any way to prevent a dialog from being dismissed after the **OK** button has been pressed.)

Chapter 14

Customizing the Graphing Code

This chapter describes the custom code modifications we will make to the modules that implement the **GraphWindow** and **Chart** dialogs. The methods will be separated into two sets, with those pertaining to the **Chart** dialog being presented first.

There are a great number of changes to both the **Chart** and **GraphWindow** modules. Each of these modules has a major role in providing the ability to construct graphs from the existing worksheet data.

The **Chart** module is responsible for running the dialog in which the chart type, data range, label range, and scaling choices are made. Once selected, the choices are retained, so that they can easily be changed.

The **GraphWindow** module is responsible for performing all of the drawing functions. This is accomplished in the panorama's **Draw** method, which gains control when an Update event occurs.

The new, custom code for each of the methods in the two modules is detailed in the sections that follow. Note that because the graphing function works on the existing worksheet data, there are no changes to the file format or any of the I/O methods in the **CEnsembleData** module for this version of the Ensemble application.

The chapter contains a great deal of detailed discussion of the particular algorithms involved in creating horizontal bar, vertical bar, and scatter plot charts. It is important to show the applicable methods and discuss their operation for two reasons: First, the material is germane to charting practices in general, and second, leaving it out would omit important details of the application that we have so carefully constructed and documented.

Customizing the CEnsembleDoc Code

Two new methods have been added to the **CEnsembleDoc** class to facilitate access to the **CCalcWindow** class's methods from other classes.

SetCalcWindow Method Code

The code for this method is as follows:

```
void  CEnsembleDoc::SetCalcWindow(CCalcWindow *theWindow)
{
   itsCalcWindow = theWindow;
}
```

The **SetCalcWindow** method is called by the **ICalcWindow** method in the **CCalcWindow** class when that instance is initialized. The window is saved in a new instance variable called **itsCalcWindow** in the **CEnsembleDoc** class.

GetCalcWindow Method Code

The code for this method is as follows:

```
CCalcWindow *CEnsembleDoc::GetCalcWindow(void)
{
   return itsCalcWindow;
}
```

The **GetCalcWindow** method is called by the various charting methods in the **GraphWindow** module. These methods require access to the worksheet window, so that the window's new access methods can be referenced.

Customizing the CCalcWindow Code

In order to support the inquiries of charting methods in the **CGraphWindow** module, two new methods have been added to the **CCalcWindow** class.

GetValueString Method Code

This method accepts a cell number and returns the entry string associated with that cell if the cell has been defined in the worksheet. If it has not been thus defined, the method returns an empty string:

```
void  CCalcWindow::GetValueString (Cell aCell, StringPtr  aString)
{
   long         param;
   CWSEntry  *anEntry;

   param = *(long *)&aCell;
   if((anEntry = (CWSEntry *)wsCluster->FindItem1 (FindWSCell,
      param)) != NULL)
   {
      anEntry->GetWSText(aString);
   }
   else
   {
      CopyPString("\p", aString);
   }
}
```

GetValueValue Method Code

The **GetValueValue** method accepts a cell number and returns the double-precision floating-point value associated with the cell if the cell has been defined in the worksheet. If it has not been thus defined, the method returns a value of **0.0**. The code for the method is as follows:

```
void CCalcWindow::GetValueValue (Cell aCell, double *aValue)
{
   long  param;
   CWSEntry *anEntry;

   param = *(long *)&aCell;
   if((anEntry = (CWSEntry *)wsCluster->FindItem1 (FindWSCell,
      param)) != NULL)
   {
      *aValue = anEntry->GetWSValue();
   }
   else
   {
      *aValue = 0.0;
   }
}
```

Both the **GetValueString** and **GetValueValue** methods require a cell as their input value. The horizontal component of the cell identifies the worksheet column, and the vertical component identifies the worksheet row.

The charting methods (described later in this chapter) must access the worksheet values and strings for two reasons: The values are used to determine the range of data and individual data values to be charted, while the strings are used to provide annotation of the charts with worksheet labels. In both cases, the **CWSEntry** class's methods are used to access the corresponding fields in the worksheet entry if it is found to be defined.

Customizing the Format Chart Dialog

In order for a graph to be drawn, the user must specify its type and at least the range of data to be graphed. The specification of the graph type and its parameters is performed in the **Format Chart** dialog. The custom code modifications to implement the full functionality of this dialog are listed in Table 14-1. The single new method, **Validate**, is shown in boldface type.

Table 14-1
Customized methods to implement the **Format Chart** dialog

Class	Method	Description
Chart.c global function	DoChart	Function to create and manage the chart dialog
CChart	IChart	Initializes view IDs
CChart	DoCommand	Handles click commands
CChart	ProviderChanged	Handles changes in text fields
CChart	**Validate**	Validates dialog entries

Customizing the DoChart Code

The global **DoChart** function is called from within the **DoCommand** method of the **GraphWindow** module when the user selects the **Chart** command from the **Format** menu. The function is responsible for creating and managing the **Chart** dialog, and it does so by creating an instance of the **CChart** class, initializing the instance, running the dialog, and then extracting the modified data when the user clicks on the **OK** button. If the **Cancel** button is clicked, the function does not update the saved dialog settings. The original code for this function was presented in the previous chapter (see page 372). The modifications to the code are substantial. The original dialog is quite busy, with a lot of different fields and selections. One purpose of the modified code is to hide portions of the dialog, as it is first presented to the user, so that only the previous settings are visible and active. The initial settings for the dialog are established by the **IGraphWindow** method in the **GraphWindow** module. The custom code for the **DoChart** function is as follows:

DoChart *function code (beginning)*

```
void  DoChart(CDirectorOwner  *aSupervisor)
{
   CChart       *dialog;
   long         dismisser;
   chartInfo    aSetting;

   dialog = NULL;
   TRY
   {
      dialog = new CChart;
      dialog->IChart (aSupervisor);

      //
      // initialize the dialog with its last (or first) settings
      //
      dialog->theChartInfo = ((CGraphWindow *)aSupervisor)
         ->GetChartInfo();
      dialog->theSettings = dialog->theChartInfo->GetChartInfo();
      dialog->theSettings.modified = 0;
      aSetting = dialog->theSettings;
      dialog->Group18->SetStationID(aSetting.scalingType);
      dialog->horizMinField->SetTextString(aSetting.hMinScale);
      dialog->horizMaxField->SetTextString(aSetting.hMaxScale);
      dialog->vertMinField->SetTextString(aSetting.vMinScale);
      dialog->vertMaxField->SetTextString(aSetting.vMaxScale);
      if(aSetting.scalingType == cAutomaticScaleViewID)
      {
         dialog->horizMinField->Hide();
         dialog->horizMaxField->Hide();
         dialog->vertMinField->Hide();
         dialog->vertMaxField->Hide();
```

DoChart *function code (continued)*

```
}
switch(aSetting.chartType)
{
   case cHorizontalBarViewID:
   {
      // deactivate the horizontal stuff
      dialog->HorizontalDataLabel->Hide();
      dialog->hlabelRangeCheck->Deactivate();
      dialog->hLabRngField->Hide();
      dialog->hRangeField->Hide();
      dialog->vertMinField->Hide();
      dialog->vertMaxField->Hide();
      if(aSetting.vLabelCheck == 0)
      {
         dialog->vLabRngField->Hide();
      }
      break;
   }
   case cVerticalBarViewID:
   {
      // deactivate the vertical stuff
      dialog->VerticalDataLabel->Hide();
      dialog->vlabelRangeCheck->Deactivate();
      dialog->vLabRngField->Hide();
      dialog->vRangeField->Hide();
      dialog->horizMinField->Hide();
      dialog->horizMaxField->Hide();
      if(aSetting.hLabelCheck == 0)
         dialog->hLabRngField->Hide();
      break;
   }
}
dialog->ChartTypeGroup->SetStationID(aSetting.chartType);
dialog->titleField->SetTextString(aSetting.title);
dialog->hRangeField->SetTextString(aSetting.hDataRange);
dialog->vRangeField->SetTextString(aSetting.vDataRange);
dialog->hLabRngField->SetTextString(aSetting.hLabelRange);
dialog->hlabelRangeCheck->SetValue(aSetting.hLabelCheck);
dialog->vLabRngField->SetTextString(aSetting.vLabelRange);
dialog->vlabelRangeCheck->SetValue(aSetting.vLabelCheck);

//
// start running the dialog
//
dialog->BeginDialog ();
dismisser = dialog->DoModalDialog (cmdOK);
if (dismisser == cmdOK)
{
   /* extract values from dialog panes */
```

DoChart
function code
(concluded)

```
            aSetting = dialog->theSettings;
            aSetting.modified = 1;
            dialog->theChartInfo->SetChartInfo(aSetting);
        }
        dialog->Dispose ();
    }
    CATCH
    {
        ForgetObject (dialog);
    }
    ENDTRY;
}
```

The **DoChart** function begins by creating an instance of the **CChart** class. Then it calls the **IChart** method to initialize the instance. As was shown in the last chapter, the **IChart** method calls the **IZChart** method to create and initialize each of the user interface items in the dialog. When control returns to the **DoChart** function, the dialog has been created and initialized. The next series of steps in the function are responsible for setting the state of all the controls and fields to their previous values.

We have created a new module that defines a class called **CChartInfo** which contains methods to initialize an object of that class and access its contents. This class is described later, in conjunction with the **GraphWindow** module methods; however, it is useful for purposes of understanding the **DoChart** initialization code. Accordingly, we present the structure and contents of the data element that contains the settings that this code references. The settings are kept in a structure whose type is defined as **chartinfo**. This structure contains all of the information needed to reconstruct the settings in the **Chart** dialog. The **chartinfo** structure is as follows:

***chartinfo** structure*
declaration
(beginning)

```
typedef   struct
{
    short     modified;
    short     chartType;
    short     scalingType;
    Str9      hMinScale;
    Str9      hMaxScale;
    Str9      vMinScale;
    Str9      vMaxScale;
    Str255    title;
```

***chartInfo** structure declaration (concluded)*

```
    Str9    hDataRange;
    Str9    vDataRange;
    short   hLabelCheck;
    Str9    hLabelRange;
    short   vLabelCheck;
    Str9    vLabelRange;
} chartInfo;
```

The very first field in the structure is used to keep track of whether the control settings or fields were modified by the user. The field is set to 0 at the beginning of the **DoChart** function, and its value is set to 1 if the user clicks the **OK** button—which carries the implicit assumption that something has been modified.

Each of the controls has a short integer that contains its saved state. Each of the dialog fields is contained in a string variable of an appropriate size. The access methods provided with the **CChartInfo** class translate the string variable values to appropriate corresponding binary values when called upon to do so.

The **DoChart** function sets the appropriate scaling selection, loads the horizontal and vertical scale settings into their corresponding fields, and then checks whether automatic scaling was selected. If so, it hides the corresponding minimum and maximum scale values.

After handling the appearance of the scaling section of the dialog, the **DoChart** function checks what type of chart has been selected. Depending on whether a horizontal bar, vertical bar, or X-Y chart was selected, the function hides the irrelevant data entry fields.

Regardless of whether the data entry fields are hidden or visible, the **DoChart** function loads these fields with their previous values. Once that is complete, the **BeginDialog** and **DoModalDialog** methods are executed. At this point, the user is in full control of the dialog. If the chart type is changed, by clicking one of the other chart type buttons, a click command is sent to the **DoCommand** method to signal that fact. Similarly, if the user keys a new value into one of the text fields, that fact is noted by sending a message to the **ProviderChanged** method. These methods will be described shortly.

After the user dismisses the dialog, if the **OK** button was clicked, the current settings that have been updated locally with each change are saved by calling the **SetChartInfo** access method. If **Cancel** was clicked, all changes are discarded. In any case, the dialog is disposed of.

Customizing the CChart Code

The **CChart** class contains several methods that require changes to implement the full functionality of the **Format Chart** command. These methods are listed in Table 14-1. Each is discussed in the subsections that follow.

IChart Method Code

The **IChart** method is called by the **DoChart** function to initialize the **Chart** dialog. The first action of the method is to call the inherited **IZChart** method to create and initialize all the user interface elements in the dialog.

The code for the **IZChart** method is shown beginning on page 373. After the **IZChart** method returns, we need to assign a "view ID" to each of the radio buttons in the dialog, so that we can refer to these in the other methods. The code for the **IChart** method is as follows:

```
void  CChart::IChart(CDirectorOwner  *aSupervisor)
{
   inherited::IZChart (aSupervisor);

   //
   // initialize the ViewIDs
   //
   HorizontalBarRadio->ID = cHorizontalBarViewID;
   VerticalBarRadio->ID = cVerticalBarViewID;
   XYChartRadio->ID = cXYChartViewID;

   AutomaticScaleRadio->ID = cAutomaticScaleViewID;
   ManualScaleRadio->ID = cManualScaleViewID;

   TEAutoView (TRUE, titleField->macTE);
}
```

The values for the **viewID** instance variables (common to all descendants of the **CView** class) were arbitrarily chosen to be the same as each radio button's resource ID. The declarations for the values associated with the view ID mnemonics are as follows:

```
enum
{
   cHorizontalBarViewID = 177,
   cVerticalBarViewID,
   cXYChartViewID
};

enum
{
   cAutomaticScaleViewID = 180,
   cManualScaleViewID = 185
};
```

Assigning a view ID to each radio button allows us to refer to the button by name and also allows us to store its value into the settings structure. When the **DoChart** function is called again, the view ID can be used to initialize the dialog with the appropriate button selected.

DoCommand Method Code

The **DoCommand** method is responsible for responding to click commands associated with each of the radio button and checkbox controls. When the user clicks one of these buttons, a **DoCommand** message, identifying the control, is sent to the **CChart** class. The **DoCommand** code for our revised method is rather lengthy:

DoCommand *method code (beginning)*

```
void  CChart::DoCommand(long theCommand)
{
   switch (theCommand)
   {
      case cmdHorizontalBarRadio:
      {
         theSettings.chartType = cHorizontalBarViewID;
         VerticalDataLabel->Show();
         vRangeField->Show();
         vlabelRangeCheck->Activate();
         HorizontalDataLabel->Hide();
         hRangeField->Hide();
         hlabelRangeCheck->Deactivate();
         hLabRngField->Hide();
```

DoCommand
method code
(continued)

```
        if(theSettings.scalingType == cManualScaleViewID)
        {
            vertMinField->Hide();
            vertMaxField->Hide();
            horizMinField->Show();
            horizMaxField->Show();
        }
        if(theSettings.vLabelCheck == 0)
        {
            vLabRngField->Hide();
        }
        else
        {
            vLabRngField->Show();
        }
        break;
    }
    case cmdVerticalBarRadio:
    {
        theSettings.chartType = cVerticalBarViewID;
        HorizontalDataLabel->Show();
        hRangeField->Show();
        hlabelRangeCheck->Activate();
        VerticalDataLabel->Hide();
        vRangeField->Hide();
        vlabelRangeCheck->Deactivate();
        vLabRngField->Hide();
        if(theSettings.scalingType == cManualScaleViewID)
        {
            horizMinField->Hide();
            horizMaxField->Hide();
            vertMinField->Show();
            vertMaxField->Show();
        }
        if(theSettings.hLabelCheck == 0)
        {
            hLabRngField->Hide();
        }
        else
        {
            hLabRngField->Show();
        }
        break;
    }
    case cmdXYChartRadio:
    {
        theSettings.chartType = cXYChartViewID;
        VerticalDataLabel->Show();
        vRangeField->Show();
```

DoCommand
method code
(continued)

```
        vlabelRangeCheck->Activate();
        if(theSettings.vLabelCheck == 0)
        {
            vLabRngField->Hide();
        }
        else
        {
            vLabRngField->Show();
        }
        HorizontalDataLabel->Show();
        hRangeField->Show();
        hlabelRangeCheck->Activate();
        if(theSettings.hLabelCheck == 0)
        {
            hLabRngField->Hide();
        }
        else
        {
            hLabRngField->Show();
        }
        if(theSettings.scalingType == cManualScaleViewID)
        {
            horizMinField->Show();
            horizMaxField->Show();
            vertMinField->Show();
            vertMaxField->Show();
        }
        break;
    }
    case cmdAutomaticScaleRadio:
    {
        theSettings.scalingType = cAutomaticScaleViewID;
        horizMinField->Hide();
        horizMaxField->Hide();
        vertMinField->Hide();
        vertMaxField->Hide();
        break;
    }
    case cmdManualScaleRadio:
    {
        theSettings.scalingType = cManualScaleViewID;
        switch(theSettings.chartType)
        {
            case cHorizontalBarViewID:
            {
                vertMinField->Hide();
                vertMaxField->Hide();
                horizMinField->Show();
                horizMaxField->Show();
```

DoCommand
method code (concluded)

```
                break;
            }
            case cVerticalBarViewID:
            {
                horizMinField->Hide();
                horizMaxField->Hide();
                vertMinField->Show();
                vertMaxField->Show();
                break;
            }
            default:
            {
                horizMinField->Show();
                horizMaxField->Show();
                vertMinField->Show();
                vertMaxField->Show();
                break;
            }
        }
        break;
    }
    case cmdvlabelRangeCheck:
    {
        theSettings.vLabelCheck = vlabelRangeCheck->GetValue();
        if(theSettings.vLabelCheck == 0)
            vLabRngField->Hide();
        else
            vLabRngField->Show();
        break;
    }
    case cmdhlabelRangeCheck:
    {
        theSettings.hLabelCheck = hlabelRangeCheck->GetValue();
        if(theSettings.hLabelCheck == 0)
        {
            hLabRngField->Hide();
        }
        else
        {
            hLabRngField->Show();
        }
        break;
    }
    default:
    {
        inherited::DoCommand (theCommand);
        break;
    }
  }
}
```

The code for the **DoCommand** method consists of a big **switch** statement that has a case for each of the controls in the dialog. When the user clicks one of the radio button controls, we need to change the appearance of the dialog to reflect the requirements of the new selection. For example, if the **Horizontal Bar** chart radio button is clicked, we need to show the vertical data range and vertical label fields, but hide the corresponding horizontal data range and label fields. In addition, if manual scaling is selected, we need to hide the vertical minimum and maximum scale fields. Each of the radio buttons and checkboxes has an associated click command and code in the **DoCommand** method to modify the appearance of the dialog box according to the selection.

ProviderChanged Method Code

The **ProviderChanged** method is invoked by the **CBureaucrat** class, in response to a **BroadcastChange** message from one of the dialog's text entry fields. When the user enters, changes, or deletes text in any of the text fields, the TCL's **CEditText**, **CDialogText**, and **CAbstractText** classes contain methods that send **BroadcastChange** messages. The **CBureaucrat** class's **BroadcastChange** method, in addition to passing on the message to the **CCollaborator** class, sends a **ProviderChanged** message to the supervisor of the instance that sent the original message. In the case of the **Chart** dialog, the supervisor is the **CChart** class. The modified code for the **ProviderChanged** method is as follows:

ProviderChanged *method code (beginning)*

```
void  CChart::ProviderChanged(CCollaborator  *aProvider,
                               long      reason,
                               void*     info)

{
   if (aProvider == horizMinField)
   {
      horizMinField->GetTextString(theSettings.hMinScale);
   }
   if (aProvider == horizMaxField)
   {
      horizMaxField->GetTextString(theSettings.hMaxScale);
   }
   if (aProvider == vertMinField)
   {
      vertMinField->GetTextString(theSettings.vMinScale);
   }
```

***ProviderChanged* method code (concluded)**

```
    if (aProvider == vertMaxField)
    {
        vertMaxField->GetTextString(theSettings.vMaxScale);
    }
    if (aProvider == titleField)
    {
        titleField->GetTextString(theSettings.title);
    }
    if (aProvider == hRangeField)
    {
        hRangeField->GetTextString(theSettings.hDataRange);
    }
    if (aProvider == vRangeField)
    {
        vRangeField->GetTextString(theSettings.vDataRange);
    }
    if (aProvider == hLabRngField)
    {
        hLabRngField->GetTextString(theSettings.hLabelRange);
    }
    if (aProvider == vLabRngField)
    {
        vLabRngField->GetTextString(theSettings.vLabelRange);
    }
}
```

The **ProviderChanged** method tests the identity of the **aProvider** argument. For each of the text fields, when the associated provider is matched, the current contents of the field are saved into the appropriate field of the **theSettings** structure (an instance variable of the **CChart** class). In this way, the current settings are constantly updated with the latest contents of all the fields.

Validate Method Code

When the user clicks the **OK** button, the TCL calls an initially empty **Validate** method, which can return a TRUE or FALSE response. If the response is TRUE, the TCL allows the dialog to stop running and returns control to the **DoChart** function, immediately after the **DoModalDialog** statement. If the response is FALSE, the dialog is not dismissed. While the default method always returns TRUE, it is possible to provide an override method that performs custom validation of the fields and settings in the dialog. This is what we have done in the case of the **Chart** dialog. The custom code for the **Validate** override method is quite lengthy and is as follows:

Validate *method code (beginning)*

```
Boolean CChart::Validate (void)
{
    minMax      aScale;
    Rect        aRange1;
    Rect        aRange2;
    CChartInfo  *aDummy;
    Boolean     result;
    short       v1Length, h1Length;
    short       v2Length, h2Length;

    //
    // validate the fields of the Format Chart dialog
    // to ensure consistency with the worksheet and
    // avoid problems later when we build the chart.
    //
    TRY
    {
        result = TRUE;
        aDummy = new CChartInfo;
        aDummy->IChartInfo();
        aDummy->SetChartInfo (theSettings);
        switch (theSettings.chartType)
        {
            case cHorizontalBarViewID:
            {
                aRange1 = aDummy->GetVData();
                if(aRange1.left < 0)
                {
                    //
                    // invalid data range
                    //
                    result = FALSE;
                }
                if(aRange1.left == aRange1.right)
                {
                    v1Length = aRange1.bottom - aRange1.top;
                }
                else if(aRange1.bottom == aRange1.top)
                {
                    v1Length = aRange1.right - aRange1.left;
                }
                else
                {
                    v1Length = -1;
                }
                if(v1Length <= 0)
                {
```

Validate
method code
(continued)

```
            result = FALSE;
        }
        if(theSettings.vLabelCheck != 0)
        {
            aRange1 = aDummy->GetVLabel();
            if(aRange1.left < 0)
            {
                //
                // invalid label range
                //
                result = FALSE;
            }
            if(aRange1.left == aRange1.right)
            {
                v2Length = aRange1.bottom - aRange1.top;
            }
            else if(aRange1.bottom == aRange1.top)
            {
                v2Length = aRange1.right - aRange1.left;
            }
            else
            {
                v2Length = -1;
            }
            if(v2Length <= 0 || v2Length != v1Length)
            {
                result = FALSE;
            }
        }
        if(theSettings.scalingType == cManualScaleViewID)
        {
            aScale = aDummy->GetHScale();
            if (aScale.max <= aScale.min)
            {
                //
                // invalid scale
                //
                result = FALSE;
            }
        }
        break;
    }
    case cVerticalBarViewID:
    {
        aRange1 = aDummy->GetHData();
        if(aRange1.left < 0)
        {
            //
            // invalid data range
```

Validate *method code (continued)*

```
        //
        result = FALSE;
    }
    if(aRange1.left == aRange1.right)
    {
        h1Length = aRange1.bottom - aRange1.top;
    }
    else if(aRange1.bottom == aRange1.top)
    {
        h1Length = aRange1.right - aRange1.left;
    }
    else
    {
        h1Length = -1;
    }
    if(h1Length <= 0)
    {
        result = FALSE;
    }
    if(theSettings.hLabelCheck != 0)
    {
        aRange1 = aDummy->GetHLabel();
        if(aRange1.left < 0)
        {
            //
            // invalid label range
            //
            result = FALSE;
        }
        if(aRange1.left == aRange1.right)
        {
            h2Length = aRange1.bottom - aRange1.top;
        }
        else if(aRange1.bottom == aRange1.top)
        {
            h2Length = aRange1.right - aRange1.left;
        }
        else
        {
            h2Length = -1;
        }
        if(h2Length <= 0 || h2Length != h1Length)
        {
            result = FALSE;
        }
    }
    if(theSettings.scalingType == cManualScaleViewID)
    {
        aScale = aDummy->GetVScale();
```

Validate
method code
(continued)

```
            if (aScale.max <= aScale.min)
            {
                //
                // invalid scale
                //
                result = FALSE;
            }
        }
        break;
    }
    case cXYChartViewID:
    {
        aRange1 = aDummy->GetHData();
        if(aRange1.left < 0)
        {
            //
            // invalid data range
            //
            result = FALSE;
            break;
        }
        if(aRange1.left == aRange1.right)
        {
            v1Length = aRange1.bottom - aRange1.top;
        }
        else if(aRange1.bottom == aRange1.top)
        {
            v1Length = aRange1.right - aRange1.left;
        }
        else
        {
            v1Length = -1;
        }
        if(v1Length < 0)
        {
            result = FALSE;
            break;
        }
        if(theSettings.hLabelCheck != 0)
        {
            aRange1 = aDummy->GetHLabel();
            if(aRange1.left < 0)
            {
                //
                // invalid label range
                //
                result = FALSE;
            }
            if(aRange1.left == aRange1.right)
```

Validate *method code (continued)*

```
        {
            v2Length = aRange1.bottom - aRange1.top;
        }
        else if(aRange1.bottom == aRange1.top)
        {
            v2Length = aRange1.right - aRange1.left;
        }
        else
        {
            v2Length = -1;
        }
        if(v2Length < 0)
        {
            result = FALSE;
        }
    }
    if(theSettings.scalingType == cManualScaleViewID)
    {
        aScale = aDummy->GetVScale();
        if (aScale.max <= aScale.min)
        {
            //
            // invalid scale
            //
            result = FALSE;
        }
    }
    aRange2 = aDummy->GetVData();
    if(aRange2.left < 0)
    {
        //
        // invalid data range
        //
        result = FALSE;
    }
    if(aRange2.left == aRange2.right)
    {
        h1Length = aRange2.bottom - aRange2.top;
    }
    else if(aRange2.bottom == aRange2.top)
    {
        h1Length = aRange2.right - aRange2.left;
    }
    else
    {
        h1Length = -1;
    }
    if(h1Length < 0)
    {
```

Validate *method code (continued)*

```
        result = FALSE;
    }
    if(theSettings.vLabelCheck != 0)
    {
        aRange2 = aDummy->GetVLabel();
        if(aRange2.left < 0)
        {
            //
            // invalid label range
            //
            result = FALSE;
        }
        if(aRange2.left == aRange2.right)
        {
            h2Length = aRange2.bottom - aRange2.top;
        }
        else if(aRange2.bottom == aRange2.top)
        {
            h2Length = aRange2.right - aRange2.left;
        }
        else
        {
            h2Length = -1;
        }
        if(h2Length < 0)
        {
            result = FALSE;
        }
    }
    if(theSettings.scalingType == cManualScaleViewID)
    {
        aScale = aDummy->GetHScale();
        if (aScale.max <= aScale.min)
        {
            //
            // invalid scale
            //
            result = FALSE;
        }
    }
    if(v1Length != h1Length)
    {
        //
        // must have same number of X-Y points
        //
        result = FALSE;
    }
    break;
}
```

Validate *method code (concluded)*

```
            default:
            {
               result = FALSE;
            }
         }
      }
      CATCH
      {
         ForgetObject (aDummy);
         result = FALSE;
      }
      ENDTRY;

      if(result == FALSE)
      {
         SysBeep(30);
      }
      aDummy->Dispose();
      return result;
}
```

The code in the **Validate** method is broken up into cases that apply to each of the potential chart types. The switch statement jumps to the validation code for the case that is associated with the currently selected chart type. In the code for each case, each of the fields that participate in setting parameters for the chart is tested to ensure that it is valid. In addition, each is tested to ensure that it is consistent with other related parameters for that chart type. If any of the tests fail, the method plays the selected "system beep" sound and returns FALSE to the TCL. A TRUE result is returned only if all the validation checks are successful. Basically, the validation consists of the following steps:

1. A dummy instance of the **CChartInfo** object is created, and the current settings are placed into its instance variable. (This is done so that we can use the access methods in the **CChartInfo** object to convert the string variables to cell ranges and binary numeric values.)

2. If the chart type is a horizontal or vertical bar chart, **Validate** calls the **GetVData** or **GetHData** access method to acquire the worksheet's cell range for the chart data. If the range was specified improperly, or if the ending cell has a value that is not greater than the value of the beginning cell, a FALSE result is returned. The validation proce-

dure continues by determining whether a label range has been specified. If so, the **GetVLabel** or **GetHLabel** access method is called to return the worksheet cell range occupied by the labels. If an invalid range was specified, then if the number of label cells does not equal the number of data cells, a FALSE result is returned. Finally, if the manual scaling option is chosen, the **GetHScale** or **GetVScale** access method is called to return the appropriate minimum and maximum values. If these are invalid, a FALSE result is returned. Only if all of these checks are successful is TRUE returned.

3. If the chart type is an X-Y chart, the checks for both horizontal and vertical cell ranges, label ranges, and scaling are performed. Only if all the values are valid and consistent with one another is a TRUE result returned to the TCL.

Although the code for the **Validate** method is rather long, it is quite straightforward and easy to follow.

Customizing the GraphWindow Code

This section describes the custom code modifications made to the routines in the **GraphWindow** module. The methods for which new custom code has been written are shown in boldface type in Table 14-2. Methods which have only been modified are shown in plain type in the figure.

Customizing the CGraphWindow Methods

Table 14-2 shows a number of methods in the **CGraphWindow** class that have been modified to complete the functionality of the graphing features in the Ensemble application. In general, each of these methods is concerned only with the window and its commands, although two new access methods have been provided.

IGraphWindow Method Code

The **IGraphWindow** method is responsible for calling the **IZGraphWindow** method in the superclass to create and install the window, its scroll pane, and its panorama and then

Table 14-2
Customized methods in the **GraphWindow** module to implement the selected chart types

Class	Method	Description
CGraphWindow	IGraphWindow	Initializes GraphWindow
CGraphWindow	UpdateMenus	Enables and disables menu items
CGraphWindow	DoCommand	Handles menu commands
CGraphWindow	**GetCalcWindow**	Access method for worksheet
CGraphWindow	**GetChartInfo**	Access method for chart settings
CUser4	IViewTemp	Initializes drawing panorama
CUser4	Draw	Draws panorama contents
CUser4	**DrawHBarChart**	Draws horizontal bar chart
CUser4	**DrawVBarChart**	Draws vertical bar chart
CUser4	**DrawXYChart**	Draws X-Y chart
CUser4	**GetBarThickness**	Determines optimum bar size
CUser4	**GetLabelMax**	Determines width of longest label
CUser4	**GetDataMinMax**	Determines minimum and maximum data values for chart
CUser4	**DrawChartFrame**	Draws frame for chart
CUser4	**DrawHorizTicks**	Draws horizontal axis annotation
CUser4	**DrawVertTicks**	Draws vertical axis annotation
CUser4	**GetFormat**	Determines annotation format
global	**log10x**, **exp10x**, **RoundDown**, **RoundUp**, **lookUp**, **lookDown**	Miscellaneous global functions to support the calculation of automatic scaling and selection of appropriate axis annotation values.

perform any additional initialization. The code for **IGraphWindow** is as follows:

IGraphWindow *method code (beginning)*

```
void  CGraphWindow::IGraphWindow(CDirector    *aSupervisor,
                                 CEnsembleData    *theData)
{
   Str255   theFilename;

   itsData = theData;
```

IGraphWindow *method code (concluded)*

```
    inherited::IZGraphWindow (aSupervisor);
    gDecorator->StaggerWindow (itsWindow);

    //
    // create a new CChartInfo instance
    //
    itsChartInfo = new CChartInfo;
    itsChartInfo->IChartInfo();

    //
    // Put the window's name in the title
    //
    if(((CEnsembleDoc *) aSupervisor)->itsFile != NULL)
    {
        ((CEnsembleDoc *) aSupervisor)->itsFile->GetName(theFilename);
        itsWindow->SetTitle(theFilename);
    }
}
```

After control returns from the superclass's **IZGraphWindow** method, the **IGraphWindow** method sends the **gDecorator** a message to stagger the window with respect to the others on the screen. Then **IGraphWindow** creates a new instance of the **CChartInfo** class to hold the chart settings and writes the title of the existing document (if there is any) into the window's title bar.

UpdateMenus Method Code

The **UpdateMenus** method has been revised to enable only the **Chart** command in the **Format** menu when the **GraphWindow** is frontmost on the screen. The code for the updated version of **UpdateMenus** is as follows:

```
void  CGraphWindow::UpdateMenus(void)
{
    inherited::UpdateMenus ();
    //
    // disable Close, Format Notebook, Format Worksheet, enable
    // Format Chart if GraphWindow is the frontmost window
    //
    gBartender->DisableCmd (cmdClose);
    gBartender->DisableCmd (cmdNotebook);
    gBartender->DisableCmd (cmdWorksheet);
    gBartender->EnableCmd (cmdChart);
}
```

Note that in addition to disabling the **Notebook** and **Worksheet** commands, the method also disables the **Close** command in the **File** menu. The windows can be closed only when the **Notebook** window is frontmost on the screen.

DoCommand Method Code

The primary addition to the **DoCommand** method is the code that recognizes the **Chart** command.

Although the **Chart** command is recognized in the **zEnsembleDoc** class's re-generated **DoCommand** method, by intercepting it in the **CGraphWindow** class, we can respond to it first, and allow it only to be enabled in the **Format** menu when the **CGraphWindow** is frontmost on the screen.

The revised code for the **DoCommand** method is as follows:

```
void   CGraphWindow::DoCommand(long theCommand)
{
   switch (theCommand)
   {
      case cmdChart:
      {
         DoChart(this);
         if(itsChartInfo->chartSettings.modified)
         {
            User4->Refresh();
         }
         break;
      }
      default:
      {
         inherited::DoCommand (theCommand);
         break;
      }
   }
}
```

Any command other than **cmdChart** will be handled by the default case, which calls the inherited **DoCommand** method.

GetCalcWindow Method Code

The **GetCalcWindow** method is responsible for providing access to the worksheet from other methods. It is called from the various chart-drawing methods in the **CUser4** class. The method makes use of the fact that the **CEnsembleDoc** class is the supervisor of the **CGraphWindow** class, which calls the document's **GetCalcWindow** method to retrieve the handle to the worksheet instance. The code for **GetCalcWindow** is as follows:

```
CCalcWindow *CGraphWindow::GetCalcWindow()
{
   return ((CEnsembleDoc *) itsSupervisor)->GetCalcWindow();
}
```

GetChartInfo Method Code

When the **CChartInfo** instance was created in the **IGraph-Window** method, the initialization message sent to that instance created the default settings for the chart. When the **DoChart** function is called by the **DoCommand** method, it calls the **GetChartInfo** method to retrieve the handle to the current **CChartInfo** instance. The code for the **GetChartInfo** access method is as follows:

```
CChartInfo *CGraphWindow::GetChartInfo(void)
{
   return itsChartInfo;
}
```

Customizing the CUser4 Methods

All of the drawing operations for the **GraphWindow** take place with respect to its panorama element, for which App-Maker has created a new class named **CUser4**. The new and customized methods in that class are covered in this section.

IViewTemp Method Code

The modified code for the **IviewTemp** method is as follows:

IViewTemp *method code (beginning)*

```
void  CUser4::IViewTemp(CView *anEnclosure,
                        CBureaucrat *aSupervisor,
                        Ptr       viewData)
{
   LongRect   itsBounds;
```

***IViewTemp* method code (concluded)**

```
    inherited::IViewTemp (anEnclosure, aSupervisor, viewData);
    SetLongRect (&itsBounds, 0L, 0L, 540L, 720L);
    SetBounds (&itsBounds);
}
```

The **IViewTemp** method first calls its inherited method and then sets the bounds of the rectangle enclosing the panorama. In this case, bounds values of 540 and 720 pixels correspond to a 7.5- by 10.0-inch image area.

Draw Method Code

The **Draw** method is called by the TCL whenever the panorama's contents need to be redrawn. The generated code for this method has made the right preparations for the code that we have added (see page 371). The customized code is as follows:

***Draw* method code (beginning)**

```
void  CUser4::Draw(Rect  *area)
{
    Rect        theFrame;
    PenState    savePen;
    short       curFont;
    short       curSize;
    Style       curStyle;

    GetPenState (&savePen);
    PenNormal ();
    FrameToQDR (&frame, &theFrame);

    //
    // save font settings
    //
    curFont  = macPort->txFont;
    curSize  = macPort->txSize;
    curStyle = macPort->txFace;

    itsChartInfo = ((CGraphWindow *)itsSupervisor)->GetChartInfo();
    if(itsChartInfo->chartSettings.modified == 1)
    {
        switch(itsChartInfo->chartSettings.chartType)
        {
            case cHorizontalBarViewID:
            {
                DrawHBarChart(theFrame);
                break;
            }
```

***Draw** method code (concluded)*

```
            case cVerticalBarViewID:
            {
                DrawVBarChart(theFrame);
                break;
            }
            case cXYChartViewID:
            {
                DrawXYChart(theFrame);
                break;
            }
        }
    }

    //
    // restore port font info
    //
    TextFont (curFont);
    TextSize (curSize);
    TextFace (curStyle);

    SetPenState (&savePen);
}
```

After the current settings for the port have been saved and the **PenNormal** toolbox call has reset the default pen state, the current window's frame coordinates are converted from the long coordinates in which they are kept into standard QuickDraw coordinates.

The port's font, size, and style settings are saved, and then the chart settings are accessed via the **GetChartInfo** access method in the **CGraphWindow** class.

Using the settings, the **Draw** method determines whether the chart settings have been modified (which will occur only if the user has filled in and dismissed the **Chart** dialog with the **OK** button). If the settings have not been modified, the draw method restores the font and port settings and returns control to its caller. If the settings have been modified, the **Draw** method determines what type of chart has been requested and calls the appropriate method to produce the chart.

The bulk of the code to draw the various charts is contained in new methods that we have defined, called **DrawHBarChart**, **DrawVBarChart**, and **DrawXYChart**. The amount of code in each of these routines is large; thus, it is presented over several sections.

DrawHBarChart Method Code

The **DrawHBarChart** method is responsible for drawing a horizontal bar chart, using the specifications from the recently completed **Chart** dialog. The code in this method must deal with accessing the worksheet data and label cells, perform automatic scaling of the data (if requested), and draw a bar chart that will fit within the existing window frame (if possible). For reasons of simplicity, all axis annotations are created on the basis that the axis (horizontal in this case) will be divided into five segments, each of which will be annotated. In theory, the axis can be divided into any number of segments, depending on the range of the data values to be portrayed; however, this would just add complexity to an already complicated task. The first section of **DrawHBarChart** is as follows:

DrawHBarChart
method code
(section 1)

```
void  CUser4::DrawHBarChart(Rect theFrame)
{
   CCalcWindow  *theWorksheet;
   chartInfo       settings;
   minMax          hScaleInfo;
   Rect             vDataRange, vLabelRange, chartBorder;
   short            deltaY, deltaX, frameHeight, frameWidth;
   short            numBars, barHeight, chartWd, chartHt;
   short            leftMargin, topMargin;
   short            labelMaxWid, labelDX, labelDY;
   double          minValue, maxValue, valueDiff, xDiff;
   decform         theFormat;

   //
   // access the chart settings, get the data range
   // information, and compute the number of bars.
   //
   theWorksheet = ((CGraphWindow *)itsSupervisor)->GetCalcWindow();
   settings = itsChartInfo->chartSettings;
   vDataRange = itsChartInfo->GetVData();
   deltaX = deltaY = 0;
   if(vDataRange.left == vDataRange.right)
   {
      numBars = vDataRange.bottom - vDataRange.top + 1;
      deltaY = 1;
   }
   else
```

***DrawHBarChart* method code (section 1, continued)**

```
{
   numBars = vDataRange.right - vDataRange.left + 1;
   deltaX = 1;
}
```

The first section of the **DrawHBarChart** contains the function definition, its local variable definitions, and some initialization code. The variable **theWorksheet** is set to hold the handle to the **CCalcWindow** instance, and the variable called **settings** is set to hold the **Chart** dialog settings. Following this, the code acquires the cell values for the vertical data range setting and determines whether the range is specified as a horizontal row or vertical column of cells. It also calculates the number of bars that will be produced.

***DrawHBarChart* method code (section 2)**

```
//
// calculate the frame settings, so that we can
// generate a graph that fills the active window.
//
frameHeight = theFrame.bottom - theFrame.top;
frameWidth  = theFrame.right  - theFrame.left;
barHeight = GetBarThickness (numBars, frameHeight - TOPMARGIN
   - BOTMARGIN);
chartHt = numBars * (barHeight + BARSPACING);
leftMargin = LHTMARGIN;
labelMaxWid = 0;
labelDX = labelDY = 0;
if(settings.vLabelCheck)
{
   vLabelRange = itsChartInfo->GetVLabel();
   if(vLabelRange.left == vLabelRange.right)
      labelDY = 1;
   else
      labelDX = 1;
   labelMaxWid = GetLabelMax (theWorksheet, vLabelRange);
}
if(settings.scalingType == cAutomaticScaleViewID)
{
   GetDataMinMax (theWorksheet, vDataRange, &minValue,
      &maxValue, &xDiff);
}
else
{
   hScaleInfo = itsChartInfo->GetHScale();
   minValue = hScaleInfo.min;
   maxValue = hScaleInfo.max;
```

DrawHBarChart *method code (section 2, continued)*

```
    xDiff = (maxValue - minValue) / 5.0; // constant number of intervals
}
valueDiff = maxValue - minValue;
theFormat = GetFormat (xDiff);
```

The second section of the **DrawHBarChart** method calculates the dimensions of the window frame, sets up the appropriate bar thickness (using the **GetBarThickness** method), and calculates the left margin width. It checks to see whether a label range was specified, and if so, it calculates the width of the longest label by using the **GetLabelMax** method. Following this, if automatic scaling was selected, **DrawHBarChart** calls the **GetDataMinMax** method to ascertain the minimum, maximum, and rounded **xDiff** value in the worksheet data values. If manual scaling was specified, the minimum and maximum values are specified explicitly, and the **xDiff** value is calculated using a constant of five divisions in the horizontal axis. Finally, the **valueDiff** value is calculated as the difference between the minimum and maximum values. Using **xDiff** (the division difference), the **GetFormat** method chooses an appropriate format for the axis annotation.

DrawHBarChart *method code (section 3)*

```
//
// if everything looks okay, get ready to
// draw the chart, complete with labels.
//
if(maxValue > minValue)
{
    Rect    barRect;
    Point   dataCell, labelCell;
    short   index, labelH, labelV;
    double  dataValue;
    short   barLength, labelWidth, totalWidth;
    Str255  label;

    chartBorder.left = leftMargin + labelMaxWid + VLABSPACE;
    chartBorder.right = frameWidth - RHTMARGIN;
    chartWd = chartBorder.right - chartBorder.left;
    chartBorder.top = TOPMARGIN;
    chartBorder.bottom = chartBorder.top + chartHt;
    DrawChartFrame (chartBorder);

    //
    // draw title if specified
    //
    if(settings.title[0] > 0)
    {
```

DrawHBarChart
method code
(section 3, continued)

```
    totalWidth   = chartBorder.right - LHTMARGIN;
    TextFont (0);      // use system font
    TextSize (14);     // use 14-point type
    TextFace (0);      // use plain labels
    labelWidth   = StringWidth (settings.title);
    labelCell.h  = LHTMARGIN + (totalWidth - labelWidth) / 2;
    labelCell.v  = TOPMARGIN / 2;
    MoveTo (labelCell.h, labelCell.v);
    DrawString (settings.title);
}

barRect.top = chartBorder.top + BARSPACING;
barRect.left = chartBorder.left;
barRect.bottom = chartBorder.top + barHeight;
dataCell.h  = vDataRange.left;
dataCell.v  = vDataRange.top;
```

The third section of the **DrawHBarChart** method is concerned with drawing the chart border and title (if specified). The border consists of the horizontal and vertical axis lines only. If the title is specified, it is drawn using the system font, in its 14-point size, centered within the frame. Finally, the location of the first bar to be drawn and the location of the worksheet cell corresponding to the first bar are initialized.

At this point, we are almost ready to begin drawing the bars and their corresponding labels (if specified). The only remaining operations are to set the text font, size, and style for drawing the labels and to specifythe location of the first label to be drawn.

DrawHBarChart
method code
(section 4)

```
//
// set label font
//
TextFont (0);      // use system font
TextSize (12);     // use 12-point type
TextFace (0);      // use plain labels

labelH = leftMargin;
labelV = barRect.top + barHeight/2;
labelCell.h = vLabelRange.left;
labelCell.v = vLabelRange.top;
for(index=0; index < numBars; index++)
{
    theWorksheet->GetValueValue (dataCell, &dataValue);
    barLength = ((dataValue - minValue) / valueDiff) * chartWd;
    barRect.right = barRect.left + barLength;
```

DrawHBarChart *method code (section 4, concluded)*

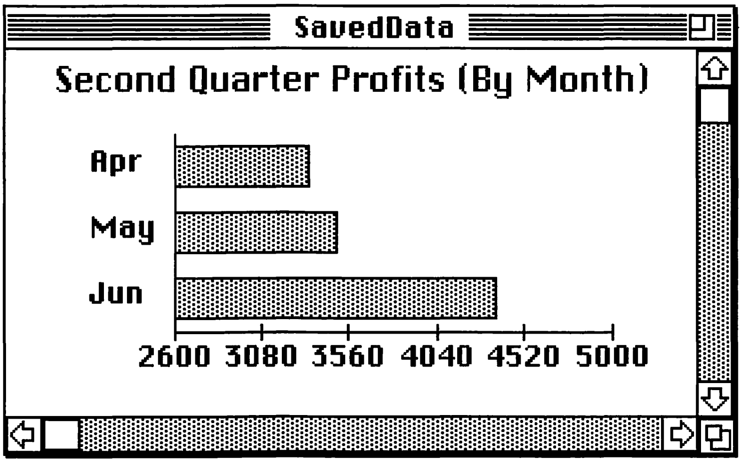
```
            //
            // draw the bar
            //
            FillRect (&barRect, ltGray);
            FrameRect (&barRect);
            //
            // draw the label
            //
            if(settings.vLabelCheck)
            {
                theWorksheet->GetValueString (labelCell, label);
                MoveTo (labelH, labelV);
                DrawString(label);
            }
            //
            // update the settings for the next bar
            //
            barRect.top      += (barHeight + BARSPACING);
            barRect.bottom   += (barHeight + BARSPACING);
            dataCell.h       += deltaX;
            dataCell.v       += deltaY;
            labelV           += (barHeight + BARSPACING);
            labelCell.h      += labelDX;
            labelCell.v      += labelDY;
        }
        DrawHorizTicks (chartBorder, minValue, maxValue, theFormat);
    }
}
```

The last section of the **DrawHBarChart** method contains the loop that draws each of the horizontal bars and its corresponding label, if one was specified. As previously indicated, prior to entering the loop, the label font, size, and style and the location of the first label are initialized.

The loop will draw the number of bars (and labels) specified in the **numBars** variable. For each bar, the double-precision floating-point value of the indicated worksheet cell is acquired, and the length of the bar is computed by using the ratio of the current value to the overall value range, scaled by the chart width in pixels and stored as an integer value. Once the bar's length has been computed, its right-hand coordinate can be computed, and the bar is drawn as a filled and outlined rectangle. Only a single shade of light gray is used for all bars. After the bar has been drawn, its corresponding label is also drawn, using the prespecified font, size, and style. The location of the next bar and label are computed at the bottom of the loop. The next data and label worksheet cells are also computed.

When the loop is complete, we call the **DrawHorizTicks** method to annotate the horizontal axis with the values associated with each of the five horizontal divisions of the chart. A typical chart has the appearance shown in Figure 14-1.

Figure 14-1
Sample horizontal bar chart

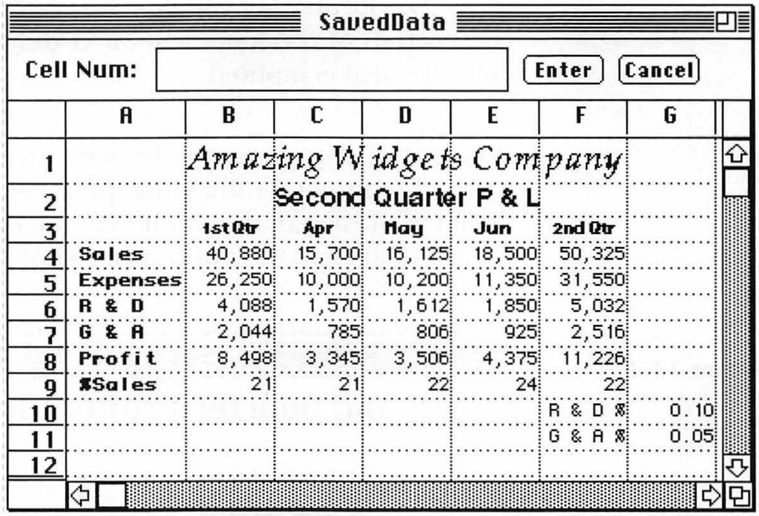


The values for creating the chart in Figure 14-1 were taken directly from the worksheet shown in Figure 14-2. This worksheet expands on the previously entered data for the fictitious Amazing Widgets Company. The data were automatically scaled and plotted using a vertical data range of **C8..E8** and a label range of **C3..E3**. The data values charted in this instance are **3345**, **3506**, and **4375**.

DrawVBarChart Method Code

The **DrawVBarChart** method is very similar to the **DrawHBarChart** method, except that the bars are vertical, instead of horizontal. The code for both methods could probably be combined for efficiency of storage, but the combined code would be more complex to describe. Therefore, each method is self-contained, except for the common methods that each calls.

Figure 14-2
Sample worksheet window from which values for charts are taken


SavedData

Cell Num: Enter Cancel

	A	B	C	D	E	F	G
1		*Amazing Widgets Company*					
2			Second Quarter P & L				
3		1st Qtr	Apr	May	Jun	2nd Qtr	
4	Sales	40,880	15,700	16,125	18,500	50,325	
5	Expenses	26,250	10,000	10,200	11,350	31,550	
6	R & D	4,088	1,570	1,612	1,850	5,032	
7	G & A	2,044	785	806	925	2,516	
8	Profit	8,498	3,345	3,506	4,375	11,226	
9	%Sales	21	21	22	24	22	
10						R & D %	0.10
11						G & A %	0.05
12							

As with the code for **DrawHBarChart**, the **DrawVBarChart** method is presented in several sections. The first section contains the method declaration, the declaration of local variables used throughout the method, and the code to acquire the worksheet handle and data range information. The code for this section is as follows:

DrawVBarChart *method code (section 1)*

```
void  CUser4::DrawVBarChart(Rect theFrame)
{
   CCalcWindow      *theWorksheet;
   chartInfo        settings;
   minMax           vScaleInfo;
   Rect             hDataRange, hLabelRange, chartBorder;
   short            deltaY, deltaX, frameHeight, frameWidth;
   short            numBars, barSize, chartWd, chartHt;
   short            leftMargin, topMargin;
   short            labelMaxWid, labelDX, labelDY, tickLabelWd;
   double           minValue, maxValue, valueDiff, yDiff;
   decform          theFormat;

   //
   // access the chart settings, get the data range
   // information, and compute the number of bars.
   //
   theWorksheet = ((CGraphWindow *)itsSupervisor)->GetCalcWindow();
   settings = itsChartInfo->chartSettings;
   hDataRange = itsChartInfo->GetHData();
```

DrawVBarChart
method code
(section 1, continued)

```
deltaX = deltaY = 0;
if(hDataRange.left == hDataRange.right)
{
   numBars = hDataRange.bottom - hDataRange.top + 1;
   deltaY = 1;
}
else
{
   numBars = hDataRange.right - hDataRange.left + 1;
   deltaX = 1;
}
```

The first section of code is responsible for acquiring the worksheet instance handle, getting the horizontal data range, and calculating whether the data are stored in a column or row orientation. The number of bars to be drawn is also determined here.

DrawVBarChart
method code
(section 2)

```
//
// calculate the frame settings, so that we can
// generate a graph that fills the active window.
//
frameHeight = theFrame.bottom - theFrame.top;
frameWidth  = theFrame.right  - theFrame.left;
barSize = GetBarThickness (numBars, frameWidth - LHTMARGIN
    - RHTMARGIN);
chartWd = numBars * (barSize + BARSPACING);
leftMargin = LHTMARGIN;
labelMaxWid = 0;
labelDX = labelDY = 0;

if(settings.hLabelCheck)
{
   hLabelRange = itsChartInfo->GetHLabel();
   if(hLabelRange.left == hLabelRange.right)
   {
      labelDY = 1;
   }
   else
   {
      labelDX = 1;
   }
   labelMaxWid = GetLabelMax (theWorksheet, hLabelRange);
}
tickLabelWd = GetLabelMax (theWorksheet, hDataRange);

if(settings.scalingType == cAutomaticScaleViewID)
{
```

DrawVBarChart *method code (section 2, continued)*

```
    GetDataMinMax (theWorksheet, hDataRange, &minValue,
        &maxValue, &yDiff);
}
else
{
    vScaleInfo = itsChartInfo->GetVScale();
    minValue = vScaleInfo.min;
    maxValue = vScaleInfo.max;
    yDiff = (maxValue - minValue) / 5.0; // constant number of intervals
}
valueDiff = maxValue - minValue;
theFormat = GetFormat (yDiff);
```

The second section of code is responsible for calculating the dimensions of the frame, the chart width, and the thickness of the bars (by calling the **GetBarThickness** method), as well as for determining whether labels have been specified and accessing the label range if so. In addition, if automatic scaling is selected, the **GetDataMinMax** method is used to determine appropriate minimum and maximum values, as well as **yDiff**, the value associated with each division on the vertical axis. If manual scaling is selected, the specified values are acquired from the settings by using the **GetVScale** access method, and then the **yDiff** value is computed, based on a constant five divisions on the y-axis. Finally, the total difference between the minimum and maximum values is calculated, and the format for displaying the y-axis annotations is determined by calling the **GetFormat** method.

DrawVBarChart *method code (section 3)*

```
//
// if everything looks okay, get ready to
// draw the chart, complete with labels.
//
if(maxValue > minValue)
{
    Rect    barRect;
    Point   dataCell, labelCell;
    short   index, labelH, labelV;
    double  dataValue;
    short   barLength, labelWidth, totalWidth;
    Str255  label;

    chartBorder.left = leftMargin + tickLabelWd + VLABSPACE;
    chartBorder.right = chartBorder.left + chartWd;
    chartBorder.top = TOPMARGIN;
    chartBorder.bottom = frameHeight - BOTMARGIN;
    chartHt = chartBorder.bottom - chartBorder.top;
```

DrawVBarChart
method code
(section 3, continued)

```
DrawChartFrame (chartBorder);

//
// draw title if specified
//
if(settings.title[0] > 0)
{
   totalWidth   = chartBorder.right - LHTMARGIN;
   TextFont (0);      // use system font
   TextSize (14);     // use 14-point type
   TextFace (0);      // use plain labels
   labelWidth   = StringWidth (settings.title);
   labelCell.h  = LHTMARGIN + (totalWidth - labelWidth) / 2;
   labelCell.v  = TOPMARGIN / 2;
   MoveTo (labelCell.h, labelCell.v);
   DrawString (settings.title);
}

barRect.bottom = chartBorder.bottom + 1;
barRect.left = chartBorder.left + BARSPACING;
barRect.right = chartBorder.left + barSize;
dataCell.h  = hDataRange.left;
dataCell.v  = hDataRange.top;

//
// set label font
//
TextFont (0);      // use system font
TextSize (12);     // use 12-point type
TextFace (0);      // use plain labels

labelH = barRect.left + barSize/2 - 1;
labelV = barRect.bottom + (BOTMARGIN / 2);
labelCell.h = hLabelRange.left;
labelCell.v = hLabelRange.top;
```

The third section of code begins the actual drawing of the chart. After calculating the final dimensions of the chart, it calls the **DrawChartFrame** method to draw the horizontal and vertical axes of the chart. In addition, if a title was specified, it is drawn in the 14-point system font. After the title is drawn, the text font, size, and style are changed to prepare for drawing the labels in the final section of the code. The 12-point version of the system font is used for this purpose. The final set of statements calculates the position of the first label if one is to be drawn.

DrawVBarChart *method code (section 4, concluded)*

```
        for(index=0; index < numBars; index++)
        {
            theWorksheet->GetValueValue (dataCell, &dataValue);
            barLength = ((dataValue - minValue) / valueDiff) * chartHt;
            barRect.top = barRect.bottom - barLength;

            //
            // draw the bar
            //
            FillRect (&barRect, ltGray);
            FrameRect (&barRect);

            //
            // draw the label
            //
            if(settings.hLabelCheck)
            {
                theWorksheet->GetValueString (labelCell, label);
                MoveTo (labelH - (StringWidth(label)/2), labelV);
                DrawString(label);
            }
            //
            // update the settings for the next bar
            //
            barRect.left    += (barSize + BARSPACING);
            barRect.right   += (barSize + BARSPACING);
            dataCell.h      += deltaX;
            dataCell.v      += deltaY;
            labelH          += (barSize + BARSPACING);
            labelCell.h     += labelDX;
            labelCell.v     += labelDY;
        }
        DrawVertTicks (chartBorder, minValue, maxValue, theFormat);
    }
}
```

The last section of code contains the main loop that draws each of the specified number of bars and their labels. The bars are drawn with the **FillRect** and **FrameRect** toolbox calls, and the labels, if selected, are drawn using the **DrawString** toolbox routine. After each bar and label are drawn, the location for the next bar and label is computed, in preparation for the next iteration of the loop. Finally, after all the bars and labels have been drawn, the **DrawVertTicks** method is called to annotate the vertical axis. A sample vertical bar chart, drawn with **DrawVBarChart** is shown in Figure 14-3.

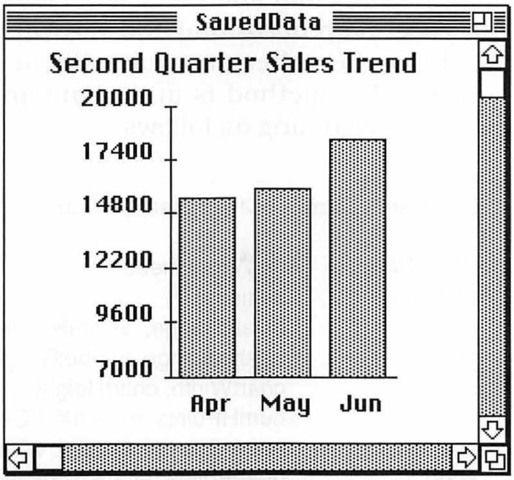


Figure 14-3
Sample vertical bar chart

The data for the vertical bar chart shown in the figure were taken directly from the worksheet range **C4..E4**, and the labels come from the range **C3..E3**. The scaling is automatic, and the data values depicted by the bars are **15700**, **16125**, and **18500**, as shown in the worksheet depicted in Figure 14-2.

DrawXYChart Method Code

The **DrawXYChart** method uses portions of the same logic as both the horizontal and vertical bar chart methods. However, with it, the data points are plotted at the intersection of the x-axis and y-axis values. For the points, we have chosen to use the '•' character (Option-8), which is available in the system font.

The **DrawXYChart** method is quite different from the previous methods with respect to drawing the chart labels. In this case, the label is not associated with an individual data point, but rather, serves as an axis title. If a range is given for the label, each cell is taken to be a word in the label and will be displayed with an automatically appended space character prior to displaying the subsequent word. The horizontal label is displayed on the x-axis, and the vertical label is displayed, vertically, on the y-axis.

As with horizontal and vertical bar charts, the data values (in this case, both horizontal and vertical values are required inputs) may be either automatically or manually scaled. The code for this method is quite long and is shown in several sections, beginning as follows:

DrawXYChart
method code (section 1)

```
void CUser4::DrawXYChart (Rect theFrame)
{
   CCalcWindow     *theWorksheet;
   chartInfo       settings;
   Rect            hDataRange, vDataRange, chartBorder;
   Rect            hLabelRange, vLabelRange;
   short           chartWidth, chartHeight;
   short           numHPoints, hDeltaX, hDeltaY;
   short           numVPoints, vDeltaX, vDeltaY;
   short           hLabPoints, hLabDx, hLabDy;
   short           vLabPoints, vLabDx, vLabDy;
   Str255          hLabel, vLabel, tLabel;
   short           vAxisLabelWd, index, temp;
   minMax          hScaleInfo, vScaleInfo;
   double          hMin, hMax, hDiff, hValueRange;
   double          vMin, vMax, vDiff, vValueRange;
   decform         hFormat, vFormat;

   //
   // get the worksheet reference and the initial
   // horizontal and vertical data settings
   //
   theWorksheet = ((CGraphWindow *)itsSupervisor)->GetCalcWindow();
   settings = itsChartInfo->chartSettings;
   hDataRange = itsChartInfo->GetHData();
   hDeltaX = hDeltaY = 0;
   if(hDataRange.left == hDataRange.right)
   {
      numHPoints = hDataRange.bottom - hDataRange.top + 1;
      hDeltaY = 1;
   }
   else
   {
      numHPoints = hDataRange.right - hDataRange.left + 1;
      hDeltaX = 1;
   }
   vDataRange = itsChartInfo->GetVData();
   vDeltaX = vDeltaY = 0;
   if(vDataRange.left == vDataRange.right)
   {
      numVPoints = vDataRange.bottom - vDataRange.top + 1;
```

DrawXYChart *method code (section 1, continued)*

```
        vDeltaY = 1;
    }
    else
    {
        numVPoints = vDataRange.right - vDataRange.left + 1;
        vDeltaX = 1;
    }
    vAxisLabelWd = GetLabelMax (theWorksheet, vDataRange);
    chartBorder.top = theFrame.top + TOPMARGIN;
    chartBorder.bottom = theFrame.bottom - (BOTMARGIN
        + BOTMARGIN / 2);
    chartBorder.left = theFrame.left + LHTMARGIN + vAxisLabelWd
        + VLABSPACE;
    chartBorder.right = theFrame.right - RHTMARGIN;
    chartWidth  = chartBorder.right  - chartBorder.left;
    chartHeight = chartBorder.bottom - chartBorder.top;
```

The first section of the **DrawXYChart** method contains the method declaration and the declaration of the local variables. In addition, it has code to access the worksheet's instance handle, the settings from the **Chart** dialog, and the horizontal and vertical data range specifications. It also determines whether each of the horizontal and vertical data ranges is stored in a row or column orientation. The number of points in each range is computed as well. Finally, the dimensions of the chart are computed, based upon the height and width of the window frame.

DrawXYChart *method code (section 2)*

```
    //
    // get any horizontal or vertical axis label information
    //
    hLabDx = vLabDx = hLabDy = vLabDy = 0;
    if(settings.hLabelCheck)
    {
        hLabelRange = itsChartInfo->GetHLabel();
        if(hLabelRange.left == hLabelRange.right)
        {
            hLabPoints = hLabelRange.bottom - hLabelRange.top + 1;
            hLabDy = 1;
        }
        else
        {
            hLabPoints = hLabelRange.right - hLabelRange.left + 1;
            hLabDx = 1;
        }
    }
    if(settings.vLabelCheck)
    {
```

DrawXYChart
method code
(section 2, continued)

```
        vLabelRange = itsChartInfo->GetVLabel();
        if(vLabelRange.left == vLabelRange.right)
        {
            vLabPoints = vLabelRange.bottom - vLabelRange.top + 1;
            vLabDy = 1;
        }
        else
        {
            vLabPoints = vLabelRange.right - vLabelRange.left + 1;
            vLabDx = 1;
        }
    }

    //
    // scale the horizontal and vertical data
    //
    if(settings.scalingType == cAutomaticScaleViewID)
    {
        GetDataMinMax (theWorksheet, hDataRange, &hMin, &hMax,
&hDiff);
        GetDataMinMax (theWorksheet, vDataRange, &vMin, &vMax,
&vDiff);
    }
    else
    {
        hScaleInfo = itsChartInfo->GetHScale();
        hMin = hScaleInfo.min;
        hMax = hScaleInfo.max;
        hDiff = (hMax - hMin) / 5.0;

        vScaleInfo = itsChartInfo->GetVScale();
        vMin = vScaleInfo.min;
        vMax = vScaleInfo.max;
        vDiff = (vMax - vMin) / 5.0;
    }
    hValueRange = hMax - hMin;
    vValueRange = vMax - vMin;
```

The second section of the **DrawXYChart** method determines whether horizontal or vertical labels were specified and, if so, accesses their cell ranges and orientations.

If the chart is intended to be automatically scaled, the code calls the **GetDataMinMax** method to calculate the minimum and maximum values, and the value per division of both the x-axis and y-axis data ranges.

If manual scaling was selected, the code accesses the specified minimum and maximum values and then calculates the value per division for both axes. The difference between the minimum and maximum values in each range is also calculated.

It is important to emphasize that each of the settings has been validated before the **Chart** dialog is dismissed. In the case of the X-Y chart, the number of data values in the horizontal and vertical ranges must be equal, and the label ranges must specify a pair of cell values, even though the cell numbers are equal (e.g., **A3..A3**)

DrawXYChart
method code
(section 3)

```
//
// draw title if specified
//
if(settings.title[0] > 0)
{
   short     totalWidth, labelWidth;
   Point     labelCell;

   totalWidth  = chartBorder.right - LHTMARGIN;
   TextFont (0);       // use system font
   TextSize (14);      // use 14-point type
   TextFace (0);       // use plain labels
   labelWidth  = StringWidth (settings.title);
   labelCell.h = LHTMARGIN + (totalWidth - labelWidth) / 2;
   labelCell.v = TOPMARGIN / 2;
   MoveTo (labelCell.h, labelCell.v);
   DrawString (settings.title);
}

//
// draw the border, horizontal and vertical labels
//
DrawChartFrame (chartBorder);
if(settings.hLabelCheck)
{
   Point labelCell;
   short labelWidth, labelH, labelV;

   labelCell.h = hLabelRange.left;
   labelCell.v = hLabelRange.top;
   theWorksheet->GetValueString (labelCell, hLabel);
   for(index=1; index < hLabPoints; index++)
   {
      labelCell.h += hLabDx;
      labelCell.v += hLabDy;
      ConcatPStrings (hLabel, "\p ");
```

***DrawXYChart* method code (section 3, continued)**

```
        theWorksheet->GetValueString (labelCell, tLabel);
        ConcatPStrings (hLabel, tLabel);
    }
    TextFont (0);       // use system font
    TextSize (12);      // use 12-point type
    TextFace (0);       // use plain labels
    labelWidth = StringWidth (hLabel);
    labelH = chartBorder.left + (chartWidth/2) - (labelWidth/2);
    labelV = chartBorder.bottom + BOTMARGIN;
    MoveTo (labelH, labelV);
    DrawString (hLabel);
}
if(settings.vLabelCheck)
{
    Point labelCell;
    FontInfo fi;
    short labelHeight, labelH, labelV;
    labelCell.h = vLabelRange.left;
    labelCell.v = vLabelRange.top;
    theWorksheet->GetValueString (labelCell, vLabel);
    for(index=1; index < vLabPoints; index++)
    {
        labelCell.h += vLabDx;
        labelCell.v += vLabDy;
        ConcatPStrings (vLabel, "\p ");
        theWorksheet->GetValueString (labelCell, tLabel);
        ConcatPStrings (vLabel, tLabel);
    }
    TextFont (0);       // use system font
    TextSize (12);      // use 12-point type
    TextFace (0);       // use plain labels
    GetFontInfo(&fi);
    labelHeight = (fi.ascent+fi.descent) * vLabel[0];
    labelH = theFrame.left + (LHTMARGIN/2);
    labelV = chartBorder.top + (chartHeight/2) - (labelHeight/2);
    for(index=1; index <= vLabel[0]; index++)
    {
        MoveTo (labelH - (CharWidth (vLabel[index])/2), labelV);
        DrawChar (vLabel[index]);
        labelV += (fi.ascent + fi.descent);
    }
}
```

The third section of the **DrawXYChart** method is responsible for drawing the chart border, the title (if specified), and the horizontal and vertical axis labels (if specified). The title is drawn in the 14-point system font, while the labels are drawn in the 12-point system font. The title is centered in the frame, at the top of the graph. The horizontal axis label, if specified, is drawn inside the bottom margin of the frame, also centered within the frame. The vertical axis label is drawn inside the left margin of the frame, and not only is it vertically centered in the frame, but each character is centered with respect to the others.

DrawXYChart *method code (section 4)*

```
//
// draw the axis ticks and scaling labels
//
hFormat = GetFormat (hDiff);
vFormat = GetFormat (vDiff);
DrawHorizTicks (chartBorder, hMin, hMax, hFormat);
DrawVertTicks  (chartBorder, vMin, vMax, vFormat);

//
// finally, plot the data points
//
if(numHPoints > 0)
{
   char      plotIt;
   Point     charLoc, hData, vData;
   double    hValue, vValue;
   short     charWd, deltaH, deltaV;

   TextFont (0);        // set system font
   TextSize (12);       // set 12-point size
   TextFace (0);        // set plain style
   plotIt = '•';
   charWd = CharWidth(plotIt)/2;
   hData.h = hDataRange.left;
   hData.v = hDataRange.top;
   vData.h = vDataRange.left;
   vData.v = vDataRange.top;
   for(index=0; index < numHPoints; index++)
   {
      theWorksheet->GetValueValue (hData, &hValue);
      theWorksheet->GetValueValue (vData, &vValue);
      deltaH = ((hValue - hMin)/hValueRange) * chartWidth;
      deltaV = ((vValue - vMin)/vValueRange) * chartHeight;
      charLoc.h = chartBorder.left + deltaH - charWd;
      charLoc.v = chartBorder.bottom - deltaV - charWd;
      MoveTo (charLoc.h, charLoc.v);
      DrawChar (plotIt);
      hData.h += hDeltaX;
      hData.v += hDeltaY;
      vData.h += vDeltaX;
      vData.v += vDeltaY;
```

***DrawXYChart** method code (section 4, concluded)*

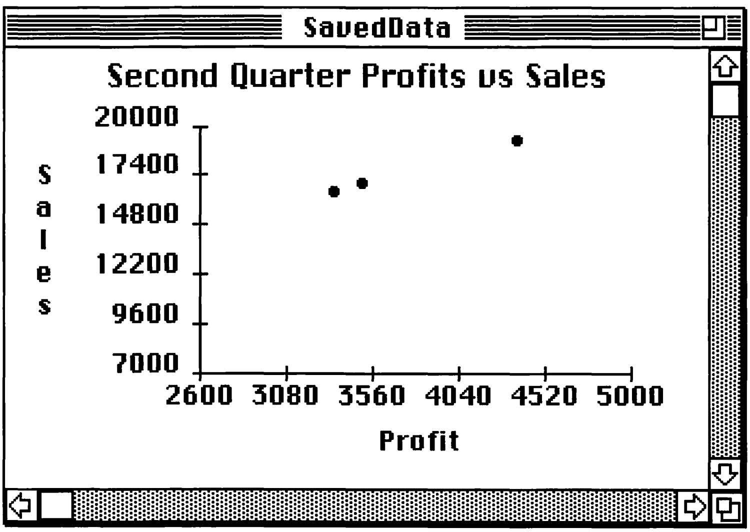
```
        }
    }
}
```

The final section of the **DrawXYChart** method performs the function of drawing the axis annotations, followed by drawing the individual data points. The annotation values for the axes are determined by calling the **GetFormat** method for both the horizontal and vertical axis, using the value difference computed for each of the five divisions in each axis.

The horizontal and vertical axes are annotated by calling the **DrawHTicks** and **DrawVTicks** methods, respectively.

After annotation of the axes is complete, the loop to draw the individual data points is entered. The '•' character is plotted, with its center located at the junction of the horizontal and vertical data positions to which it corresponds. After each point is plotted, the cell coordinates for both the horizontal and vertical data ranges are incremented to the location of the next point. After the last point has been plotted, the chart is complete. A sample X-Y chart is shown in Figure 14-4.

Figure 14-4
Sample X-Y chart



As with the other sample charts, the data values for the chart shown in Figure 14-4 are taken directly from the sample worksheet depicted in Figure 14-2. In this case, the pairs of data values being plotted are **(3345, 15700)**, **(3506, 16125)**, and **(4375, 18500)**. This is clearly an upward trend for our fictitious company.

GetBarThickness Method Code

The **GetBarThickness** method is called by both the horizontal and vertical bar chart methods to determine the appropriate width, or thickness, of the bars. The thickness is computed on the basis of the number of bars and the size of the horizontal or vertical space in which they must fit. The code is as follows:

```
short CUser4::GetBarThickness (short numBars, short frameSize)
{
   short    height;
   short    barSize;

   barSize = (frameSize - numBars * BARSPACING) / numBars;
   if (barSize < MINBARSIZE)
   {
      return MINBARSIZE;
   }
   if (barSize > MAXBARSIZE)
   {
      return MAXBARSIZE;
   }
   return barSize;
}
```

The code uses a few predefined constants that specify the minimum and maximum thickness of bars to be drawn. The values for these constants in the current implementation are set to 9 and 36 pixels, respectively. The desired thickness is first computed and then compared against the MINBARSIZE and MAXBARSIZE constants. It is clipped to either the minimum or maximum value if it is not within the specified range. The BARSPACING constant ensures that bars are spaced from one another by a standard amount. The default value for this constant is 5 points (i.e., five pixels at 72 dots per inch).

GetLabelMax Method Code

The **GetLabelMax** method is responsible for finding the length of the longest label in the specified label range. The code for this method is as follows:

```
short CUser4::GetLabelMax (CCalcWindow  *theWorksheet,
                           Rect  labelRange)
{
   short    deltaX, deltaY, num, i, maxWidth, labelWidth;
   Point    labelCell;
   Str255   label;
   deltaX = deltaY = 0;
   maxWidth = -1;
   if(labelRange.top == labelRange.bottom)
   {
      deltaX = 1;
      num = labelRange.right - labelRange.left + 1;
   }
   else
   {
      deltaY = 1;
      num = labelRange.bottom - labelRange.top + 1;
   }
   TextFont (0);         // use system font
   TextSize (12);        // use 12-point type
   TextFace (0);         // use plain style
   SetPt(&labelCell, labelRange.left, labelRange.top);
   for(i=0; i < num; i++) {
      theWorksheet->GetValueString (labelCell, label);
      labelWidth = StringWidth(label);
      if(labelWidth > maxWidth)
         maxWidth = labelWidth;
      labelCell.h += deltaX;
      labelCell.v += deltaY;
   }
   return  maxWidth;
}
```

The **GetLabelMax** method accomplishes its task by using the handle to the worksheet and the specified label range to access each label and calculate its width, based upon its being drawn on the screen with 12-point plain type in the system font. The **StringWidth** toolbox method is used to calculate each label's width. **GetLabelMax** returns the largest width.

GetDataMinMax Method Code

The **GetDataMinMax** method is called by all the charting methods when automatic scaling is selected. The purpose of this method is to select minimum and maximum values that best fit the data to be charted, as well as to calculate the value corresponding to each division of the chart. The code for this method is as follows:

```
void CUser4::GetDataMinMax (CCalcWindow *theWorksheet,
                            Rect   dataRange, double *minValue,
                            double *maxValue, double *xDiff)
{
   short    deltaX, deltaY, num, i;
   double   min, max, value, diff;
   Cell     dataCell;

   min =  9.9e999;
   max = -9.9e999;
   deltaX = deltaY = 0;
   if(dataRange.top == dataRange.bottom)
   {
      deltaX = 1;
      num = dataRange.right - dataRange.left + 1;
   }
   else
   {
      deltaY = 1;
      num = dataRange.bottom - dataRange.top + 1;
   }
   SetPt(&dataCell, dataRange.left, dataRange.top);
   for(i=0; i < num; i++)
   {
      theWorksheet->GetValueValue (dataCell, &value);
      if(value < min)
      {
         min = value;
      }
      if(value > max)
      {
         max = value;
      }
      dataCell.h += deltaX;
      dataCell.v += deltaY;
   }
   min  = RoundDown (min);
   max  = RoundUp (max);
   diff = RoundUp ((max - min) / 5.0);
   *minValue = min - diff;
   *maxValue = max;
   *xDiff = diff;
}
```

The **GetDataMinMax** method computes the appropriate minimum and maximum values by accessing each worksheet cell in the specified data range, calculating the actual minimum and maximum values, calling the **RoundDown** function to calculate a new minimum value, and then calling the **RoundUp** function to calculate a new maximum value. These functions use a table of logarithms to aid in rounding the values to the nearest lesser or greater value, as will be described later. After the new minimum and maximum values are computed, the difference between these is divided by five and then rounded up, using the same **RoundUp** function. A new minimum value is then computed to be the previous (rounded-down) minimum value less the value of the difference per division. Thus, the displayed minimum value will always be less than the actual minimum value, which guarantees that a bar or point for that value will always appear within the chart and not be drawn on the corresponding axis. In particular, in a horizontal or vertical bar chart, the bar will have a nonzero length. The newly computed minimum and maximum values, and the difference per division are stored into the variables to which their pointer arguments refer.

DrawChartFrame Method Code

The **DrawChartFrame** method is responsible for drawing the frame for all three types of charts. The frame consists of lines that represent the horizontal and vertical axes of the corresponding chart. The code for this method is as follows:

```
void  CUser4::DrawChartFrame (Rect chartBorder)
{
   MoveTo (chartBorder.left, chartBorder.top);
   LineTo (chartBorder.left, chartBorder.bottom);
   LineTo (chartBorder.right, chartBorder.bottom);
}
```

The method takes a Rect as an argument and draws the axes using standard Quickdraw commands.

DrawHorizTicks Method Code

The **DrawHorizTicks** method is responsible for drawing the tick marks and numerical annotations for the horizontal axis of a chart. The code is as follows:

```
void CUser4::DrawHorizTicks (Rect chartBorder, double min,
                             double max, decform format)
{
    short       tickH, tickV, tickHt;
    short       index, width;
    extended    start;
    double      range, chartWidth, delta, value;
    Str32       label;

    chartWidth = chartBorder.right - chartBorder.left;

    //
    // draw the tick marks and axis labels
    // based on a constant 5 ticks / chart
    //
    range = max - min;
    delta = range / 5.0;
    value = min;
    tickH = chartBorder.left;
    tickV = chartBorder.bottom;
    tickHt = 3;
    for(index=0; tickH <= chartBorder.right; index++)
    {
        MoveTo (tickH, tickV - tickHt);
        LineTo (tickH, tickV + tickHt);
        x96tox80 (&value, &start);
        num2str (&format, start, label);
        width = StringWidth (label);
        MoveTo (tickH - (width >> 1), tickV + 15);
        DrawString(label);
        value += delta;
        tickH = chartBorder.left + (((value - min) / range) * chartWidth);
    }
}
```

The **DrawHorizTicks** method takes the chart border and the minimum and maximum values, and the specified data format and draws each tick mark and label on the horizontal axis, according to the value associated with each horizontal division of the chart.

DrawVertTicks Method Code

The **DrawVertTicks** method is similar to the previous method, but draws tick marks and annotations on the vertical, rather than the horizontal, axis. The code is as follows:

```
void  CUser4::DrawVertTicks (Rect chartBorder, double min,
                                        double max, decform format)
{
    short         tickH, tickV, tickWd, index, width;
    extended    start;
    double       range, chartHeight, delta, value;
    Str32         label;

    chartHeight = chartBorder.bottom - chartBorder.top;

    //
    // draw the tick marks and axis labels
    // based on a constant 5 ticks / chart
    //
    range = max - min;
    delta = range / 5.0;
    value = min;
    tickH = chartBorder.left;
    tickV = chartBorder.bottom;
    tickWd = 3;
    for(index=0; tickV >= chartBorder.top; index++)
    {
        MoveTo (tickH - tickWd, tickV);
        LineTo (tickH + tickWd, tickV);
        x96tox80 (&value, &start);
        num2str (&format, start, label);
        width = StringWidth (label);
        MoveTo (tickH - width - VLABSPACE, tickV);
        DrawString(label);
        value += delta;
        tickV  = chartBorder.bottom - (((value - min) / range) * chartHeight);
    }
}
```

The **DrawVertTicks** method uses the chart border Rect and the minimum and maximum values, and the annotation format as input. It calculates the **delta** value by computing the difference between the minimum and the maximum values and then divides this by five (divisions). The tick marks are drawn at the **delta** interval, accompanied by the corresponding axis values formatted according to the specified format.

GetFormat Method Code

The **GetFormat** method is responsible for determining how many decimal digits will be displayed for the axis annotation values when these are drawn in the **DrawHorizTicks** and **DrawVertTicks** methods.

The intention is to show only as many decimals as are necessary to guarantee that the axis annotations are each unique within the minimum and maximum data ranges. For example, if the data range is between 0.00 and 0.08, then it would be important to display at least two decimal digits in the annotations. By contrast, if the data range is 3,000 to 6,000, then it is not necessary to show any decimal digits, as the distinction between values would be difficult to discriminate visually to that degree of resolution. Therefore, the code for this method adopts a simple precept. It calculates the base-10 logarithm of the **valueDiff** argument and saves its integral portion into the local **digits** variable for comparison. If **digits** is 0, then one decimal digit is included in the format returned. If the **digits** value is greater than 0, then the format will be set to zero decimal digits. If the **digits** value is negative, then the negative of that value (a positive number) plus 1 is used for the number of decimal digits.

```
decform CUser4::GetFormat (double valueDiff)
{
    decform     aFormat;
    short       digits;
    digits = (short)log10x (valueDiff);
    if(digits >= 0)
    {
        if (digits > 0)
        {
            digits = 0;
        }
        else
        {
            digits = 1;
        }
    }
    else
    {
        digits = -digits + 1;
    }
    aFormat.style  = FIXEDDECIMAL;
    aFormat.digits = digits;
    return aFormat;
}
```

The preceding code creates a display format that shows a single decimal digit for a value range of 1 to 9, no decimal digits for values larger than that, and one additional decimal digit for fractional value ranges less than 1.

Global Functions Used by the CUser4 Class Methods

The **CUser4** class methods just presented refer to several routines that are coded as global functions. These routines are used only by the **CUser4** methods, but are defined to be global by the nature of the functions that they perform.

log10x Function Code

The **log10x** function computes the base-10 logarithm of the input value and returns this result as a double-precision floating-point value. The function uses the SANE library function for the natural logarithm and the formula

$$\log(x) = \ln(x)\log(e)$$

to calculate the base-10 logarithm. The base-10 logarithm of *e* is precomputed as a constant value. The code for the **log10x** function is as follows:

```
#define LOG10E  0.4342944819032518278L
double log10x (double x)
{
   double      dLog10x;
   extended    eLogx, eLog10e;
   x96tox80 (&x, &eLogx);
   eLogx = log (eLogx);
   x80tox96 (&eLogx, &dLog10x);
   return (dLog10x * LOG10E);
}
```

In order to use the SANE library functions, the incoming double-precision floating-point value must be converted to a 10-byte extended format value using the **x96tox80** SANE function. The natural logarithm of this value is taken, and then it is converted back to a 12-byte double-precision value. The result returned is the product of the natural logarithm of the input and the common logarithm of the value ***e***.

exp10x Function Code

The **exp10x** function computes the value of 10 raised to the value of the input parameter. It uses the SANE library function **exp** and the formula

$$10^x = e^{\ln(10)x}$$

to compute the result. The code for this function is as follows:

```
#define LOGe10  2.3025850929940456840L
double exp10x (double x)
{
   extended   temp;
   double     result;

   result = x * LOGe10;
   x96tox80 (&result, &temp);
   temp = exp (temp);
   x80tox96 (&temp, &result);
   return result;
}
```

The code first calculates the product of the input value and the natural logarithm of 10. It then converts this product to a 10-byte SANE extended value and uses the **exp** function to calculate the exponential. The result of that calculation is converted back to a 12-byte double-precision value and is returned to the calling routine.

Lookup Tables for Global Functions

The remaining global functions (**RoundUp**, **RoundDown**, **lookUp**, and **lookDown**) refer to a set of tables of logarithms to accomplish their tasks. Two sets of tables have been predefined for this purpose. The first set contains the common logarithms for values between 1 and 10, the other set for values between 0.1 and 1.0.

These tables apply equally well to even larger and smaller positive quantities. Negative values are handled by the logic of the functions that use the tables. The contents of the two sets of tables are as follows:

Tables of integral and fractional logarithms

```
//
// tables of logarithms for computing
// ranges of the data being charted.
//

double   posLogs[] =
{
   0.0000000000000000000L,  // 1.0
   0.3010299956639811952L,  // 2.0
   0.4771212547196624374L,  // 3.0
   0.6020599913279623904L,  // 4.0
   0.6989700043360188048L,  // 5.0
   0.7781512503836436326L,  // 6.0
   0.8450980400142568306L,  // 7.0
   0.9030899869919435856L,  // 8.0
   0.9542425094393248747L,  // 9.0
   1.0000000000000000000    // 10.0
};
double   negLogs[] =
{
   -1.0000000000000000000L, // 0.1
   -0.6989700043360188046L, // 0.2
   -0.5228787452803375624L, // 0.3
   -0.3979400086720376094L, // 0.4
   -0.3010299956639811952L, // 0.5
   -0.2218487496163563672L, // 0.6
   -0.1549019599857431684L, // 0.7
   -0.0969100130080564141L, // 0.8
   -0.0457574905606751252L, // 0.9
    0.0000000000000000000L  // 1.0
};
```

RoundDown Function Code

The **RoundDown** function is called by the **GetDataMinMax** method to round a minimum value down to the nearest value appropriate to its magnitude. The code for this method is as follows:

RoundDown *function code (beginning)*

```
double RoundDown (double x)
{
   double   logX, fracX, intX;
   if(x < 0.0)
      return (-RoundUp (-x));
   logX  = log10x (x);
   intX  = ((short) logX);
```

RoundDown
function code
(concluded)

```
    fracX = logX - intX;
    if(fracX < 0)
    {
        fracX = lookDown (fracX, negLogs);
        if (fracX == -1.0)
        {
            fracX = 0.0;
            intX -= 1.0;
        }
    }
    else
    {
        fracX = lookDown (fracX, posLogs);
    }
    logX = intX + fracX;
    return (exp10x (logX));
}
```

The **RoundDown** function first determines whether the value to be rounded is positive or negative. If it is negative, we want it to be more negative, so we call the **RoundUp** function with the input value negated and then return the negation of that result. If the input value is positive, we calculate its common logarithm and then compute its integer and fractional parts. If the fractional part is negative, then the input value was less than 1.0. In this case, we call the **lookDown** function to find the first logarithm in the **negLogs** table that has a lower value. If the one that was found has a value of **-1.0**, then we set the fractional component of the result to **0.0** and reduce the integral part of the logarithm by **1**.

If the fractional part is positive, then we call the **lookDown** function to find the next lower valued logarithm in the **posLogs** table and use that as the new fractional part of the result. The final action combines the new integer and fractional parts and returns the exponential function's value as the final result.

RoundUp Function Code

The **RoundUp** function is called by the **GetDataMinMax** method to find the next higher value for the corresponding input value, according to its magnitude. The code for this function is as follows:

```
double RoundUp (double x)
{
   double    logX, fracX, intX;

   if(x < 0.0)
   {
      return (-RoundDown (-x));
   }
   logX  = log10x (x);
   intX  = ((short) logX);
   fracX = logX - intX;
   if(fracX < 0)
   {
      fracX = lookUp (fracX, negLogs);
      if (fracX == 0.0)
      {
         intX += 1.0;
      }
   }
   else
   {
      fracX = lookUp (fracX, posLogs);
      if (fracX == 1.0)
      {
         fracX = 0.0;
         intX += 1.0;
      }
   }
   logX = intX + fracX;
   return (exp10x (logX));
}
```

The **RoundUp** function is essentially the mirror image of the **RoundDown** function. If the input value is negative, we call the **RoundDown** function with the negation of the input value and then return the negation of the result. If the input value is positive, then we compute its common logarithm and separate it into its integral and fractional parts.

If the fractional part is negative, we call the **lookUp** function, using the **negLogs** table, to find the first logarithm whose value is larger than the input fraction.

If the return fraction is **0.0**, then we increment the integral portion of the resulting logarithm. If the fractional part is positive, we call the **lookUp** function, using the **posLogs** table, to find the first logarithm whose value is larger than the input fraction.

If the result that is returned has the value **1.0**, then we set its fractional part to **0.0** and increment its integral part by **1**. The final step is to combine the new integral and fractional components and take the value returned by the exponential function as our final result.

lookUp Function Code

The **lookUp** function searches the specified table of logarithms from beginning to end, looking for the first value that is larger than the input value. The value found is returned. The code for the **lookUp** function is as follows:

```
double  lookUp (double  log, double  *table)
{
    short       index;

    for (index=0; index < 10; index++)
    {
        if(log <= table[index])
        {
            return table[index];
        }
    }
    return table[9];
}
```

lookDown Function Code

The **lookDown** function searches the specified table of logarithms from end to beginning, looking for the first entry that has a smaller value than the input parameter. The code for this function is as follows:

lookDown
function code (beginning)

```
double  lookDown (double log, double *table)
{
    short       index;
    for (index=9; index >= 0; index--)
    {
        if(log >= table[index])
        {
            return table[index];
```

lookDown *function code (concluded)*

```
        }
    }
    return table[0];
}
```

This concludes the description of the additions and changes we have made to the **GraphWindow** module. The next section describes the new **ChartInfo** support class.

Adding New ChartInfo Code

In order to keep an object that contains the current settings for the **Chart** dialog and also support access to its information, we have created new source files called **ChartInfo.c** and **ChartInfo.h**. These new source files define a class called **CChartInfo**, that inherits its behavior from the TCL class **CObject**. The class declaration taken from the **ChartInfo.h** header file is as follows:

```
class CChartInfo : public CObject
{
public:
    chartInfo      chartSettings;
    void           IChartInfo(void);
    chartInfo      GetChartInfo(void);
    void           SetChartInfo (chartInfo theData);
    minMax         GetHScale(void);
    minMax         GetVScale(void);
    Rect           GetHData(void);
    Rect           GetVData(void);
    Rect           GetHLabel(void);
    Rect           GetVLabel(void);
    Rect           Range2Rect(StringPtr range);
    unsigned char  GC (StringPtr s, short *index, short len);
};
```

In the class declaration, there is a single instance variable called **chartSettings** that is of type **chartInfo**. This is a structure that is also defined in the **chartInfo.h** file, whose contents are shown on page 387. A new data type called **minMax** has also been specified:

```
typedef struct
{
   double   min;
   double   max;
}
minMax;
```

Defining the New CChartInfo Methods

The **CChartInfo** class implements the access methods shown in the preceding class declaration. These methods are used by both the **Chart** dialog and the various charting methods in the **GraphWindow** module.

IChartInfo Method Code

The **IChartInfo** method is responsible for initializing a new instance of the **CChartInfo** class. The initialization consists of storing default values into each of the fields of the **chartSettings** instance variable. The code is as follows:

```
void  CChartInfo::IChartInfo(void)
{
   chartSettings.modified = 0;          // unmodified to start with
   chartSettings.chartType = 177;       // cHorizontalBarViewID
   chartSettings.scalingType = 180;     // cAutomaticScaleViewID
   chartSettings.hMinScale[0] = 0;      // no contents
   chartSettings.hMaxScale[0] = 0;      // no contents
   chartSettings.vMinScale[0] = 0;      // no contents
   chartSettings.vMaxScale[0] = 0;      // no contents
   chartSettings.title[0] = 0;          // no contents
   chartSettings.hDataRange[0] = 0;     // no contents
   chartSettings.vDataRange[0] = 0;     // no contents
   chartSettings.hLabelCheck = 0;       // not checked
   chartSettings.hLabelRange[0] = 0;    // no contents
   chartSettings.vLabelCheck = 0;       // not checked
   chartSettings.vLabelRange[0] = 0;    // no contents
}
```

In the foregoing code, the **modified** field is initialized to 0, indicating that the settings have not yet been specified by the user. The **chartSettings** structure is set up initially with horizontal bar chart and automatic scaling selections; all of the other fields are set to 0.

GetChartInfo Method Code

The **GetChartInfo** method returns the current contents of the **chartSettings** structure. The code is as follows:

```
chartInfo CChartInfo::GetChartInfo(void)
{
   return  chartSettings;
}
```

SetChartInfo Method Code

The **SetChartInfo** method stores the specified settings into the **chartSettings** instance variable. The code is as follows:

```
void   CChartInfo::SetChartInfo (chartInfo theInfo)
{
   chartSettings = theInfo;
}
```

GetHScale Method Code

The **GetHScale** method converts the **hMinScale** and **hMaxScale** strings in the **chartSettings** variable into double-precision floating-point values in a type-**minMax** structure. The code is as follows:

```
minMax  CChartInfo::GetHScale(void)
{
   extended    minVal, maxVal;
   minMax      theScale;
   minVal = str2num (chartSettings.hMinScale);
   maxVal = str2num (chartSettings.hMaxScale);
   x80tox96 (&minVal, &theScale.min);
   x80tox96 (&maxVal, &theScale.max);
   return theScale;
}
```

The **GetHScale** method uses the SANE library **str2num** function to convert the strings to extended floating-point values. Then these are converted to double-precision values and stored into the structure, using the **x80tox96** SANE function.

GetVScale Method Code

The **GetVScale** method converts the **vMinScale** and **vMaxScale** strings in the **chartSettings** variable into double-precision floating-point values in a type-**minMax** structure. The code is as follows:

```
minMax  CChartInfo::GetVScale(void)
{
   extended   minVal, maxVal;
   minMax     theScale;

   minVal = str2num (chartSettings.vMinScale);
   maxVal = str2num (chartSettings.vMaxScale);
   x80tox96 (&minVal, &theScale.min);
   x80tox96 (&maxVal, &theScale.max);
   return theScale;
}
```

GetHData Method Code

The **GetHData** method converts the string representation of the horizontal data range stored in the **hDataRange** field of the **chartSettings** structure into a Rect structure, where the left and top members specify the starting column and row and the right and bottom members specify the ending column and row. The code is as follows:

```
Rect  CChartInfo::GetHData(void)
{
   return Range2Rect (chartSettings.hDataRange);
}
```

This method uses a utility **Range2Rect** method that will be described shortly.

GetVData Method Code

The **GetVData** method converts the string representation of the vertical data range stored in the **vDataRange** field of the **chartSettings** structure into a Rect structure, where the left and top members specify the starting column and row and the right and bottom members specify the ending column and row. The code is as follows:

```
Rect CChartInfo::GetVData(void)
{
   return Range2Rect (chartSettings.vDataRange);
}
```

The **GetVData** method also uses the **Range2Rect** method to convert the string into a Rect structure.

GetHLabel Method Code

The **GetHLabel** method converts the string representation of the horizontal label range stored in the **hLabelRange** field of the **chartSettings** structure into a Rect structure, where the left and top members specify the starting column and row and the right and bottom members specify the ending column and row. The code is as follows:

```
Rect CChartInfo::GetHLabel(void)
{
   return Range2Rect (chartSettings.hLabelRange);
}
```

GetVLabel Method Code

The **GetVLabel** method converts the string representation of the vertical label range stored in the **vLabelRange** field of the **chartSettings** structure into a Rect structure, where the left and top members specify the starting column and row and the right and bottom members specify the ending column and row. The code is as follows:

```
Rect CChartInfo::GetVLabel(void)
{
   return Range2Rect (chartSettings.vLabelRange);
}
```

Both the **GetHLabel** and **GetVLabel** method use the **Range2-Rect** method to convert their respective strings to the Rect structure form.

Range2Rect Method Code

The **Range2Rect** method is a utility called by the various data and label range access methods to convert the cell ranges specified in the **Chart** dialog to numeric column and row values. The beginning and ending column and row values are returned in the form of a Rect structure:

Range2Rect *method code (beginning)*

```
Rect  CChartInfo::Range2Rect(StringPtr range)
{
   short          len, i, col, row, OK = 0;
   Point          from, to;
   Rect           aRect;
   StringPtr      s;
   unsigned char ch;

   len = range[0];
   for(i=0, s=&range[1]; (ch=GC(s, &i, len)) == ' ' && i < len; )
   {
      // skip over blanks
      continue;
   }
   if(ch >= 'A' && ch <= 'Z' || ch >= 'a' && ch <= 'z')
   {
      col = (ch & ~0x20) - 'A';
      if((ch=GC(s, &i, len)) >= '0' && ch <= '9')
      {
         row = ch - '0';
         OK = 1;
         if((ch=GC(s, &i, len)) >= '0' && ch <= '9')
         {
            row *= 10;
            row += (ch - '0');
         }
         else if (ch == '.' && GC(s, &i, len) == '.')
         {
            from.h = col;
            from.v = row-1;
            OK = 2;
         }
         else
         {
            OK = 0;
         }
      }
      else
      {
```

Range2Rect
method code
(continued)

```
            OK = 0;
        }
    }
    else
    {
        OK = 0;
    }
    switch (OK)
    {
        case 0:
        {
            SetRect (&aRect, -1, -1, -1, -1);
            return aRect;
            break;
        }
        case 1:
        {
            from.h = col;
            from.v = row-1;
            if ((ch=GC(s, &i, len)) == '.' && GC(s, &i, len) == '.')
            {
                OK = 2;
            }
            else
            {
                OK = 0;
            }
            break;
        }
    }
    if (OK != 2)
    {
        SetRect (&aRect, -1, -1, -1, -1);
        return aRect;
    }
    OK = 0;
    if((ch=GC(s, &i, len)) >= 'A' && ch <= 'Z' || ch >= 'a' && ch <= 'z')
    {
        col = (ch & ~0x20) - 'A';
        if((ch=GC(s, &i, len)) >= '0' && ch <= '9')
        {
            row = ch - '0';
            OK = 1;
            if((ch=GC(s, &i, len)) >= '0' && ch <= '9')
            {
                row *= 10;
                row += (ch - '0');
            }
            else if (ch == '\0')
```

Range2Rect *method code (concluded)*

```
            {
                OK = 1;
            }
            else
            {
                OK = 0;
            }
        }
        else
        {
            OK = 0;
        }
    }
    else
    {
        OK = 0;
    }
    if(OK == 1)
    {
        SetRect(&aRect, from.h, from.v, col, row-1);
        return aRect;
    }
    else
    {
        SetRect(&aRect, -1, -1, -1, -1);
        return aRect;
    }
}
```

The **Range2Rect** method parses the string that contains a data range and creates a Rect structure in which the left and top members contain the beginning column and row of the range and the right and bottom members contain the ending column and row of the range. Ranges are specified as a pair of worksheet cells, where columns are labeled from **A** to **Z** and rows are numbered from **1** to **50**. A range consists of a cell number, followed by two periods and a second cell number (e.g., **A3..A9** or **B13..W13**). A range can only be a single column or row—never a rectangular group of columns or rows.

The method begins with a local variable called **OK** set to 0, indicating that the result is initially invalid. Only if the specified range meets the requirements of a proper range is the **OK** value set to 1. After the string is parsed, the **OK** value is tested, and if it is 1, the proper Rect structure is returned; otherwise, a Rect of {-1, -1, -1, -1} is returned.

GC Method Code

The **Range2Rect** method uses the **GC** method to fetch the next character from the range string. The code for this method is as follows:

```
unsigned char CChartInfo::GC(StringPtr s, short *index, short len)
{
    unsigned char ch;

    if (*index < len)
    {
        ch = s[(*index)++];
    }
    else
    {
        ch = '\0';
    }
    return ch;
}
```

If the end of the string is reached, the **GC** method returns a binary 0 character. Otherwise, the next character in the string is returned, and the string index is incremented in preparation for the subsequent call.

Exercises

1. Explain the necessity of initializing the Chart dialog with its previous settings. Describe how the **DoChart** function performs this task.

2. Explain the operation of the **DoCommand** method in the **CChart** class. In what way does the method respond to the user's selections in the dialog? In what way does the appearance of the dialog change when each of the different chart types is chosen, in turn? What happens if automatic versus manual scaling is selected?

3. What is the purpose of the **Validate** method in the **CChart** class? What principle feature of object-oriented programming does the definition of this method illustrate?

4. Describe the operation of the **DrawHBarChart** method. What challenges does it face, and how are these handled with respect to the choice of automatic or manual scaling? How could the automatic scaling be improved?

5. The charting methods use the current size of the **GraphWindow** to determine the optimal drawing area. In what cases is this preference modified?

6. How could the charting methods be improved to use a varying number of divisions on the "value" axis? In what way would this improve the usefulness of the charts? Implement your suggestions.[1]

7. Could the X-Y chart be modified to provide a useful line-chart capability? What would be required to implement this feature for arbitrary cell values in a range? (*Hint*: Most line charts are drawn with increasing values when viewed from left to right or bottom to top.)

8. If the charting functions of the Ensemble application are modified to support pie charts, what are the major problems that you would face in annotating the charts? What modifications would be necessary to integrate the new features into the code described in this chapter? Implement these modifications.[2]

---

1. Modifying the number of divisions in a chart is a task of medium complexity. It will require some additional work, but should not be a major task. It could be assigned as an extra-credit project.
2. Implementation of pie charts is a very extensive task. It will require good knowledge of the built-in Quickdraw facilities for drawing wedges and will require the implementation of complex techniques for labeling the wedges. It could be assigned as an extensive extra-credit project.

9. The exercises in Chapters 8 and 11 indicate that you should consider the implementation of multiple contiguous or noncontiguous cell selections in the worksheet. How would implementation of this feature relate to the charting function? What modifications to the design and implementation of the charting function would facilitate interfacing with the worksheet module?[1] (*Hint*: Charting a range of preselected cells is a somewhat standard practice in major spreadsheet applications.)

1. Implementation of a more direct interface between the worksheet and the charting functions is a complex task. It should only be undertaken as an extensive extra-credit project.

Chapter 15

Printing Ensemble's Windows

This chapter describes the new custom code that we have added to allow the user to print the contents of the Notebook, Worksheet, or Chart windows.

The THINK Class Library implements a method to print the contents of the pane whose handle is stored in the **itsMainPane** instance variable in the **CEnsembleDoc** class. There is no direct provision for printing other, subsidiary, window panes.

In the TCL, the **CPrinter** instance manages the communication between *a document* and the Macintosh print manager. Each document can have its own printer object, but there can only be one printer object per document. Because the Ensemble application is a single-document multiple-window model, we need to provide some additional code to permit printing the additional window panes.

The remainder of the chapter discusses the custom code that has been added to allow the user to print the contents of the frontmost window of the Ensemble application. By activating each window, in turn, the contents of the Notebook, Worksheet, and Chart windows can be printed.

Printing the MainWindow's Pane

The **CMainWindow** instance defines a window that contains a single **CEditText** pane. This window is directly owned by the **CEnsembleDoc** class, which is a direct descendant of the **CDocument** class in the TCL. Thus, it is very easy to print the contents of the window's main pane (called **itsMainPane**). At the time the user has activated the **MainWindow**, the Ensemble application has the dynamic structure shown in Figure 15-1.

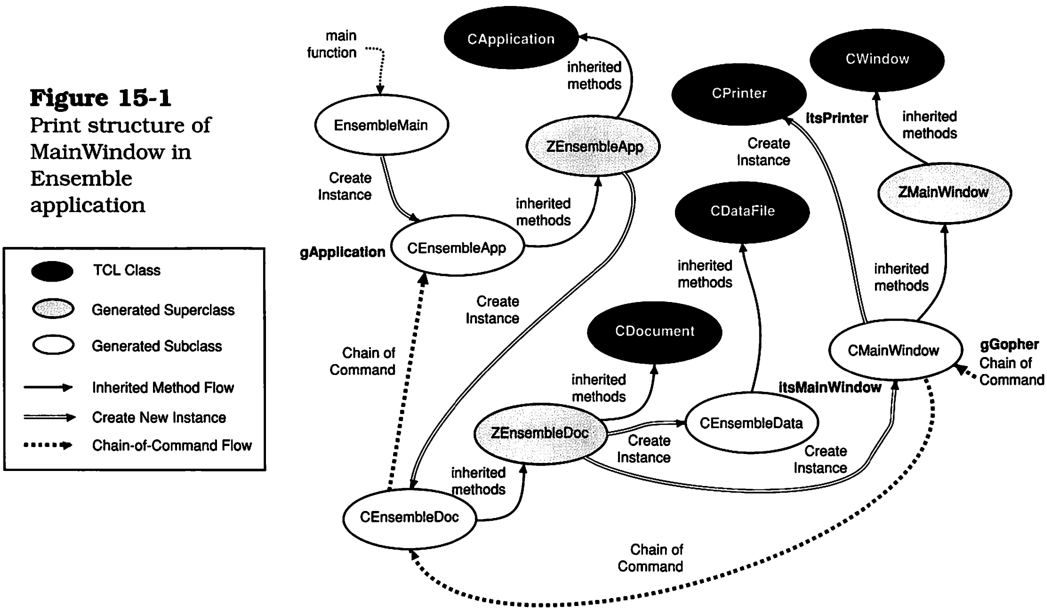


Figure 15-1
Print structure of MainWindow in Ensemble application

Notice that the other two windows aren't shown. For all practical purposes, they are not included in the chain of command when the **MainWindow** is active. All menu commands are directed to the current **gGopher**, which points to the instance of **CMainWindow**. Therefore, the **DoCommand** method in that class will be the first to receive any commands from the user, including the **Print** and **Page Setup** commands in the **File** menu. The code in Ensemble's **MainWindow** module doesn't currently handle these commands, so they are passed on in the chain of command to be handled by the **CDocument** class in the TCL.

Before the **CDocument** class will handle either the **Print** or **Page Setup** commands, an instance of **CPrinter** must have been created and stored into the document's **itsPrinter** instance variable. This is accomplished as a side effect of the creation of the **CEnsembleDoc** class, in the **ZEnsembleApp CreateDocument** method. By passing the value TRUE as the second parameter to the **IEnsembleDoc** method (see page 32), the document's **printable** instance variable will be set to TRUE, and during the execution of the inherited **IDocument** method, an instance of **CPrinter** will be created and stored in the document's **itsPrinter** instance variable.

The one remaining task to enable the text contained in the **MainWindow** to be printed out is to set the document's **itsMainPane** instance variable. Previously, AppMaker's default-generated code set the **CMainWindow** instance's **itsMainPane** to NULL in the **IZMainWindow** method (see page 71). The document's **itsMainPane** instance variable is set directly from the value contained in the corresponding variable in the **MainWindow** class, as shown in the listing of the **BuildWindows** method on page 365. To rectify the situation of the **itsMainPane** being NULL in the **CMainWindow** class, we have added one line of code in the **IMainWindow** method:

```
void  CMainWindow::IMainWindow(CDirector    *aSupervisor,
                               CEnsembleData  *theData)
{
   itsData = theData;
   inherited::IZMainWindow (aSupervisor);
   itsMainPane = Field3;
}
```

The code for the revised **IMainWindow** method puts the handle to the **EditText** pane (**Field3**) into the **itsMainPane** instance variable, from which the **BuildWindows** method can extract the value and store it into the document's **itsMainPane** instance variable.

When the preceding simple modification has been made, and the application is recompiled, the user will be able to print the contents of the **CMainWindow** class's **EditText** pane. The TCL contains all of the logic to accomplish this task. The following steps enumerate the actions taken by the relevant code:

1. The user chooses the **Print** command from the **File** menu, and this command is sent to the **CMainWindow** class's **DoCommand** method. This method does not need to handle the command, so it passes it on to its inherited method.

2. The **CDocument** class intercepts the **Print** command (**cmdPrint**) and verifies that the **itsPrinter** instance is not NULL. The method then sends a **DoPrint** message to the correct **itsPrinter** instance.

3. The **DoPrint** method calls the toolbox **Print Manager** to show the print job dialog. The user is given the opportunity to set the page range, number of copies, and other parameters that are provided in this standard dialog. If the user dismisses the dialog with **Cancel**, the print process is terminated at that point. Otherwise, the **DoPrint** method calls the **PrintPageRange** method (in **CPrinter**) to print out the necessary number of pages for the current document's **itsMainPane**.

4. The first act of the **PrintPageRange** method is to send an **AboutToPrint** message to the document (**CEnsembleDoc** in this case). This message contains the first and last page numbers to be printed, as determined from the print job dialog in the previous step. In the absence of an override method, the **AboutToPrint** method in the **CDocument** class is called.

    a. The **AboutToPrint** method calculates the **pageHeight** and **pageWidth** instance variables, checks the beginning and ending page numbers for consistency, and calculates the **pageCount** value.

    b. Then, an **AboutToPrint** message is sent to the **itsMainPane** instance. This enables the pane to perform any necessary initialization prior to commencing the print operation.

5. The **CPrinter** class's **PrintPageRange** method continues by calling the toolbox **PrValidate** function to verify that the printing information is valid and that a printer is attached to the system. If a valid result is returned, the method calls the **ResetPagination** method to clear out any previously set horizontal and vertical strip counts for the document. Then, if no pagination has previously been performed, a **Paginate** message is sent to the document.

a. The **CDocument** class's **Paginate** method gets the existing print record (print job settings) and recalculates the **pageHeight** and **pageWidth** values.

b. Then, the **Paginate** method sends a **Paginate** message to the document's **itsMainPane** instance, allowing the pane to paginate its contents according to the type of information contained in the pane.

c. In lieu of an override method, the **Paginate** message is intercepted by the method with the same name in the **CAbstractText** class. This method calls upon the **CEditText** class's **GetNumLines** method to determine the number of lines of text in the current EditText buffer. Using the **GetNumLines** method, **Paginate** calls the **CPrinter** class's **SetVertPageBreak** method with appropriate vertical positions for pages to be broken, creating a set of horizontal strips (pages) containing complete lines of text. No pages are broken with partial lines (e.g., a line split horizontally within its characters between pages). If a line will not fit on a page, the entire line is moved to the next page on which it will fit.

6. The **PrintPageRange** method continues by opening the print manager and calling the toolbox **PrOpenDoc** routine to initialize a printer grafPort and make it the active port. Then the method enters a loop, where the following actions are taken:

a. The toolbox **PrOpenPage** is called to prepare the printer grafPort to receive the QuickDraw instructions to print the current page.

b. The document is sent a **PrintPageOfDoc** message, which the **CDocument** class intercepts and sends on as a **PrintPage** message to the **itsMainPane** instance.

c. The **CEditText** class has a **PrintPage** method that switches the EditText port to the current (printer) port, expands the size of the port to encompass an entire page width and height, calls the inherited **PrintPage** method, and then resets the EditText port back to the screen port.

d. The inherited **PrintPage** method is found in the **CPanorama** class. This method sends the **CPrinter** instance a message to get the area (a Rect) associated with the current page, scrolls the pane to the beginning position of the page, and then calls the **DrawAll** method in the **CPane** class.

e. The **DrawAll** method in the **CPane** class calls the **Draw** method for the current pane and all of its subpanes (of which there are none for the EditText pane). The **Draw** method in the **CEditText** class calls **TEUpdate** to cause the text at the current pane position to be redrawn.

7. The loop concludes in the **PrintPageRange** method of the **CPrinter** class by calling the toolbox **PrClosePage** routine to perform any post-page-printing actions.

8. When all of the pages in the document's **itsMainPane** have been printed, the **PrintPageRange** method calls the **PrCloseDoc** toolbox routine to finish printing the last page of the document, closes the printer port, and ends the printing task. The **ClosePrintMgr** method in the **CPrinter** instance is called to ensure that the printing-oriented controls and variables are reset. Then, the method sends a **DonePrinting** message to the document, which, in turn, sends that message to the **itsMainPane** instance.

9. The **CEditText** class intercepts the **DonePrinting** message to recalculate the EditText pane dimensions, reset the coordinates, and activate the EditText pane (which will redisplay the blinking cursor in that pane).

10. The **PrintPageRange** method returns to the **DoPrint** method, which, in turn, returns to the **CDocument** class's **DoCommand** method, completing the execution of the **Print** command.

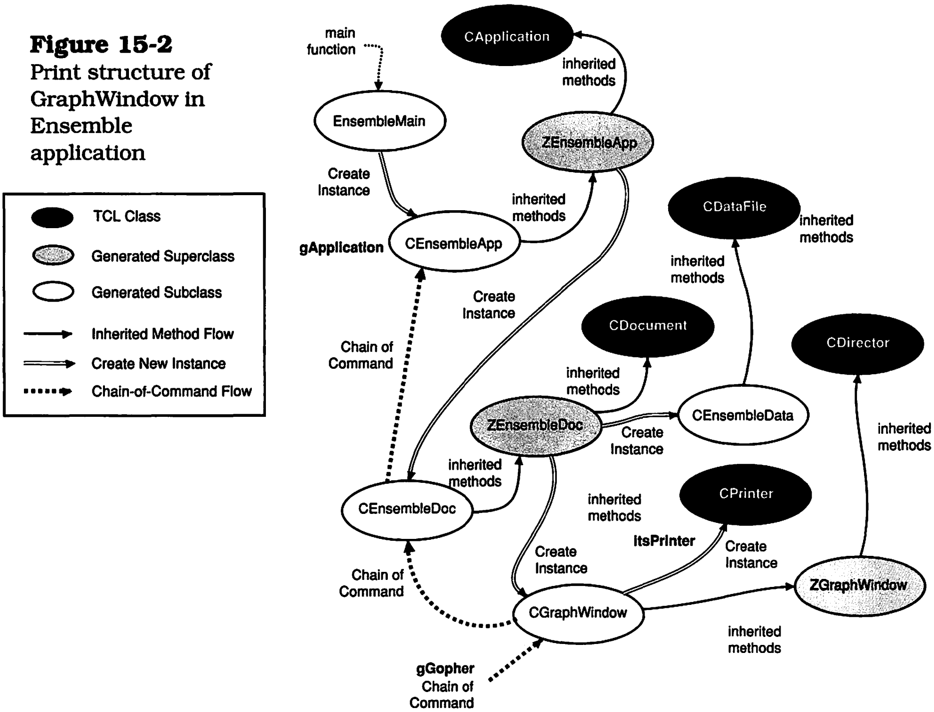
If the printer is connected via AppleTalk and the print monitor or other spooler is active, the process just described will be accomplished in a very short time. After all of the pages have been drawn, the Ensemble application will be ready to accept any further commands.

Printing the GraphWindow's Pane

This section describes the custom code modifications that we have made to support printing the charts drawn in the **CGraphWindow** window.

The THINK Class Library is not designed for printing other than a single main pane, so to print the panes in the additional windows, we have to "lie" to the TCL about which pane is the document's **itsMainPane** and also about which instance of **CPrinter** is **itsPrinter**. Fortunately, the code to circumvent the design of the TCL is quite straightforward. When the **GraphWindow** is the frontmost window, the dynamic structure of the Ensemble application appears as displayed in Figure 15-2.

Figure 15-2
Print structure of GraphWindow in Ensemble application



The figure shows that **CGraphWindow** and its superclass, **ZGraphWindow**, inherit their behavior from the TCL's **CDirector** class. There is no code in the TCL that provides the ability to directly print a window owned by a **CDirector**. This is not the case for the **MainWindow**, which is owned by the document.

The first step in providing support for printing the main pane in the **CGraphWindow** class is to define two new instance variables called **itsMainPane** and **itsPrinter**, respectively. The **itsPrinter** instance is created and initialized in the **IGraphWindow** method. The **itsMainPane** instance variable will be initialized later. The new **IGraphWindow** code is as follows:

```
void  CGraphWindow::IGraphWindow(CDirector     *aSupervisor,
                                 CEnsembleData  *theData)
{
    Str255   theFilename;

    itsData = theData;
    inherited::IZGraphWindow (aSupervisor);
    gDecorator->StaggerWindow (itsWindow);

    // any additional initialization for your window
    itsChartInfo = new CChartInfo;
    itsChartInfo->IChartInfo();
    itsPrinter = new CPrinter;
    itsPrinter->IPrinter((CDocument *)aSupervisor, NULL);

    //
    // Put the window's name in the title
    //
    if(((CEnsembleDoc *) aSupervisor)->itsFile != NULL)
    {
        ((CEnsembleDoc *) aSupervisor)->itsFile->GetName(theFilename);
        itsWindow->SetTitle(theFilename);
    }
}
```

The next step in supporting printing for the main pane in the **CGraphWindow** class is to add some special code to the **DoCommand** method. Fortunately, when it is active, the **CGraphWindow** is at the head of the chain of command. This means that its **DoCommand** method will receive the **Print** and **Page Setup** commands first (because the **gGopher** points to the **CGraphWindow** instance).

If you inspect the customized **DoCommand** method for the **CGraphWindow** class, as presented on page 406, you'll see that it contains only a case to invoke the **Chart** dialog and a default case that calls the inherited method in the **ZGraph-Window** superclass. You can see that this method merely passes on the command, as shown in the listing on page 367. What we need to do is intercept the **Print** and **Page Setup** commands inside the **DoCommand** method in the **CGraph-Window** class. The revised code for this method is as follows:

```
void  CGraphWindow::DoCommand(long theCommand)
{
   switch (theCommand)
   {
      case cmdChart:
      {
         DoChart(this);
         if(itsChartInfo->chartSettings.modified)
         {
            User4->Refresh();
         }
         break;
      }
      case cmdPageSetup:
      case cmdPrint:
      {
         PrintChart((CEnsembleDoc *) itsSupervisor, theCommand);
         break;
      }
      default:
      {
         inherited::DoCommand (theCommand);
         break;
      }
   }
}
```

Notice that in the new version of the **DoCommand** method, both the **cmdPageSetup** and **cmdPrint** commands are being intercepted. In both of these cases, we call a new method called **PrintChart**, with arguments of the **CGraphWindow's** supervisor (which happens to be the **CEnsembleDoc** instance, as you can see from the cast) and also the command that is to be executed. The code for the new **PrintChart** method is as follows:

```
void  CGraphWindow::PrintChart (CDocument  *itsSupervisor
                                           long  theCommand)
{
   CPane        *savedPane;
   CPrinter     *savedPrinter;

   savedPane = itsSupervisor->itsMainPane;
   savedPrinter = itsSupervisor->itsPrinter;
   itsSupervisor->itsMainPane = User4;
   itsSupervisor->itsPrinter  = itsPrinter;
   itsPrinter->ResetPagination();

   inherited::DoCommand (theCommand);

   itsSupervisor->itsMainPane = savedPane;
   itsSupervisor->itsPrinter  = savedPrinter;
}
```

The **PrintChart** method is the key to printing additional windows using the current structure of the TCL. The method accesses the **itsMainPane** and **itsPrinter** instance variables in the **CEnsembleDoc** instance **(itsSupervisor)** and temporarily saves these values in the **savedPane** and **savedPrinter** variables. It then sets the document's **itsMainPane** to the **User4** instance, which is the instance of **CPanorama** in which the charts are drawn. The document's **itsPrinter** is replaced by the **CGraphWindow's** instance of **CPrinter** (created in the **IGraphWindow** method, shown on page 460). The **itsPrinter** variable is sent a **ResetPagination** message, to clear out any horizontal or vertical strips that may previously have been set.

After the preceding preparations are complete, we can call the inherited **DoCommand** method, which will pass the **Print** or **PageSetup** command through the chain of command, until it is intercepted by the **CDocument DoCommand** method, as described in step 2 on page 455.

The operation of the TCL with regard to handling the **Print** and **Page Setup** commands is almost identical to that previously described, up to the point where the **Paginate** and **Draw** methods are called. In the case of the **CPanorama** methods that handle the **User4** instance of **itsMainPane**, pagination will depend on the size of the panorama. We have purposely defined its dimensions to correspond to a letter-sized page (see page 407).

The existing **CUser4 Draw** method will draw the currently selected chart into the printer port, according to the code provided in the TCL for switching the printer and screen ports when necessary.

After the **Print** or **Page Setup** command has been handled, control will return to the **DoCommand** method in the **CGraphWindow** class, immediately following the call to the inherited **DoCommand** method. This is key, because upon return of control, the code restores the values of the document's **itsMainPane** and **itsPrinter** to the values saved in **savedPane** and **savedPrinter**, respectively. At this point, the printing operation is complete.

Printing the CalcWindow's Pane

The provisions to print the **CCalcWindow** class's main pane are only slightly more elaborate than those shown for printing the **CGraphWindow's** pane. When the **CCalcWindow** is frontmost, the dynamic structure of the Ensemble application is as shown in Figure 15-3. As was the case with the **CGraphWindow** class, the first step in supporting printing of the **CCalcWindow's** main pane is to create two new instance variables named **itsMainPane** and **itsPrinter**. The **itsPrinter** variable is created and initialized in the **ICalcWindow** method. Because the majority of this code is identical to the listing of the **ICalcWindow** method in Chapter 11, only the initial portion of the code is shown:

ICalcWindow *method code (beginning)*

```
void  CCalcWindow::ICalcWindow(CDirector      *aSupervisor,
                               CEnsembleData  *theData)
{
   Rect        aRect;
   Str255      theFilename;
   long        index;
   CCellData   *aStyle;
   cellInfo    cellStyle;

   itsData = theData;
   inherited::IZCalcWindow (aSupervisor);
   gDecorator->StaggerWindow (itsWindow);
```

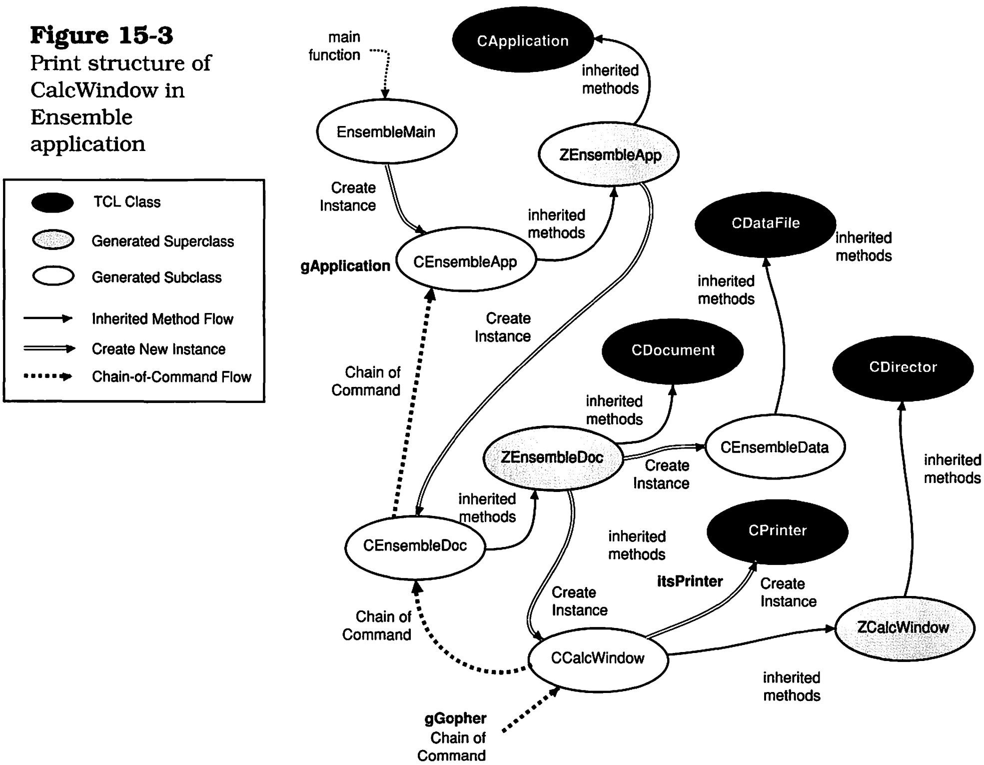
***ICalcWindow* method code (concluded)**

```
    itsPrinter = new CPrinter;
    itsPrinter->IPrinter((CDocument *)aSupervisor, NULL);

    //
    // REMAINDER OF METHOD'S CODE
    //

}
```

Figure 15-3
Print structure of CalcWindow in Ensemble application



The creation of the **CPrinter** instance and its storage into the **itsPrinter** instance variable are shown in this abbreviated listing of the **ICalcWindow** code. To handle the **Print** and **Page Setup** commands, we have enhanced the **DoCommand** method in the **CCalcWindow** class. In addition, we have had to override a couple of the methods in the **CList15** class (the class that implements the **CArrayPane** instance that contains the worksheet data). The code for the **AboutToPrint** and **DonePrinting** override methods will be presented shortly.

The new code for the **DoCommand** method is very similar to what was presented for the **CGraphWindow** class's method. The previous listing of the **DoCommand** method is shown in Chapter 11, and only a few lines have been added to implement printing. Only these lines are shown in the following code:

DoCommand *method code (beginning)*

```
void  CCalcWindow::DoCommand (long theCommand)
{
   Cell          aCell;
   short         height;
   short         width;
   short         changeStyle;
   cellInfo      styleInfo;
   cellInfo      oldStyle;
   CWSEntry      *anEntry;
   long          param;

   switch (theCommand)
   {
      case cmdEnterButton:
      {
         DoEnterButton ();
         break;
      }
      case cmdCancelButton:
      {
         DoCancelButton ();
         break;
      }
      case cmdPageSetup:
      case cmdPrint:
      {
         PrintWS((CEnsembleDoc *) itsSupervisor, theCommand);
         break;
      }

      case cmdWorksheet:
      {
         //
         // ALL THE CODE INSIDE THIS CASE HAS
         // PREVIOUSLY BEEN SHOWN IN CHAPTER 11
         //
      }
```

***DoCommand** method code (concluded)*

```
        default:
        {
            inherited::DoCommand (theCommand);
            break;
        }
    }
}
```

The foregoing listing includes the addition of a case to handle both the **Print** and **Page Setup** commands. The code calls a new method called **PrintWS**, with arguments of **itsSupervisor** (which is the **CEnsembleDoc** instance, as can be seen from the cast) and **theCommand** (which is the command to be executed). The code for the new **PrintWS** method is as follows:

```
void CCalcWindow::PrintWS (CDocument    *itsSupervisor,
                                        long   theCommand)
{
    CPane       *savedPane;
    CPrinter    *savedPrinter;

    savedPane = itsSupervisor->itsMainPane;
    savedPrinter = itsSupervisor->itsPrinter;
    itsSupervisor->itsMainPane = List15;

    //
    // don't print the worksheet borders
    //
    List15->SetColBorders(0, patCopy, ltGray);
    List15->SetRowBorders(0, patCopy, ltGray);

    itsSupervisor->itsPrinter = itsPrinter;
    itsPrinter->ResetPagination();

    inherited::DoCommand (theCommand);

    itsSupervisor->itsMainPane = savedPane;
    itsSupervisor->itsPrinter = savedPrinter;

    //
    // reset the worksheet borders
    //
    List15->SetColBorders(1, patCopy, ltGray);
    List15->SetRowBorders(1, patCopy, ltGray);
    List15->Refresh();
}
```

As was shown in the listing of the **PrintChart** method (on page 462), the first act of the **PrintWS** method is to save the current contents of the document's **itsMainPane** and **itsPrinter** instance variables into the **savedPane** and **savedPrinter** local variables. It then sets the document's **itsMainPane** to the **List15** instance (which is the **CArrayPane** instance that holds the worksheet display).

Because we don't want the cell borders to show up on the printout, we send commands to the **List15** instance that set the column and row border widths to 0. This effectively suppresses printing of the borders because the TCL checks their widths to determine whether they should be drawn. Next, the document's **itsPrinter** instance variable is replaced by the **itsPrinter** instance created in the **ICalcWindow** method, and a **ResetPagination** message is sent to the instance to clear any previously defined horizontal or vertical page strip settings. At this point, the inherited **DoCommand** method can be called to initiate printing of the pane. All of the code previously described regarding printing of the **CMainPane** instance (beginning with step 2 on page 455) is carried out for the **Print** or **Page Setup** command. The description of the printing process indicates (in step 4) that the **PrintPageRange** method calls the document's **AboutToPrint** method, which, in turn, calls the corresponding method in the **itsMainPane** instance (described in step 4a on page 456). In the case of the **CCalcWindow** pane, we have supplied an override for the **AboutToPrint** method. The code for this override is as follows:

```
void CList15::AboutToPrint (short *firstPage, short *lastPage)
{
   inherited::AboutToPrint (firstPage, lastPage);
   saveHOrigin = hOrigin;
   saveVOrigin = vOrigin;
   Offset(hOrigin, vOrigin, FALSE);
}
```

We need to override the **AboutToPrint** method because the worksheet pane (**List15**) isn't located at the top left corner of the window. Printing its contents in the current orientation will cause the cells to be offset both horizontally and vertically. To correct this situation, we have defined two new instance variables in the **CList15** class that will hold the existing horizontal and vertical pane origin values. The **AboutToPrint** code first calls the inherited method, then stores the **hOrigin** and **vOrigin** values into the new **saveHOrigin** and **saveVOrigin** instance variables, and then sends an **Offset** message to the pane to move it physically to the top left corner of the window's frame. By passing FALSE as the third argument to this method, we prevent the pane from being redrawn in its new position; therefore, the screen display is not changed.

The **Paginate** method for the worksheet pane is inherited from the **CTable** class. The method will break up the worksheet into horizontal and vertical strips that contain an integral number of whole cells.

Drawing the contents of the worksheet is accomplished by a combination of the **PrintPageRange** and its subsidiary methods, as well as by code in the **DrawCell** and **DrawWSCell** methods for the **CList15** class. The TCL takes care of switching the ports between the printer port and screen display port, as required.

When the worksheet's contents are completely printed (and it should be noted that only the portions of the worksheet that actually contain data will be printed), the **PrintPageRange** method calls the document's **DonePrinting** method, which, in turn, calls the corresponding method for the **itsMainPane** instance. These actions are described in step 8, on page 458. In our case, an override method in the **CList15** class has been provided for the **DonePrinting** method. The code is as follows:

```
void   CList15::DonePrinting (void)
{
   inherited::DonePrinting();
   Offset(-saveHOrigin,  -saveVOrigin, TRUE);
}
```

The **DonePrinting** method first calls the inherited method and then moves the **List15** pane back to its original location, using the values saved in the **saveHOrigin** and **saveVOrigin** instance variables. We also provide a third argument of TRUE to the **Offset** method, which causes the pane to be redrawn.

When the **DonePrinting** method returns to the TCL and all of the other cleanup tasks are complete, control returns to the **PrintWS** method in the **CCalcWindow** class (see page 466). Control returns to the code immediately following the call to the inherited **DoCommand** method. The code that follows restores the **itsMainPane** and **itsPrinter** instance variable values in the **CEnsembleDoc** instance and then restores the 1-pixel cell borders by calling the **SetColBorders** and **SetRowBorders** methods for the **CList15** object. The final action is to send a **Refresh** message to the pane, to redraw the worksheet with its borders.

The printed version of the worksheet consists of a series of vertical and horizontal strips, each of which occupies a printed page, as shown in Figure 15-4.

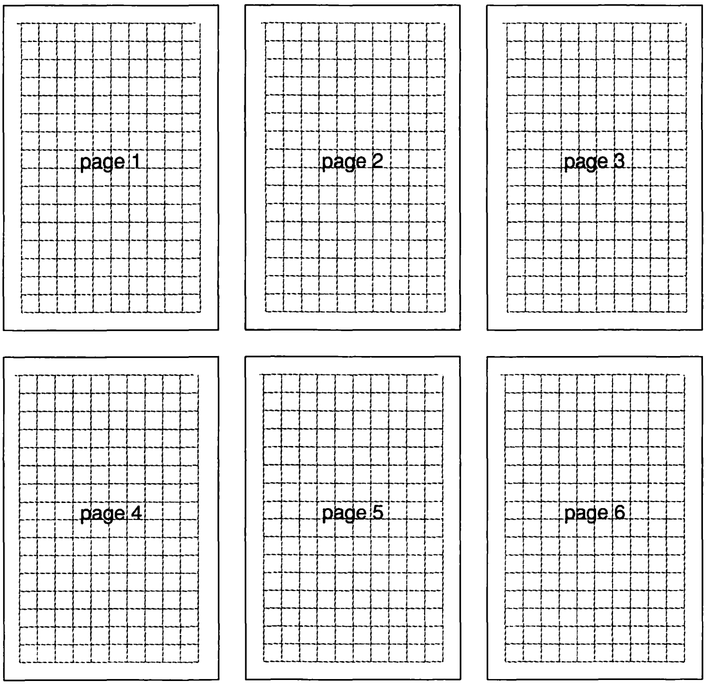


Figure 15-4
Pagination of the **CList15** worksheet

According to the pagination algorithm, the worksheet is divided into page-sized groups of cells. Pages can have a variable number of cells, depending on the row and column widths in the worksheet. The pages are printed from left to right and from top to bottom. The top left cell on page 1 is cell A1.

If the worksheet consists only of a few cells, it is advantageous to locate them beginning in cell A1 and in cells in that region of the worksheet. By defining cells in the rightmost columns or bottommost rows, a set of blank pages might be printed before the desired data are output.

There is no provision for printing a portion of the worksheet in this design. As indicated, if the worksheet contains only a few cells and the cells are placed in the top left area of the worksheet, only a single page will be printed.

The printing strategy presented for the **CGraphWindow** and **CCalcWindow** panes will apply equally to additional windows if these are added to the Ensemble design. In every case, it will be necessary to intercept the **Print** and **Page Setup** commands, so that the **itsMainPane** and **itsPrinter** variables in the document instance can be replaced by corresponding variables for the window whose pane is to be printed. It is also necessary to restore the saved values after printing is complete.

Exercises

1. Describe the difference between printing the contents of the MainWindow and of the subsidiary windows. Discuss how this difference could be eliminated. (*Hint*: Perhaps a different application model would be required, or perhaps the printing and command-handling methods could be insulated from the programmer, as they are when printing the MainWindow.)

2. Explain why the worksheet pane had to be moved so that it would print with its top left cell at the top left margin of the page.

3. Describe what modifications would be necessary to print only a portion of the worksheet window. In what way could this relate to the selection of multiple contiguous or noncontiguous cells in the worksheet, as suggested in Chapters 8 and 11?

4. How would you handle printing a title on the first page of a printout and page numbers or other header or footer material? What TCL classes and methods would you need to override, and what additional methods would need to be written?[1]

1. Examination of the complexities of pagination and printing individual pages is an involved task. Preparation of an appropriate answer to this question would require a great deal of research. We suggest that this be assigned only as an extra-credit project or be undertaken as a section of the lesson plan. Printing is both a complex and necessary task. Many applications will require formatted printed output, as opposed to the printouts that mimic the screen displays implemented in the Ensemble application.

Chapter 16

Completing the Ensemble Application

This chapter describes the final steps that will transform the Ensemble application into a stand-alone double-clickable Macintosh application.

We will discuss a couple of simple code additions that assign a Creator code to the application and also specify a Type code for the application's data file.

We will also provide a tutorial on using Apple's ResEdit program to create the custom resources that provide unique icons for both the application and its data file.

Finally, we will compile the Ensemble project one more time and link it into a stand-alone application. Once that has been done, it will be available for use by double-clicking either the application or one of its data files.

Defining Ensemble's Creator and File Type Codes

As you are probably aware, all Macintosh files (including applications) have both a Creator code and a Type code. The Type code for an application is always '**APPL**'; however, its Creator code can be any four-character identifier that doesn't conflict with one used for another application.

We will use '**Nsbl**' for Ensemble's Creator code, as this code is not presently being used (as far as I can tell). In addition to the Creator code, we will need a unique Type code for the application's data file. We have decided to use '**Nsbf**' for this purpose. Therefore, the data files written and read by Ensemble, after we have made the changes to be described, will have a Creator code of '**Nsbl**' and a Type code of '**Nsbf**'.

The changes to implement these codes are very simple. They are made in the **CEnsembleApp.h** file, to establish the definitions for the **kSignature** and **kFileType** constants. These new definitions are as follows:

```
#define kSignature 'Nsbl'
#define kFileType 'Nsbf'
```

The definitions are used in various places in the application, especially in the **CEnsembleApp** class's **SetUpFileParameters** method:

```
void CEnsembleApp::SetUpFileParameters(void)
{
   inherited::SetUpFileParameters ();
   sfNumTypes = 1;
   sfFileTypes [0] = kFileType;
   gSignature = kSignature;
}
```

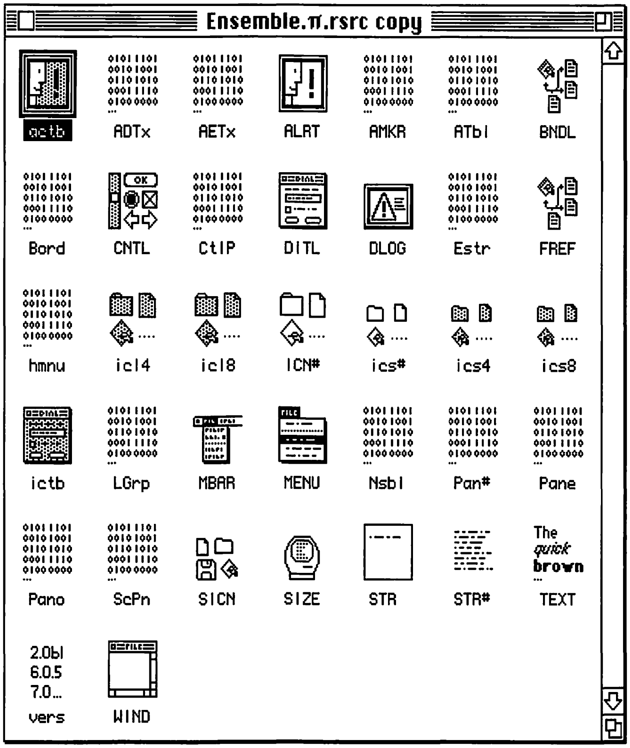
The **sfFileTypes** array is initialized to contain a single new Type code corresponding to the **kFileType** definition.

Creating Unique Application and File Icons

After the changes have been made to define the Creator and Type codes, the source code for the Ensemble application is complete and can be saved. The next step in creating a custom application is to design the family of icons that are displayed on the desktop for the application and its files, as well as other resources that establish the unique identity of the application. The best way to accomplish this is to make a copy of the **Ensemble.π.rsrc** AppMaker resource file and use the ResEdit application that came with your THINK C product. The following steps describe how to modify the file:

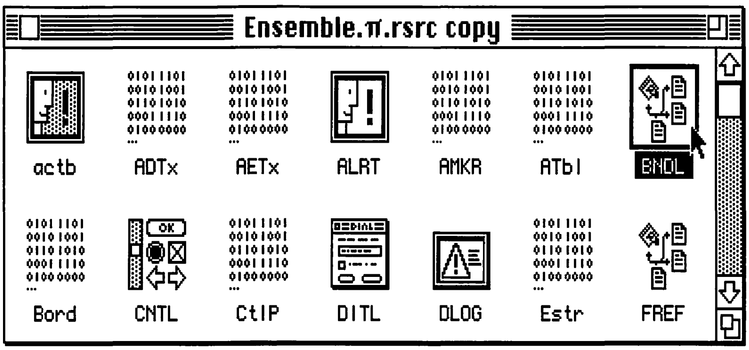
1. Launch your copy of ResEdit (version 2.1 or later) and open the **Ensemble.π.rsrc copy** file. You should see a window similar to what is shown in Figure 16-1.

Figure 16-1
Resources in the **Ensemble.π.rsrc copy**



2. Locate the resource icon labeled **BNDL**. This is the **Bundle** resource. Double-click to open this resource, as shown in Figure 16-2.

Figure 16-2
Opening the **BNDL** resource



3. You should see a list of **BNDL** resources, as shown in Figure 16-3. Notice that only a **BNDL** with a resource ID of 128 is shown.

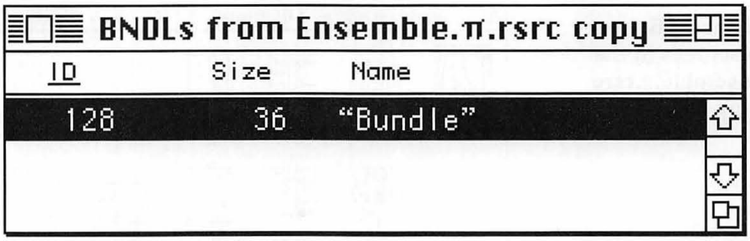


Figure 16-3
List of **BNDL** resources

4. Double-click on the **BNDL** whose ID is 128 to open it. You should see a window with the appearance shown in Figure 16-4. Notice that the bundle contains icons for a generic application ('APPL') and also a generic document ('DOC ').

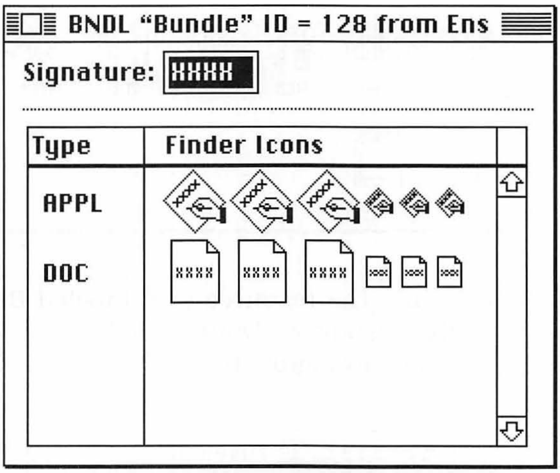


Figure 16-4
BNDL with ID = 128 open

5. Choose the **Extended View** command from the **BNDL** menu at the top of the screen, as shown in Figure 16-5.

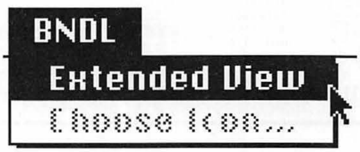
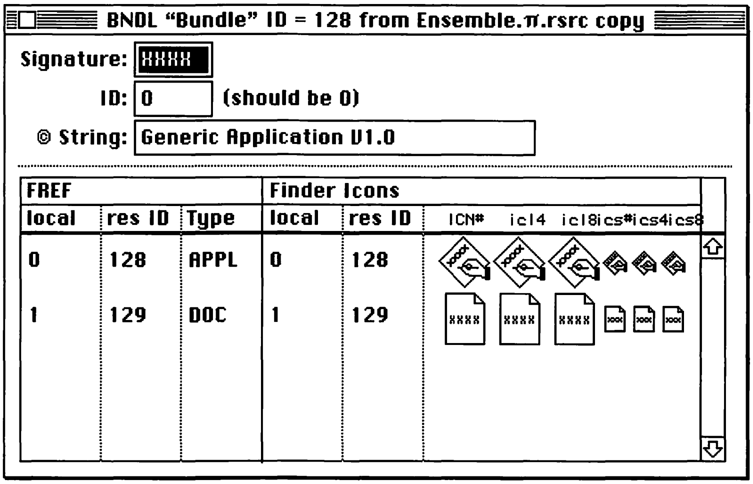


Figure 16-5
BNDL extended view selected

6. You should see an extended view of the **BNDL #128** resource, as shown in Figure 16-6. This view includes

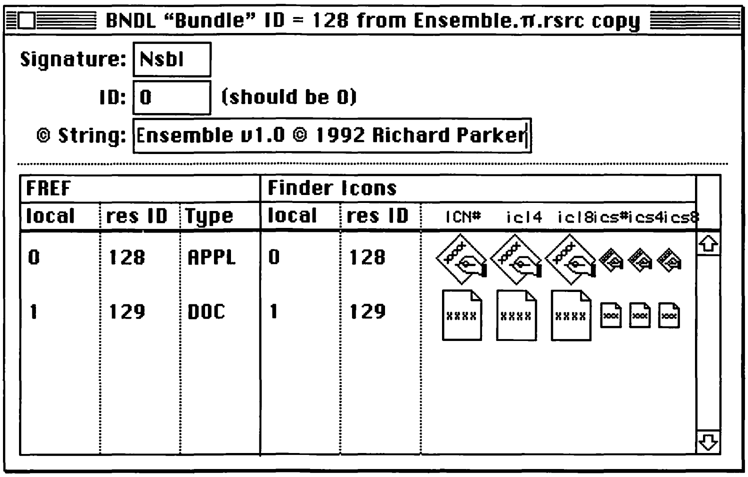
some additional fields that we must complete to set the application's signature (Creator code) and to create the version resource.

Figure 16-6
BNDL #128
extended view



7. Fill in the additional fields in the extended view to match the settings shown in Figure 16-7. The modified resource should have '**Nsbl**' in place of the '**XXXX**' in the **Signature** field, and the **ID String** should contain **"Ensemble v1.0 © 1992 Richard Parker"**, instead of the "**Generic Application v1.0**" text that was shown in Figure 16-6.

Figure 16-7
BNDL #128
extended fields modified



8. After the additional fields have been modified, choose the **Extended View** command from the **BNDL** menu, as shown in Figure 16-5, to collapse the view.

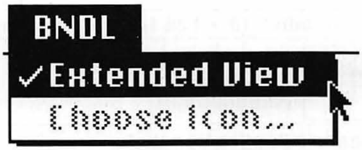


Figure 16-8
BNDL Extended View cleared

9. The next step is to open the icon editor for the APPL icons by double-clicking on the Finder Icons pane, as shown in Figure 16-9.

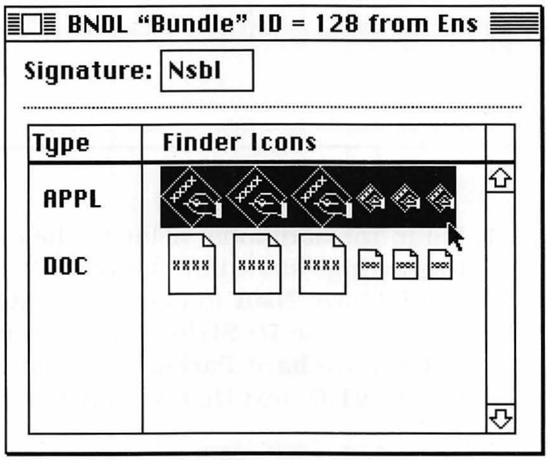


Figure 16-9
Opening **APPL** icon editor

10. The **APPL** icon editor is shown in Figure 16-10. This editor allows you to create application icons for both monochrome and color displays. The monochrome icon is called **ICN#**, and its small version is called **ics#**. It also allows the creation of large and small 8-bit color (**icl8** and **ics8**) icons, as well as 4-bit color (**icl4** and **ics4**) icons. (It is not possible to show the color versions of the icons in this book.) To begin the creation of the monochrome icon, choose the **ICN#** icon, as shown in the figure.

11. The next step requires that you select the entire icon in the large "Fat-Bits" window by choosing the **Select All** command from the **Edit** menu or by pressing the Command-A key combination. The entire default icon will be

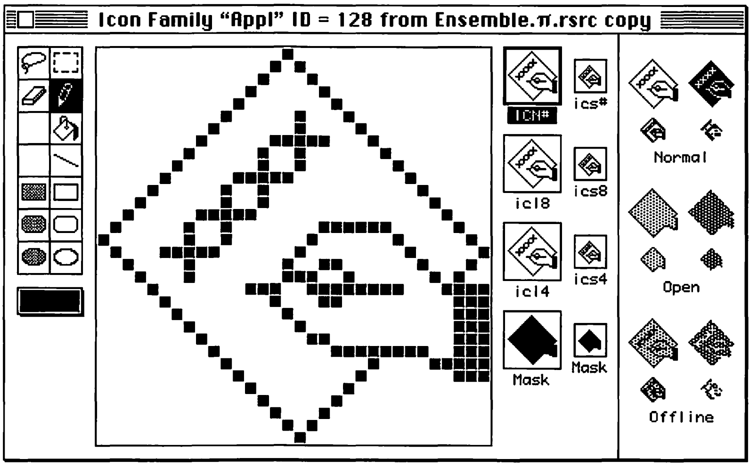


Figure 16-10
icon editor showing the default **APPL** icons

surrounded by a "marquee." Press the **delete** key to delete the default icon.

12. Select the open-circle tool from the tool palette, position the cross hair cursor at the top left corner of the square icon pane, and drag down and to the right until you reach the bottom right corner of the pane. When you release the mouse button, you should see an unfilled circle (a black circle with a white interior) in the pane. If this didn't work as you expected, you can delete whatever was drawn and try again.

13. The next step is to select the pencil tool and create the block "E" that occupies most of the interior of the icon. The final step in completing the icon is to create the mask. Click to select the mask, and delete its current contents, in the same manner as you deleted the default **ICN#** icon. Select the filled-circle tool, position the cross hairs at the top left corner of the large pane, and drag down and to the right until a filled circle occupies the interior of the pane. The completed result, with both the **ICN#** and the mask, is shown in Figure 16-11.

14. Figure 16-12 shows the completed versions of the other icons. The small monochrome icon (**icn#**) was created from scratch using the same techniques described for creating the large icon (**ICN#**) and its mask. To create the

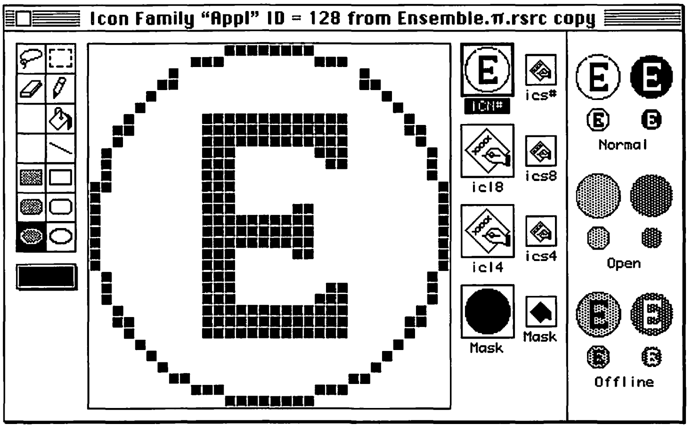


Figure 16-11
ICN# icon created

color versions of the icons, we copied the monochrome versions and pasted them into the corresponding large or small color icons. We changed the black circle to a shade of green by selecting the color from the color palette, and we used the pencil or bucket tools to change individual and a series of connected pixels, respectively. The block "E" was created in the same shade of green, and the interior of the circle was filled with a yellow color. The 8-bit and 4-bit icons use the same color scheme.

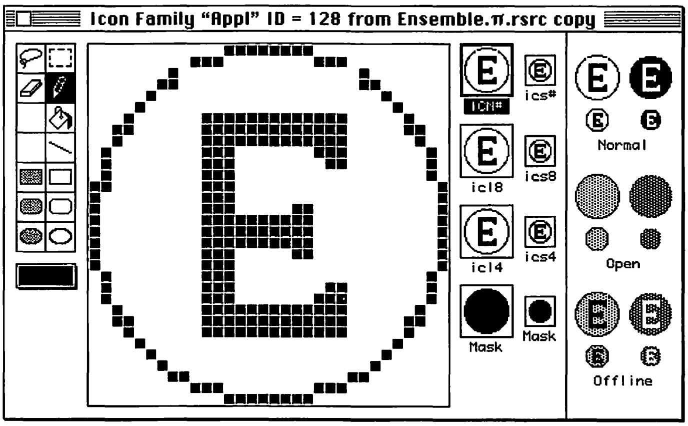


Figure 16-12
Completed family of application icons

15. When all of the application icons (or only the monochrome versions if you don't have a color monitor) are complete, you can dismiss the icon editor by clicking in its close box. The **BNDL** pane will reappear, but will contain the newly created icons for the **APPL** file type.

16. The next step is to create custom icons for the data file that is read and written by the Ensemble application. First, click in the space where the **DOC** type name is displayed, and change it to read **Nsbf**, as shown in Figure 16-13. Then reopen the icon editor by double-clicking on the Finder icons pane, as shown in the figure.

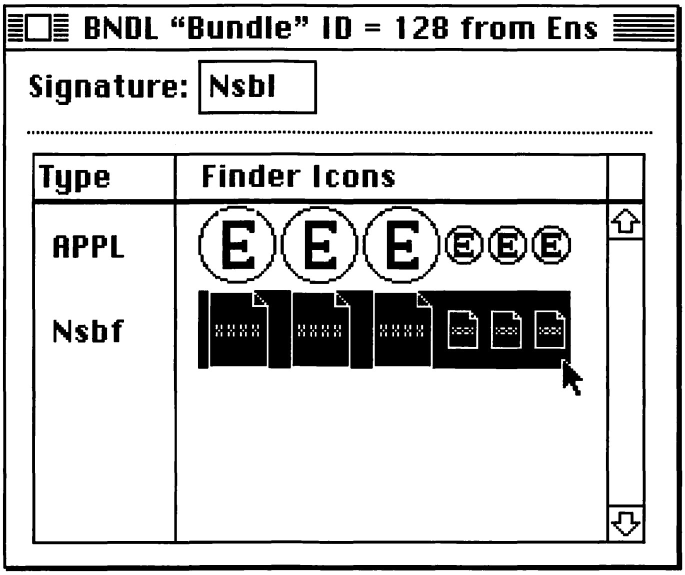


Figure 16-13
Nsbf icon editor being opened

17. Creating the icons for the **Nsbf** file type is carried out in much the same way as the steps previously described for creating the application icons. In this case, however, we don't want to delete the existing icons completely; rather, we merely want to delete the "**XXXX**" that appears in each and replace this with a block "E" that fills most of the document's outline. We will not have to change the mask for any of the new icons, because it already covers the complete outline of the document. To delete the X's, you can either select the pencil tool and click on each black pixel individually or use the marquee tool to select the entire group of pixels making up the "**XXXX**" image and then press the **delete** key. The completed set of document icons for the **Nsbf** file type is shown in Figure 16-14.

The color versions of these icons use the same green "E" and yellow background as in the application icon design.

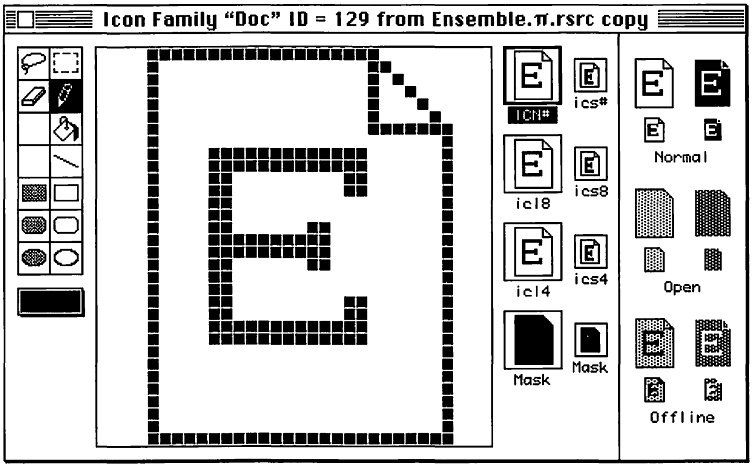


Figure 16-14
Nsbf icon designs complete

18. When the **Nsbf** designs are complete, you can dismiss the icon editor by clicking in its close box. The **BNDL** pane will now display both sets of completed icons, as shown in Figure 16-15.

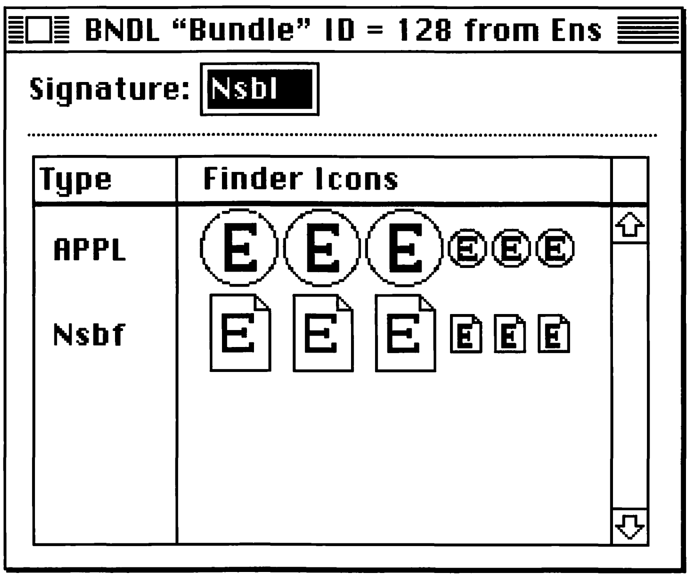
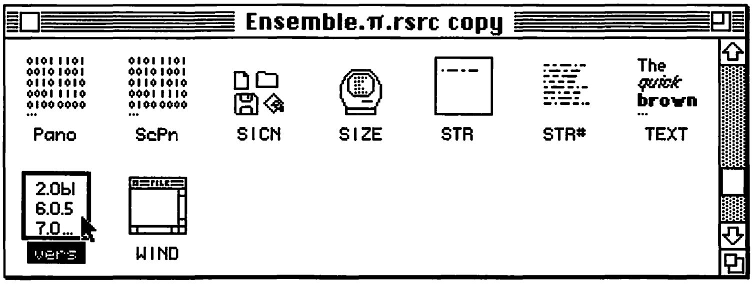


Figure 16-15
BNDL with all icon designs complete

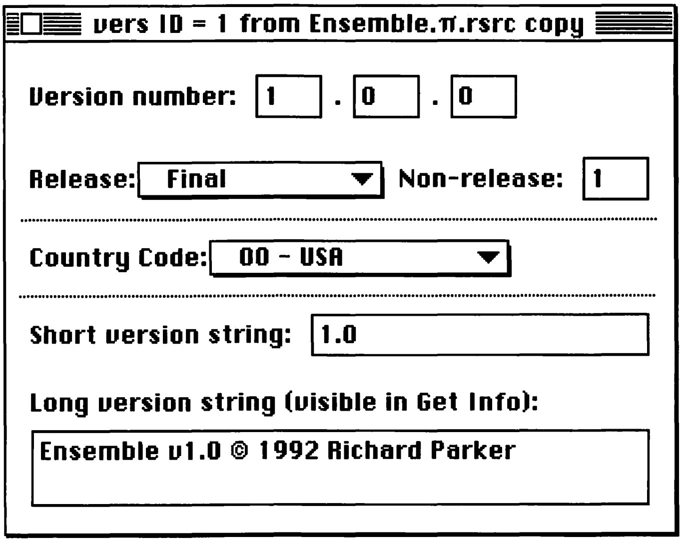
19. You can now close the **BNDL 128** pane and its corresponding list of **BNDL** resources. Next, double-click on the **vers** (version) resource to open it up, as shown in Figure 16-16.

Figure 16-16
Opening the **vers** resource



20. Modify the fields in the **vers** resource to match the settings in Figure 16-17. We've changed the version number to **1.0.0**, the release to **Final**, the short version string to **1.0**, and the long version string to "**Ensemble v1.0** © **1992 Richard Parker**".

Figure 16-17
Modified **vers** resource



21. When the **vers** resource has been modified as specified, you can close its window. The next step is to modify the **FREF** (File Reference) resource. Double-click on the

FREF icon, as shown in Figure 16-18. This displays the list of **FREF** resources.

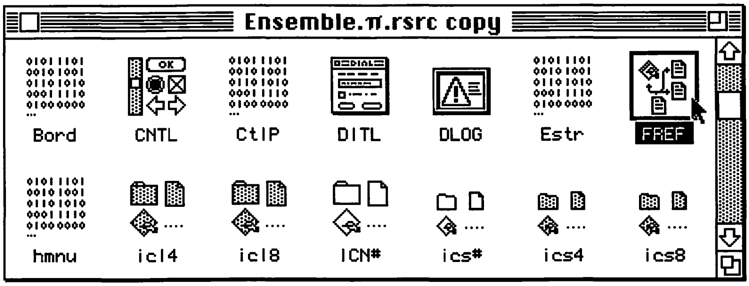


Figure 16-18
Opening the **FREF** resource

22. Choose the **FREF** resource **ID #129**, whose name is "**DOC**" in the list shown in Figure 16-19. Just click once on the list item to select it.

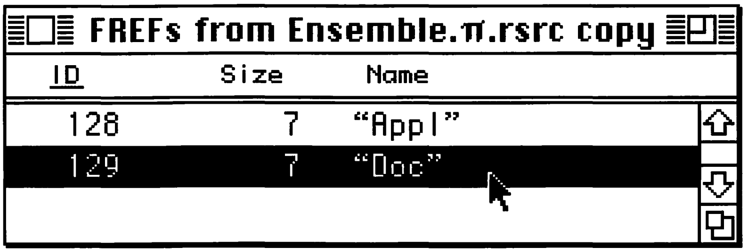


Figure 16-19
FREF 129 chosen

23. After selecting the **FREF ID #129** entry, pull down the **Resource** menu and choose the **Get Resource Info** command, as shown in Figure 16-20.

24. The "resource information" window will show the settings for the **FREF ID #129** selection, as shown in Figure 16-21. Change the Name field from **DOC** to **Nsbf**, as shown. Close the "info" window, and save the resource file by pulling down the **File** menu and choosing **Save**.

25. This completes the modifications to the AppMaker **Ensemble.π.rsrc copy** file. You can quit ResEdit by pulling down the **File** menu and choosing **Quit**.

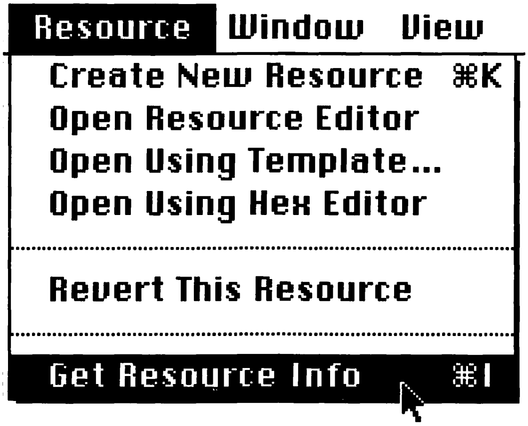


Figure 16-20
Choosing the **Get Resource Info** command

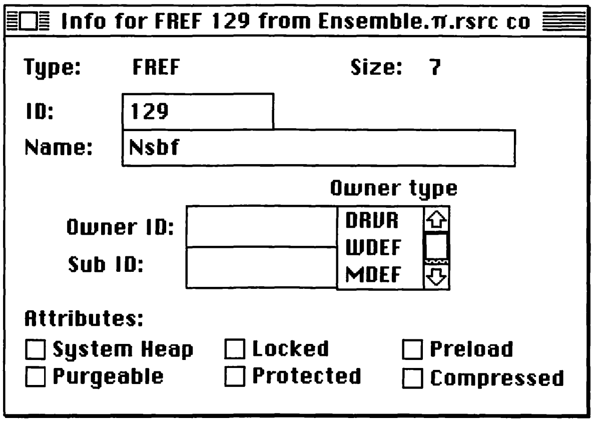


Figure 16-21
Name of **FREF ID #129** changed to **Nsbf**

Creating the Stand-alone Ensemble Application

If the modifications to the **Ensemble.π.rsrc copy** file were accomplished without any problems, you can throw the original **Ensemble.π.rsrc** file into the trash and rename the copy with the original's name. If you ran into any problems, you can throw the copy into the trash, make a new copy of the original and redo the steps in the previous section.

In this section, we are going to create the final stand-alone Ensemble application. To accomplish this objective, we perform the following steps:

1. Launch THINK C by double-clicking on the **Ensemble.π** project file.

2. Pull down the **Project** menu and choose the **Bring Up To Date** command. The compiler might not need to recompile the project. This is just a precautionary measure.

3. Pull down the **Project** menu and choose the **Set Project Type** command.

4. In the **Project Type** dialog, change the Creator code from **XXXX** to **Nsbl**, as shown in Figure 16-22, and then click the **OK** button.

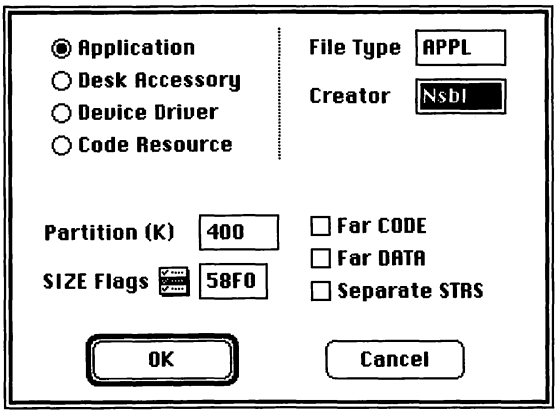


Figure 16-22
Changing the Creator to **Nsbl**

5. Pull down the **Project** menu and choose the **Build Application** command.

6. THINK C will display the **Build Application** dialog shown in Figure 16-23. Make sure that the file name for the application is **Ensemble**, and then click **Save**. THINK C will link and then write out the executable application file. When this is complete, you can quit the THINK C application.

Completing the Process

When you quit THINK C, as indicated in the previous section, you may find that the application does not yet display its new application icon. If this is the case, you can try closing the

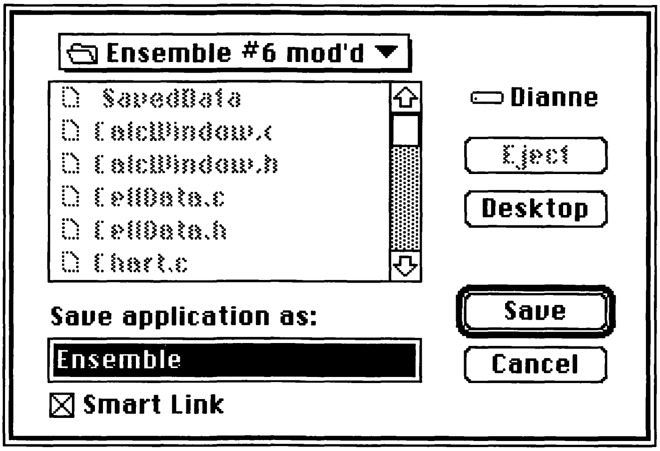


Figure 16-23
The **Build Application** dialog

folder in which it resides and then reopening it. If the icon still doesn't appear, you'll have to restart your computer and rebuild your desktop.

Rebuilding the desktop is simple. In the Finder, choose the **Restart** command from the **Special** menu. Then, immediately press and hold down both the Command and Option keys. Continue holding down these keys, until the operating system displays a dialog that asks you whether you really want to rebuild your desktop. Bear in mind that if you do, you will lose any comments you have keyed into the **Get Info** boxes for any of your files or applications. If you don't want to lose any of your comments, click the **Cancel** button, which will bypass the rebuilding process and restart the computer in the normal fashion. If you do choose to rebuild, click the **OK** button and the desktop will be rebuilt.

If you have multiple hard disks, you can choose to rebuild the desktop only on the disk that contains the Ensemble application. Clicking **Cancel** in the dialog requesting permission to rebuild another disk's desktop will not prevent a dialog for each disk from being shown. Click **OK** for the one that contains the Ensemble application.

When the process is complete, you should see the new icon for your Ensemble application. Figure 16-24 shows the complete set of files for the Ensemble project, including a file that contains the "Amazing Widgets" data file. The files are shown

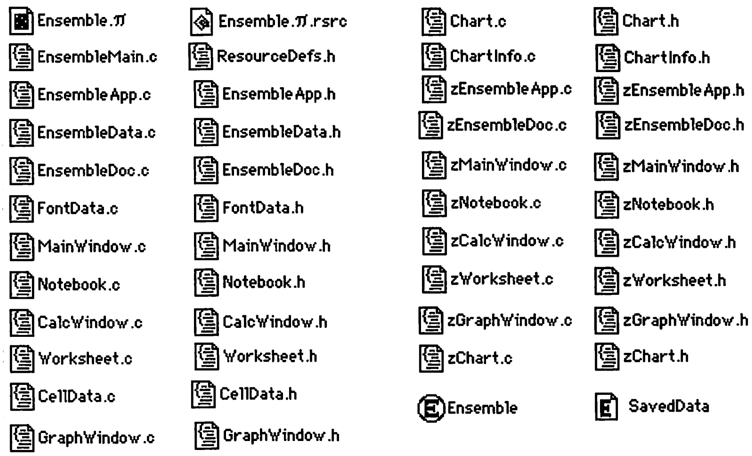


Figure 16-24
A complete set of files for the Ensemble application

in the "by Small icon" view. The application and its data file are shown at the bottom right of the figure.

If you want to see the application's icon as well as its version string, click on the application to select it, and then pull down the **File** menu and choose the **Get Info** command. A picture of the **Get Info** window is shown in Figure 16-25.

To change the Type and Creator codes of the existing "SavedData" file (so that it will be recognized by the Ensemble application), you should do the following:

1. Launch ResEdit, pulling down its **File** menu and then choosing the **Get File/Folder Info** command.

2. ResEdit will display a standard Open File dialog box, and you can navigate to the folder in which the "SavedData" file is stored, select the file, and click **Get Info**, as shown in Figure 16-26.

3. When the **Get Info** button is clicked, ResEdit will display a large dialog that contains many settings for the file. Change the file Type and Creator to **Nsbf** and **Nsbl**, respectively. Also, make sure that the "Inited" checkbox is not checked, as shown in Figure 16-27.

After these changes have been made to the file, you should quit ResEdit, saving the changes to the file, and the Finder

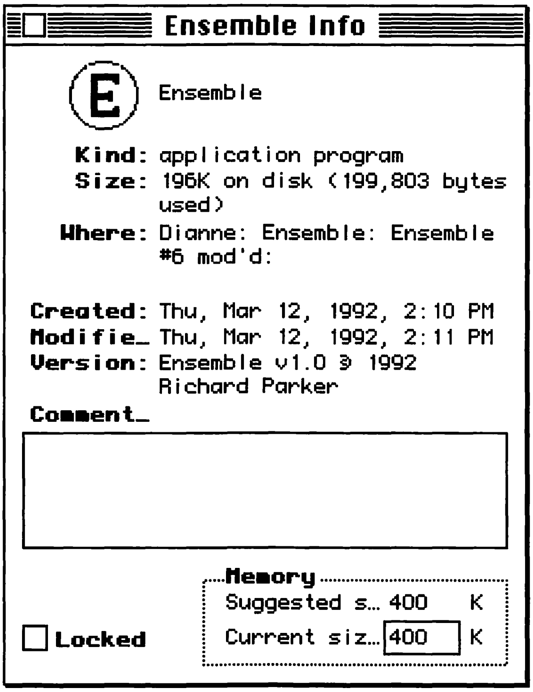


Figure 16-25 Contents of the Ensemble application's **Get Info** window

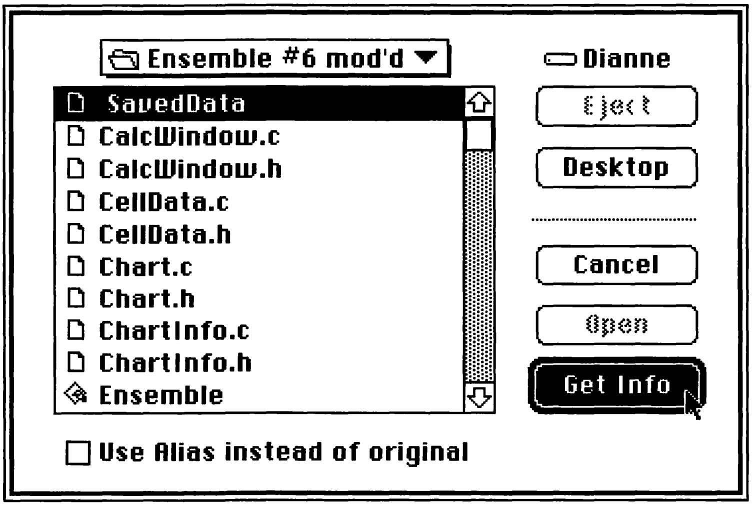


Figure 16-26 ResEdit's standard file dialog with **Get Info** selected

icon for the "SavedData" file should display the new icon. If it does not, you will once again have to rebuild the desktop file, following the method described at the beginning of this sec-

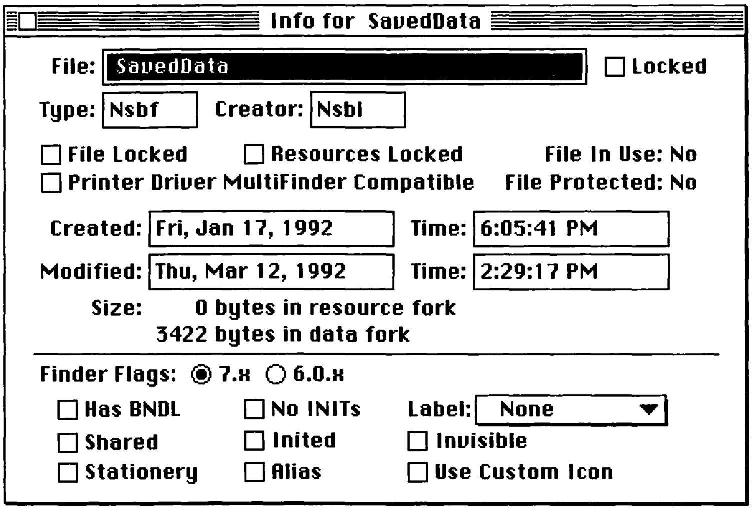


Figure 16-27
ResEdit Info for SavedData

tion. It is not necessary to change the information for the "SavedData" file, unless you want to open the file with the Ensemble application. If you don't change the file's Type and Creator codes, when you choose the **Open** command from Ensemble's **File** menu, the file will not be visible in the dialog. Only files with Creator and Type codes of **Nsbl** and **Nsbf**, respectively, will be seen.

If you change the Type and Creator codes of the "SavedData" file, you can launch the Ensemble application by double-clicking on the file.

If you don't change the Type and Creator codes, any subsequent files written by the application will automatically be assigned the correct codes and will display the unique icons.

Summary: Application Development

The Ensemble application illustrates many of the important design and programming considerations that go into the development of a nontrivial THINK C application. The combination of AppMaker and THINK C is very powerful and offers a streamlined approach to object-oriented programming.

While your own applications will differ from the specifics of the Ensemble application, many of the techniques used to create the user interface and to interface with both the generated code and the THINK Class Library will be very similar to those shown in this book.

One conclusion is sure: It is easily possible to create a complex application by using these tools in a series of incremental steps, with the ability to verify the functionality of the application at each stage of development. This alone should give you the desire to begin using AppMaker with THINK C for all of your application development projects.

Exercises

1. Describe what changes are needed to support additional file types in an application. What areas of the code are affected if multiple types of input data are supported?

2. Describe the function of Type and Creator codes. Where is the icon for a particular type of data file stored?

3. Define your own custom icon for the Ensemble application. Explain under what circumstances each of the **ICN#**, **icl4**, **icl8**, **ics#**, **ics4**, and **ics8** icons are used. What would happen if any of these were missing?

Index

A

D

E

F

G

N

O

P

R

W

X

Apple Macintosh 3-1/2" 800Kb disk
By Richard O. Parker

Prentice-Hall

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE OPENING THIS DISKETTE PACKAGE. OPENING THIS DISKETTE PACKAGE INDICATES YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THEM, YOU SHOULD PROMPTLY RETURN THE PACKAGE UNOPENED, AND YOUR MONEY WILL BE REFUNDED.

IT IS A VIOLATION OF COPYRIGHT LAW TO MAKE A COPY OF THE ACCOMPANYING SOFTWARE EXCEPT FOR BACKUP PURPOSES TO GUARD AGAINST ACCIDENTAL LOSS OR DAMAGE.

Prentice-Hall, Inc. provides this program and licenses its use. You assume responsibility for the selection of the program to achieve your intended results, and for the installation, use, and results obtained from the program. This license extends only to use of the program in the United States or countries in which the program is marketed by duly authorized distributors.

LICENSE

You may:

a. use the program;
b. modify the program and/or merge it into another program in support of your use of the program.

LIMITED WARRANTY

THE PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT PRENTICE-HALL, INC. OR ANY AUTHORIZED DISTRIBUTOR) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION.

SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS THAT VARY FROM STATE TO STATE.

Prentice-Hall, Inc. does not warrant that the functions contained in the program will meet your requirements or that the operation of the program will be uninterrupted or error free.

However, Prentice-Hall, Inc., warrants the diskette(s) on which the program is furnished to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your receipt.

LIMITATIONS OF REMEDIES

Prentice-Hall's entire liability and your exclusive remedy shall be:

1. the replacement of any diskette not meeting Prentice-Hall's "Limited Warranty" and that is returned to Prentice-Hall with a copy of your purchase order, or
2. if Prentice-Hall is unable to deliver a replacement diskette or cassette that is free of defects in materials or workmanship, you may terminate this Agreement by returning the program, and your money will be refunded.

IN NO EVENT WILL PRENTICE-HALL BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE SUCH PROGRAM EVEN IF PRENTICE-HALL OR AN AUTHORIZED DISTRIBUTOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY AN OTHER PARTY.

SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

GENERAL

You may not sublicense, assign, or transfer the license or the program except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign, or transfer any of the rights, duties, or obligations hereunder is void.

This Agreement will be governed by the laws of the State of New York.

Should you have any question concerning this Agreement, you may contact Prentice-Hall, Inc., by writing to:

Prentice Hall
College Division
Englewood Cliffs, N.J. 07632

Should you have any questions concerning technical support you may write to:

Richard O. Parker
1004 Magnolia Avenue
Modesto, CA 95350

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US THAT SUPERCEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

ISBN 0-13-092966-2

NOTICE TO CONSUMERS
THIS BOOK CANNOT BE RETURNED FOR CREDIT OR REFUND IF THE PERFORATION ON THE VINYL DISK HOLDER IS BROKEN OR TAMPERED WITH

BREAK AT SEAL AND PULL TO OPEN

Ensemble Application Source Files for
Easy Object Programming
for the Macintosh
Using AppMaker™ and THINK C™
by Richard O. Parker

A Simon & Schuster Company
Englewood Cliffs, N.J. 07632
ISBN 0-13-092966-2

DISK INCLUDED
DO NOT DEMAGNETIZE

$34.95

Easy Object Programming

for the Macintosh Using AppMaker™ and THINK C™

Richard O. Parker

It is an excellent book that builds from beginning to end a nontrivial application in an evolutionary manner, showing all the steps along the way.
—Spec Bowers, Bowers Development

The author presents a new approach that takes much of the drudgery out of object-oriented programming on the Mac.
—Kurt Schmucker, Apple-Advanced Technology Group

This in-depth exploration of object-oriented programming in C shows readers how a complex application can be easily created in a step-by-step manner using state-of-the-art Macintosh tools. Ideal for professionals familiar with the basic concepts of C programming, the THINK Class Library, and the fundamental concepts underlying O-O programming. This tour-de-force of application development uses the following object-oriented techniques:

- Describes the evolution of a complete, multipurpose, object-oriented application at various stages of object-oriented design and development.
- Provides data flow diagrams that illustrate the dynamic structure of the application at various stages of object-oriented design and development.
- Presents a great number of features in the THINK C Class Library with detailed descriptions and diagrams that illustrate the Library's structure and features.
- Contains a detailed examination of the AppMaker code and features tutorials on how to use AppMaker to produce the various user interface elements for the window dialog boxes as well as the menus used in the application.
- Includes a **Special Offer** on AppMaker and THINK C!

About the Author

Richard O. Parker has been an independent computer consultant for the past six years. He has written manuals and handbooks for high-tech computer companies such as Advanced Micro Devices, and has recently completed a user's manual for a major Macintosh programming tool.

PRENTICE HALL
Englewood Cliffs, NJ 07632

ISBN 0-13-092966-2

90000

9 780130 929662