Graphical User Interfaces

Tim budd

November 14, 1994

Abstract

This is a rough-draft for a proposed new chapter to be included in the second edition of my book "An Introduction to Object Oriented Programming", published by Addison-Wesley, 1991. It will probably appear between the current chapters 15 and 16. Constructive comments and criticism will be gratefully appreciated. (No date has been set for the second edition).

1 Introduction

Graphical User Interfaces have been popularized in recent years by systems such as the Macintosh, Windows 3.0, OS/2 Presentation Manager, Motif, Open Look, InterViews, and others. GUI's tend to be more flexible and intuitive than older character-based interfaces, and are thus easier to use. This is particularly true for novice computer users. Ironically however, developing applications that are easy for novice users to understand is a difficult and challanging task for the programmer. In this chapter we will explore some of the ways that object-oriented techniques can be used to simplify this task.

In large part graphical user interfaces have grown up hand-in-hand with object-oriented programming. The Smalltalk-80 system was made popular as much for its innovative interface as for its innovative language features. The fact that object-oriented techniques were used in the creation of this interface did much to popularize the idea of object-oriented programming in the mind of the computing public. The Smalltalk system also brought with it a new way of structuring programs; a technique called event driven programming.

2 Event Driven Execution

The first conceptual hurdle a programmer must overcome in attempting to understand the techniques used in developing graphical user interfaces is the idea of structuring programs around event-driven execution. A conventional program proceeds on its own agenda. We can think of execution as being something like

```
program main;
begin
   initialize application;
   while not finished do
      begin
         if user has done something interesting
            respond to it
      end;
   clean up application;
end.
```

Figure 1: The event driven style of programming

a finger that is moving through the program from beginning to end. At various points the program may interact with the user, but the nature of the responses the program is willing to deal with is very proscribed. For example a program may ask questions of the user, and respond only to simple yes/no answers typed at a keyboard.

In an event driven program, on the other hand, the user is in control. The program largely *reacts* to the user, rather than being *proactive*. In an event driven program execution is confined largely to a single loop. Figure 1 describes in pseudo-code the main program for almost all event-driven applications. The initialization and cleaning up stages start and terminate the application. Between these two extremes the program simply loops waiting for the user to do something, such as moving the mouse, hitting the keyboard, or inserting a disk into a floppy disk drive. When the user makes an action, the application reacts, then goes back to waiting for the next action.

The "interesting things" the user can do are called *events*. Since the flow of control is directed by the sequence of events, this is called event-driven execution. Not all events need to correspond to user initiated activities. Activities such as inserting a disk can cause a sequence of internally generated events, and the program will respond to these in the same way as to user generated events.

Object-oriented programming fits naturally with event-driven execution because the flow of control is so highly structured. It is relatively easy to create a library of classes which go through the motions of an event driven application, but perform no actual work. By subclassing from this library and changing the meaning of just a few methods, we can add the "work" part of an application to this library without needing to rewrite the "control" portions. This is exactly what is done by the systems we will describe in the remainder of this chapter.

Steve Burbeck of Apple [Wilson 90] has described a typical event-driven application built using a system such as the ones we will subsequently describe as being constructed using an "upside down library". In a traditional program a software library may provide bits and pieces of code, but it is the job of the programmer to write the "glue" that holds everything together. Using a GUI system the library provides the "glue", and the programmer merely specializes pieces of the library, using subclassing and overriding, to obtain application specific behavior. Because such systems define the fundamental structure of a program, they are sometimes referred to as *application frameworks.*

The primary means by which an application framework is customized to create a new program is through the use of inheritance. The application framework provides one or more abstract superclasses; classes that will be overridden by the programmer. For the purposes of explaining the mechanism used for customization, we can divide the methods defined in these classes into three categories:

1. Base methods. These provide certain essential functionality which is useful to the customization process, but will probably not be overridden by the programmer. An example might be a method which returns the coordinates of the cursor when a mouse-press occurred.

2. Template, or algorithm methods. These describe an abstract algorithm, for which the specific details are left to other methods. An example might be a method which implements the event loop of Figure 1. These methods provide the structure for the application framework, without doing any of the actual work. As with base methods, template methods are not usually overridden in a new application.

3. Abstract methods. These methods do the actual work of the application. In the application framework an abstract method is often simply a dummy procedure which performs nothing. By overriding the abstract methods, actions specific to a new application are provided.

For example, in the MacApp system we will describe later, a template method ReadFromFile is used to read information from a file. This method is defined in terms of several other methods, such as a base method which asks the user for the name of the file to open, a method which opens the given file, and an abstract method DoRead which reads the contents of the file. To specialize this process it is only necessary to change the meaning of the latter method.

Thus when building a new program using an application framework the programmer simply creates several new classes as subclasses of those provided by the application framework, overriding abstract methods to create application-specific behavior. The application framework plays the role of both the main program (the "top" part of an application) and often provides certain base functional (the "bottom" part of an application), leaving it to the programmer to fill in the details in the middle.

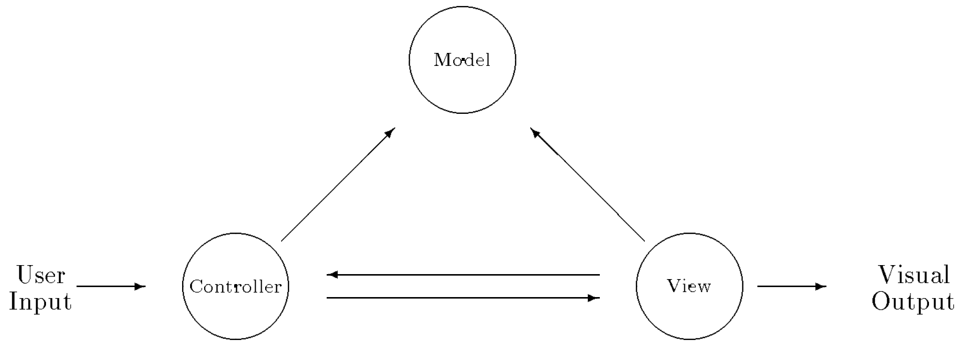

Figure 2: The Model-View-Controller Triad

3 The Smalltalk Model-View-Controller Triad

The ancestor of all application frameworks is found in the language Smalltalk-80. The Smalltalk graphical windowing system is based upon the concept of dividing an application into three categories, called the *model*, the *view*, and the *controller* (Figure 2). Each portion of the application has a specific task, and to accomplish that task each portion must interact with the other two.

The model component of the MVC triad contains the data representing the internal state of an application. The model can be an instance of any Smalltalk class; although there is a class Model that can be subclassed, it is not necessary to do so and many applications use other classes for models. For a text editor the model might be a string representing the text being manipulated; for a bar graph or pie chart it could be the data being displayed, and for a menu it might be a list of titles and associated actions. The distinguishing features of a prototypical model are that they perform some service or action, and that they know nothing of their visual appearance.

Mediating interaction between the user and the model are views and controllers. The view is responsible for the visual representation of a model on an output device, such as a display terminal. There may be several views of

initialize	set instance variables
model: aModel	establish model for view
controller: aController	establish controller for view
defaultController	return a controller for this view
containsPoint: aPoint	true if point is in view
borderWidth: aValue	set border width
borderColor: aColor	set border color
insideColor: aColor	set inside color
clear	erase view (including border)
clearInside	erase inside of view
flash	complement window twice in succession
display	redraw view

Figure 3: Messages understood by views

one model; for example portions of a spreadsheet might be displayed in a bar graph form, or a pie chart form, or as a list of numbers. A view can also have nested subviews; for example when a complex window has several individual components.

The most important message a view must repond to is the message display, which requests the visual presentation for the model be generated or updated. Views may represent an entire model or only certain aspects of the model. A view must know about the model it is representing, but need not have any knowledge concerning other views. A system supplied class View and associated subclasses provide a number of methods to assist in common activities such as drawing borders around windows, drawing title bars, and the like. A new view is almost always produced by subclassing an existing view class in the Smalltalk-80 library. Figure 3 shows a few of the messages understood by instances of the standard views.

Conventionally each view class maintains the model in an instance variable of the same name, as well as a variable containing the controller. A model, on the other hand, does not usually contain any direct connection to a view. This matches the idea that a model can exist with several views, and that these views may be created and destroyed without altering the model. This can also promote reusability of code that implements the model.

A controller class is used to guide user input. A system-wide controller traps user activity such as moving or depressing the mouse, or depressing a key at the keyboard. This system utility keeps track of all the controllers that have been created, and selects an appropriate controller for each activity, passing information to the controller in the form of a message.

As with views, controllers are usually created as instances of classes which already exist in the Smalltalk-80 library. The creation of a new controller is almost always accomplished as part of the initialization process during the creation of

isControlWanted	control inquiry
startUp	initialize control cycle
controlInitialize	initialize controller
controlLoop	cycle through control activities
isControlActive	control reinquiry
controlActivity	do control process
yellowButtonActivity	yellow button was pressed
redButtonActivity	red button was pressed
blueButtonActivity	blue button was pressed
controlTerminate	close down controller

Figure 4: Messages understood by controllers

a new view, and there is usually a one-to-one relationship between controllers and views. Standard system views, in fact, include a method (defaultController, see Figure 3) which returns an instance of the appropriate type of controller for the view. Figure 4 illustrates some of the methods defined in the standard control classes. Once a controller is given management of the system (after it has responded affirmatively to a isControlWanted message), it is in charge of responding to all activity until it relinquishes its management responsibilities.

The default behaviors defined for many of the methods shown in Figure 4 simplify the creation of new controllers. For example the method associated with the message startUp invokes the initialization and clean-up methods controlInitialize and controlTerminate, as well as starting the control loop. The control loop repeatedly asks whether control is still active (isControlActive) and if so performs the controlActivity. The default controlActivity checks for key or button presses, and if so invokes a further method. Thus a simple application may need only modify the meaning of a few messages, such as the method yellowButtonActivity. (The Smalltalk-80 classes display some of their historical traditions in not only assuming a three button mouse, but associating colors with each of the buttons).

A trace of several senarios may help in understanding the interactions between the three portions of the MVC triangle. Suppose we have an application with two windows. Each window will automatically become "active" when the cursor associated with the mouse is over the portion of the display controlled by the window. Thus when the mouse moves and no other controller is managing the system, the supervising controller will ask each of the two controllers associated with the views if they want control. The controllers will use the default method, which responds true if the cursor is within the boundary of the view. The method isControlActive will continue to return true as long as the mouse remains within the view. Moving the mouse outside of the window will cause isControlActive to return false, and the controller will relinquish management back to the system.

As part of the initialization (or activation) process for the controller the message display will be sent to the view. Just as each view maintains a direct link to its controller via an instance variable, each controller also maintains a direct link back to its view. As part of the display process, a view may query the model to obtain its current settings. The information obtained from the model is then used to create a graphic display.

Once the display is completed, action returns to the controller. Suppose now the user clicks the mouse in a manner intended to produce some change in the model. The controlActivity method will decipher the mouse click, and turn it into a message specific to the button pressed, say YellowButtonActivity. The user supplied subclass of Controller may then turn this command into an application specific message either to the view or directly to the model itself.

In this manner the model finally receives a message indicating a change should take place. For example the controller may have moved a line in a pie chart, which is translated into a message to the model to increase the percentage of some quantity. The model makes the appropriate modifications to its own internal state; and we reach a seemingly difficult quandary. How is the model to indicate to its associated views that they now need to update their displays? A fundamental principle of the MVC design is that models should be ignorant of the display. Yet surely the views must somehow be made aware of their need for modification.

The solution in Smalltalk-80 is provided through a very general mechanism of *dependencies*. A system provided dictionary is maintained of objects and their *dependents*. When a new view is created it registers its model in this dictionary and associates itself as a dependent. A single object can have many dependents, such as when a model has several views. When a model makes a significant change, it sends a message to this dictionary indicating that it has been updated. The dictionary then transmits an update message to each listed dependent. In this manner the dependents are alerted without any direct access from the model.

To summerize, the MVC triad separates the creation of a graphical user interface into three smaller components. Each of these components has a specific task, and interacts with each of the other members of the triad in a structured fashion. By providing example or template classes from which the user can subclass, the Smalltalk-80 standard class library makes the creation of new user interfaces relatively easy. A very complete description of the MVC model can be found in Volume II of the book Inside Smalltalk [LaLonde 91].

4 The Interactor Model

One complaint that is sometimes heard about the MVC model concerns the separation of input handling from output handling. Input handing in the MVC triad is the responsibility of the controller, while output handing is the pri-

```
int main (int argc, char ** argv) {

   World world("solitaire", argc, argv);

   game = new GameWindow;

   world.InsertApplication( new VBox(game,
      new HBox(new QuitButton, new StartButton)));

   world.Run();
}
```

Figure 5: Creating the main components of the solitaire application

mary responsibility of the view. Yet these two tasks are often intimately tied together, a fact reflected in the strong ties that bind these two portions of the triangle together. The InterViews system [Linton 89] simplifies the design of user interfaces by being designed around a single abstraction, the interactor.

An *interactor* occupies a rectangular region of an output device (such as a screen) and is responsible for managing the display in that area. The primary messages used to control the behavior of an interactor are Redraw, which is used to redraw a portion of the display, and Handle, which is used to catch input events. In practice the model portion of the interface remains, although its interaction is less structured than in the MVC triad. Thus we have replaced the Model-View-Controller triangle with an Interactor-Model pair.

Just as the Smalltalk-80 library provided a number of different types of ready made classes for controllers and views, the Interviews system provides a wealth of interactors. Common interactors include scenes (textual or graphical displays), buttons, and boxes. The latter permit interactors to be stacked either horizontally or vertically.

To illustrate, we will consider the Solitaire game studied in Chapter 10. The primary window is class GameWindow, which is declared as a subclass of monoscene. The interactor for the game window, however, shares the screen with two buttons. To create these, the two buttons are first placed next to each other in a horizontal box (or HBox). This box is then stacked vertically under the game window, using a vertical box (or VBox). The resulting application is then "inserted" into the variable world, which is an instance of World, the ultimate "controlling" class for InterViews (Figure 5). Once all the interactors for an application have been established, the application is set in motion by sending the message run to the world object.

To assist in the layout of interactors on the display device surface, every interactor maintains a shape value (an instance of the InterViews class Shape) in an instance variable. The shape of an interactor is not rigid; rather the shape

```
GameWindow::GameWindow(){

    // create a new shape for our drawing area
    shape = new Shape;
    shape→Rect(BoardWidth, BoardHeight);

    // create a new sensor so that we can catch mouse down events
    input = new Sensor;
    input→Catch(DownEvent);

    // create a new painter for drawing
    painter = new Painter;

    ...
}
```

Figure 6: Creating a new game window

value defines the optimum shape the interactor would desire, and instructions for how the shape can grow or shrink. These should be set in the constructor for the interactor, as part of the initialization process (Figure 6).

Each interactor also maintains an instance variable named input, which holds an instance of class Sensor. A sensor describes the type of input events (mouse motions or keypresses) the interactor is willing to handle. Only requested events will be passed on to the interactor.

Finally to produce a graphical output each interactor maintains a canvas. The canvas for an interactor can be manipulated, for example for writing or drawing lines, using an instance of class Painter. Most often this output takes place in response to a Redraw message, which is given as arguments four coordinate values describing the portion of the window to be updated. The solitaire application chooses to ignore these arguments, and instead simply redraws the entire screen (Figure 7).

The Handle message is produced whenever an input event is generated for which an appropriate sensor exists. An argument is passed along with the handle message. This argument is an *event record*, and defines the appropriate characteristics of the event. For example, the event record for a mouse down event records the coordinates of the location where the cursor was located when the mouse was pressed. An application class specifies meaning for these activities by redefining the methods associated with this message. Many of the system provided interactors turn a Handle into yet another message. For example, a Button is an interactor which turns a mouse press action into a call on the method Press. This method is then overridden to provide customized behavior for the button.

```
void GameWindow::Redraw(Coord a, Coord b, Coord c, Coord d)
{
    // first clear the entire playing area
    clearArea(0,0, BoardWidth, BoardHeight);

    // then display the card piles
    for (int i = 0; i < 13; i++)
        allPiles[i]→display();
}
```

Figure 7: The overridden method redraw in the class GameWindow

```
void GameWindow::Handle (Event& e)
{
    // we are only interested in mouse down events
    if (e.eventType != DownEvent) return;

    for (int i = 0; i < 13; i++)
        if (allPiles[i]→contains(e.x, e.y)) {
            allPiles[i]→select(e.x, e.y);
            return;
            }
    return;
}
```

Figure 8: The overridden method Handle in class GameWindow

In the Solitaire game described in Chapter 10 the GameWindow class redefined the message Handle so that it searches the piles of cards, discovering which pile (if any) was under the cursor at the time of a mouse click (Figure 8). If an appropriate pile is found the message select is sent to the pile along with arguments representing the position of the mouse click.

By combining input and output activies into one class, the InterViews class library simplifies the structure of graphical user interfaces. The interactor model, on the other hand, seems well suited for only relatively simple interfaces, and for applications in which the layout of the display remains relatively static during execution. In the next section we will examine a different approach to this problem.

5 The Application-Document-View Model

The Macintosh personal computer has arguably popularized the idea of a graphical user interface more than any other system. The Macintosh standard user interface, as described by Apple's *Human Interface Guidelines*, is flexible, intuitive, and relatively easy for novice users to understand. But this does not imply that it is easy to create new applications that fit the Macintosh ideal. In fact the Macintosh has a reputation for being an extremely difficult system to program.

Among the many problems that Macintosh programmers must face are the fact that programs in the Macintosh world can be initiated in any one of three different ways, each requiring a different response; they must be ready to deal with outside applications, such as desk accessories or the scrapbook; they must be ready to handle windows being moved and resized during execution; and under the multifinder and latest version of the operating system they must be ready at any time to be suspended and restarted. All this is complex, and makes developing a robust Macintosh application an extremely difficult task.

A number of software libraries, both object-oriented and conventional, have been created to help simplify the task of application development for the Macintosh. The central idea in all of these is to hide the details involved in dealing with window manipulation, desk accessories, menu manipulation, and the like and allow the programmer to concentrate on just the unique features of the application being created. Probaby the best known of such systems, and the one we will describe here, is the MacApp library of Object Pascal utilities [Wilson 90].

The prototypical Macintosh application is slightly different than the applications we have discussed in the previous sections. A typical Macintosh application is based around the idea of *documents*. A document, like the model portion of the MVC triad, contains the essential information used to describe the current state of an application. An example document might be the current file contents in a text editor, or the current canvas in a drawing program. However unlike the MVC triad, where we assumed we had a single model and many different views, it is assumed in the Macintosh world that an application may be working with several documents at the same time, and can easily move between them. Thus a text editor can work with several files at once; similarly a user can be drawing in several paintings at the same time.

A program is divided into three components. These are the *application*, the *document*, and the *view* (Figure 9). The application is the controlling part of the system. There is always just a single instance of the application class created in any program, a subclass of the system provided class TApplication.[1] The application initializes the system, and waits for the user to create a new document or request to work on an existing document. In the MacApp classes a new document is created using the method DoMakeDocument. The default methods in the MacApp library take care of managing the menu bar (including access to desk accessories and the scrap book), and handling of a few common methods, such as the "quit" method used to terminate the application. By over-

[1] The MacApp libraries uses an initial prefix to indicate the type of entity being named. Thus classes always begin with a "T", initialization methods with an "I", and constants with a "k", among others.

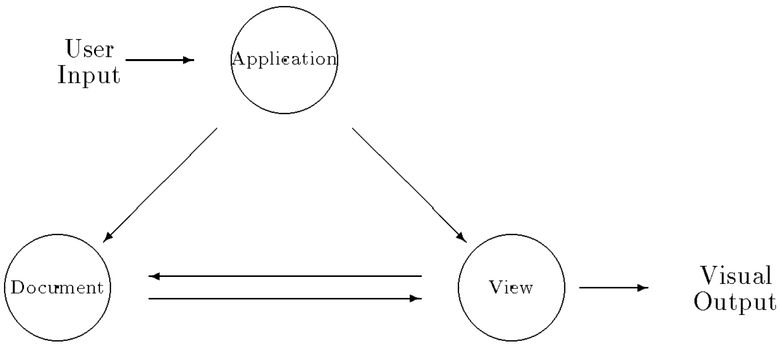

Figure 9: The Application-Document-View Triad

IApplication	initialize application
DoMakeDocument	create a new document

Figure 10: Messages understood by applications

DoInitialState	initialize document
DoRead	read in a document from disk
DoNeedDiskSpace	compute amount of disk to store document
DoWrite	store document
DoMakeViews	make views for document
DoMakeWindows	make windows for document
Free	release any storage for document

Figure 11: Messages understood by documents

riding further methods the user can add new menu items and other commands. Figure 10 shows the methods most commonly overridden by new applications.

A document, as we have noted already, contains the information describing the state of whatever objects the application is manipulating. A view provides the visual display of this information. A document must have at least one view, but can have more, just as a model could have multiple views in the MVC design.

The document is created by subclassing the class TDocument, and need only override the methods used to open, create and store documents, and methods used to respond to menu selections. Optional methods that can also be overridden control the undoing of actions and the printing of documents. Figure 11 shows some of the more common methods overridden by application documents.

MacApp distinguishes between a *window*, which includes a title, close and zoom boxes, and so on, and a *view*, which is simply the information being displayed. (In actual fact the corresponding classes are related, but it is easier from the design point of view to think of them as separate.) The simplest type of application will just use a standard window, and concentrate on customizing a special purpose view.

A new type of view is created by subclassing TView, and must redefine the methods used to draw the screen and respond to key presses and mouse clicks. If printing is permitted in the application the view is also usually responsible for the actual printing task. Figure 12 shows some of the more common messages understood by views.

A plethora of classes that we will not discuss also help simplify the tasks associated with handling menus, scroll bars, text editing, and other features commonly found in Macintosh applications.

We will illustrate MacApp by describing how an interface could be con-

Draw	redisplay view
DoMouseCommand	respond to mouse down
DoSetCursor	alter cursor when over view

Figure 12: Messages understood by MacApp views

structed to the Billiards game of Chapter 16.[2] We start by defining a new application class, which we shall call TBilliardApplication. Figure 13 shows the declaration of this class, as well as the definition of the one method it creates, and the main program for the application. The main program is sterotypical for MacApp programs. It intializes the MacApp system, creates and initializes an application object, and starts the application running. The function FailNil is a convenient function which will display an error message if the new function was unable to create a dynamic object because of memory limitations.

The method DoMakeDocument is provided with a number used to identify the document internally. After creating the document (again using FailNil to insure the creation was sucessful), the document is initialized. The arguments to the initialization method IDocument refer to the use of resources, and are not important for our discussion here.

The document created in Figure 13 is an instance of TBilliardDocument, shown in Figure 14. The document will maintain information about the state of the model in three variables, one for a window, one for the view contained in the window, and the actual data for the application maintained as an instance of the class billiardUniverse from Chapter 16. Three methods are overridden. The first is used to initialize the document, while the second and third create the view and window, respectively. The window is an instance of the class TWindow and is created by the function NewSimpleWindow. The arguments to the latter are again unimportant for our discussion here.

The view created by the method DoMakeViews is an instance of the class TBilliardView, shown in Figure 15. The billiard view maintains a pointer back to its associated document. The initialization method not only sets this pointer, but establishes the size of the view. The major tasks for the view are to redraw the image, and to respond to mouse clicks. These are accomplished by the methods Draw and DoMouseCommand, respectively. The latter is once more provided with a number of arguments that are ignored by our simple application, and returns a value indicating whether any special action should be taken to track the mouse (none, in this case).

Creating these three classes is all we need to do to produce a minimal Macintosh application. The application will permit multiple windows containing different instances of the billiard game to appear on the screen at once. Each window can be move or resized. The menu bar will be managed for for the

[2] In actual fact we need to change the Chapter 16 code slightly, but this will be taken care of in the revision.

```
type
   TBilliardApplication = object(TApplication)
      function DoMakeDocument(itsCmdNumber:CmdNumber) : TDocument;
   end;

procedure TBilliardApplication.DoMakeDocument
      (itsCmdNumber:CmdNumber):TDocument;
var
   newDocument : TBilliardDocument;
begin
   new(newDocument);
   FailNil(newDocment);
   { initialize it - arguments refer to resources }
   newDocument.IDocument('bill', 'bill', false, false, false, false);
   DoMakeDocument := newDocument;
end;

begin
   { main program }
   InitUMacApp(5);   { number of handles}
   new(theApplication);   { create the application }
   FailNil(theApplication); { make sure it worked }
   theApplication.IApplication('bill'); { initialize it}
   theApplication.Run;
end.
```

Figure 13: The class TBilliardApplication

```
type
   TBilliardDocument = object(TDocument)
      theView : TBilliardView;
      theWindow : TWindow;
      theModel : billiardUniverse;
      procedure DoInitialState; override;
      procedure DoMakeViews (forPrinting : Boolean); override;
      procedure DoMakeWindows; override;
   end;

procedure TBilliardDocument.DoInitialState;
begin
   new(theModel);
   FailNil(theModel);
   theUniverse := theModel;
   theModel.initialize;
end;

procedure TBilliardDocument.DoMakeViews (forPrinting : Boolean);
begin
   new(theView);
   failNil(theView);
   theView.ITBilliardView(self);
end;

procedure TBilliardDocument.DoMakeWindows;
const
   theWinID = 1001;
begin
   theWindow := NewSimpleWindow(theWinID, not kWantHScrollBar,
      not kWantVscrollBar, nil, theView);
   theWindow.setTitle('billiards game');
   theWindow.Open;
end;
```

Figure 14: The class TBilliardDocument

```
type
   TBilliardView = object(TView)
      myDocument : TBilliardDocument;
      procedure ITBilliardView (theDoc : TBilliardDocument);
      procedure Draw (area : Rect); override;
      function DoMouseCommand(var theMouse : Point;
         var info : EventInfo;
         var hysteresis : Point) : TCommand;
   end;

procedure TBilliardView.ITBIlliardView (theDoc : TBilliardDocument);
var
   viewSize : VPoint;
begin
   myDocument := theDoc;
   SetVPt(viewSize, 450, 350);
   self.IView(nil, nil, gZeroVpt, viewSize, sizeFixed, sizeFixed);
end;

procedure TBilliardView.Draw(area : Rect);
begin
   { set the global variable theUniverse to our data }
   theUniverse := myDocument.theModel;
   { the model knows how to draw itself }
   theUniverse.draw;
end;

function TBilliardView.DoMouseCommand(var theMouse : Point;
   var info: EventInfo; var hysteresis : Point) : TCommand;
begin
   theUniverse := myDocument.theModel;
   { hit the cue ball to start it rolling }
   theUniverse.cueBall.energy := 20.0;
   theUniverse.cueBall.direction := hitAngle(
      theUniverse.cueBall.x - theMouse.h,
      theUnvierse.cueBall.y - theMouse.v);
   { then see what happens next }
   theUniverse.updateMoveableObject;
   DoMouseCommand := gNoChanges;
end;
```

Figure 15: The class TBilliardView

application, allowing access to desk accessories and execution of common commands (such as the command quit used to exit the application). The use of the MacApp classes also makes it easy to modify or extend the application. For example, to make each of the windows scrollable all that is necessary is to remove the not from the constants in the call to newSimpleWindow. To allow games to be saved and restored all that would be necessary is to add a pair of methods to the document class. Printing an image of the window can also be easily added.

Exercises

1. This chapter has described three different types of application frameworks. For each of these describe a type of program for which the framework is better suited than either of the other two.

References

[LaLonde 91] Wilf R. LaLonde and John R. Pugh, *Inside Smalltalk*, Prentice-Hall, 1991.

[Linton 89] Mark A. Linton, John M. Vlissides, and Paul R. Calder, "Composing User Interfaces with InterViews," *Computer*, 22(2): 8-22, 1989.

[Wilson 90] David A. Wilson, Larray S. Rosenstein and Dan Shafer, *Programming with MacApp*, Addison-Wesley, Reading, Mass. 1990.