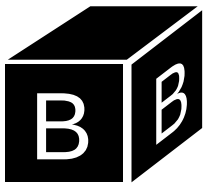


Chapter 9

The MVC Design Pattern

With most of the examples up to this point, the input of a program comes from a dialog box. The output usually goes to a dialog box, but occasionally it goes to the Log. Although input/output via a dialog box is common, it is by no means the only way to get information into or out of a program. Input can come from the focus window, assumed to have been created before the program executes. Also, the program can create a new window in which to display the output.

This chapter shows how a program can get information from the focus window and how it can create a new window on which to write its output. BlackBox uses an effective technique for window I/O called the MVC design pattern. The MVC design pattern is in turn based on a modern software design methodology called object-oriented programming (OOP). OOP and the MVC design pattern are also the underlying foundations of dialog boxes. For simple programs like those we have encountered thus far, BlackBox has hidden the details of OOP and the MVC design pattern. To program I/O with windows, however, requires a bit more knowledge of both OOP and the MVC design pattern.

Objects

Recall that the difference between an abstract data structure (ADS) and an abstract data type (ADT) is that a server module that supplies a client with an ADS supplies only one data structure, while a server module that supplies an ADT exports a type. The client can then declare more than one variable to have that type. An object is a variable that has a type similar to an ADT. In object-oriented terminology, the type is called a *class*, and the variable with that type is called an *object*.

Classes and objects

The Pbox project has an implementation of a stack designed to show the difference between an ADT and a class. In this chapter, there appears to be no advantage of the class over the ADT. The two primary advantages of using a class instead of an ADT are the object-oriented features of *inheritance* and *class composition*. Later chapters show how to program with each of these advanced techniques. Both techniques are pervasive throughout the BlackBox framework. For now, rather than incorporating these techniques into your programs, you will use some objects provided by BlackBox to program window I/O. A few details of the interfaces examined below will not be clear to you until you learn how to program with class composition and inheritance. You can easily learn the recipe of how to use the objects to program window I/O without understanding all the concepts behind OOP.

Inheritance and class composition

But, it will be better if you try to understand as many of the OOP concepts as you can, because they are the basis of most modern software development efforts.

The remainder of this section illustrates a few object-oriented principles by comparing a stack class with the stack ADT from Chapter 6. Figure 9.1 is a listing of the interface of PboxStackObj. The interface of PboxStackADT from Chapter 6 is shown with it for comparison.

```
DEFINITION PboxStackObj;

   CONST
      capacity = 8;

   TYPE
      Stack = RECORD
         (VAR s: Stack) Clear, NEW;
         (IN s: Stack) NumItems (): INTEGER, NEW;
         (VAR s: Stack) Pop (OUT val: REAL), NEW;
         (VAR s: Stack) Push (val: REAL), NEW
      END;

END PboxStackObj.


DEFINITION PboxStackADT;

   CONST
      capacity = 8;

   TYPE
      Stack = RECORD  END;

   PROCEDURE Clear (VAR s: Stack);
   PROCEDURE NumItems (IN s: Stack): INTEGER;
   PROCEDURE Pop (VAR s: Stack; OUT val: REAL);
   PROCEDURE Push (VAR s: Stack; val: REAL);

END PboxStackADT.
```

Figure 9.1
The interface for PboxStackObj and PboxStackADT for comparison.

In both interfaces, Stack is a record type, which is exported. In the stack ADT, the formal parameter list of every procedure must include a variable s of type Stack, because, for example, when a client calls the Pop procedure it must specify not only the item to get the popped value but also the stack from which to pop it. After all, the client can declare more than one stack, so it must have a way to specify the one on which to operate. In the stack class, the specification of the data structure s is not included with the other formal parameters *following* the name of the procedure, but stands alone enclosed in parentheses *before* the name of the procedure. In Component Pascal, the formal parameter before the procedure name is called the *receiver.*

Receivers

Another difference between the ADT and the class is the physical location of the procedures. In the ADT, the procedures are located outside the Stack record. In the

class, the procedures are located within the record. As with the stack ADT, there are additional items in the record of the stack class that are not visible because they are not exported. In object-oriented terminology, procedures that have receivers, and are therefore contained within a record type, are called *methods*.

Methods

Another difference between the procedures of PboxStackADT and the methods of PboxStackObj is the presence of the method attribute NEW. Because inheritance is possible with methods, Component Pascal requires the NEW method attribute to be specified on all newly declared methods. The example in this section does not use inheritance. You will not see the utility of this requirement until you study examples in later chapters of the book that illustrate inheritance.

Example 9.1 In PboxStackObj, Stack is a class, and Pop is a method. In the declaration of Pop

```
(VAR s: Stack) Pop (OUT val: REAL), NEW;
```

the receiver is (VAR s: Stack). ∎

Figure 9.2 shows a dialog box that is implemented with PboxStackObj. It is indistinguishable from the dialog box of Figure 7.9. Behind the scenes, however, this dialog box is implemented with PboxStackObj instead of PboxStackADT as is the dialog box in Figure 7.9.

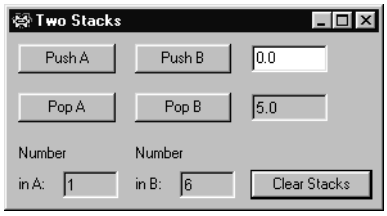


Figure 9.2
The dialog box for manipulating two stacks. It is implemented with PboxStackObj.

The listing in Figure 9.3 uses PboxStackObj to implement the dialog box of Figure 9.2. As in Chapter 3, this chapter names the modules as if the assigned two-digit number of a student in a course is 99, and her homework folder is named Hw99. The structure of the module is identical to the program in Figure 7.10 that uses PboxStackADT to implement the same dialog box. In both programs, there are two global variables, stackA and stackB. They are declared to have type Stack, which is exported from the server module.

The only significant difference between these two programs is how a method is called compared to how a procedure is called. With the ADT, to push a value onto stackB you write

```
PboxStackADT.Push(d.valuePushed, stackB)
```

With the class, to perform the same operation you write

```
stackB.Push(d.valuePushed)
```

```
MODULE Hw99Pr0980;
   IMPORT Dialog, PboxStackObj;

   VAR
      d*: RECORD
         valuePushed*, valuePopped-: REAL;
         numItemsA-, numItemsB-: INTEGER;
      END;
      stackA, stackB: PboxStackObj.Stack;

   PROCEDURE PushA*;
   BEGIN
      stackA.Push(d.valuePushed);
      d.numItemsA := stackA.NumItems();
      Dialog.Update(d)
   END PushA;

   PROCEDURE PushB*;
   BEGIN
      stackB.Push(d.valuePushed);
      d.numItemsB := stackB.NumItems();
      Dialog.Update(d)
   END PushB;

   PROCEDURE PopA*;
   BEGIN
      stackA.Pop(d.valuePopped);
      d.numItemsA := stackA.NumItems();
      Dialog.Update(d)
   END PopA;

   PROCEDURE PopB*;
   BEGIN
      stackB.Pop(d.valuePopped);
      d.numItemsB := stackB.NumItems();
      Dialog.Update(d)
   END PopB;

   PROCEDURE ClearStacks*;
   BEGIN
      stackA.Clear;
      stackB.Clear;
      d.valuePushed := 0.0; d.valuePopped := 0.0;
      d.numItemsA := 0; d.numItemsB := 0;
      Dialog.Update(d)
   END ClearStacks;

BEGIN
   ClearStacks
END Hw99Pr0980.
```

Figure 9.3
A program that uses a stack class to implement the dialog box of Figure 9.2.

The difference in syntax between these two calls illustrates a significant difference in viewpoint between a procedure and a method. With the stack ADT, the procedure belongs to the module. You must, therefore, prefix the procedure name with the name of the *module*, separated by a period. The procedure call is PboxStackADT.Push. With the stack class, the method belongs to the data structure. You must, therefore, prefix the procedure name with the name of the *object*, separated by a period. The method call is stackB.Push.

Procedures belong to modules and methods belong to objects.

A curious feature of object-oriented syntax is that the actual parameter corresponding to the receiver is not enclosed in parentheses. Because the receiver comes before the method name, it seems natural that the actual parameter would come before the method name in the call. However, a receiver contains a pair of parentheses that are not included in the call. In this call to Pop, stackB is an actual parameter along with d.valuePushed. It is not enclosed with parentheses and is separated from Pop by a period.

The table in Figure 9.4 shows the difference in terminology between the items associated with an ADT and those with a class. Unfortunately, terminology in the object-oriented community is not consistent from language to language. The latest Component Pascal language report uses the word "method" as we have here. However, it does not use the words "class" or "object" in the same way as in this text.

Procedure-oriented	Object-oriented
type	class
procedure	method
variable	object

Figure 9.4
Object-oriented terminology.

Models, views, and controllers

The MVC design pattern was developed at the Xerox Palo Alto Research Center in conjunction with an object oriented language called Smalltalk, and was adopted by the designers of BlackBox. The meaning of the letters in the MVC acronym is:

- M model
- V view
- C controller

Models

A *model* is a data structure that stores data. An example of a model is the text model in BlackBox. Text consists of more than just an array of characters. It also includes the font, the size of the font usually measured in points, and various attributes such as whether a letter is bold, italic, or underlined. All this information must be stored for each character in the model.

Another example of a model is a dialog box in the BlackBox forms subsystem. The user constructs a dialog box containing command buttons, radio buttons, text fields for input and output, and captions. The forms model for a dialog box would store data for the width and height of each control and the *x*- and *y*-coordinates for the position of the control in the dialog box. It also stores the font information for any text that appears in the control.

Views

A *view* is the visual representation of a model in a window. In the example of a text model, the view is the image of the characters. Depending on the attributes stored in the text model, the image of a character on the screen would be displayed in one font or another, large or small, bold or not bold, italic or not italic, underlined or not underlined, and so on. In the example of a form, the view is the rendering of the various control objects in the dialog box. For the text field for input or output, depending on the values of the data stored in the model, it will be rendered as tall or

short, wide or narrow, near the top of the dialog box or near the bottom, and so on.

A view cannot exist without a model.

It is clear from the meaning of models and views that a view usually does not exist without a model. Because a view is the visual rendering of a model on an output device, either screen or hardcopy, it must have something to render. From a programming perspective, you must create a model first before you can display it.

Controllers

A *controller* is an object that controls the interaction between the user and the view to manipulate the model. In the example of the text model, the user might want to insert a word at a particular point in a sentence. She would position the cursor at the insertion point with the mouse, click the mouse, then type the word to insert. The controller changes the shape of the cursor when it is positioned over a text view, and locates the position in the text model when the mouse button is clicked.

In the example of the forms model, the user might want to lengthen a text field. She would position the cursor over the field and click the mouse to select the field, which would then be displayed with handles for resizing. Dragging the cursor to the rightmost handle, clicking on the handle and dragging the mouse to the right would lengthen the field. The controller senses where the mouse is when it is clicked over the rightmost handle. As the user drags the handle, the controller sends a message to the model informing it to change the dimensions of the field accordingly.

The primary design concept in the MVC design pattern

The primary design concept in the MVC design pattern is that the view is separate from the model. One advantage of separating a view from its model is that you can have more than one view for a given model. Figure 9.5 shows one text model with two views, one of which displays the first part of the model and one of which displays the last part.

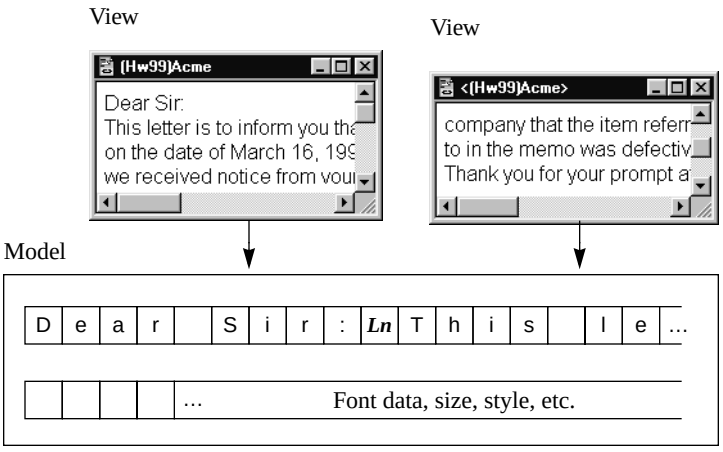


Figure 9.5
The relationship between a view and its model.

The first row of boxes in the model represent the character values. One box contains ***Ln***, which represents the line character. There are no lines in computer memory. Instead, the line character is stored in the model when the user presses the <return> key just as any other character is stored when the user presses its key. The second row of boxes represents the fact that each character in the text model must have associated with it a set of attributes. Because the precise way in which this is accomplished need not concern us now, the boxes appear empty.

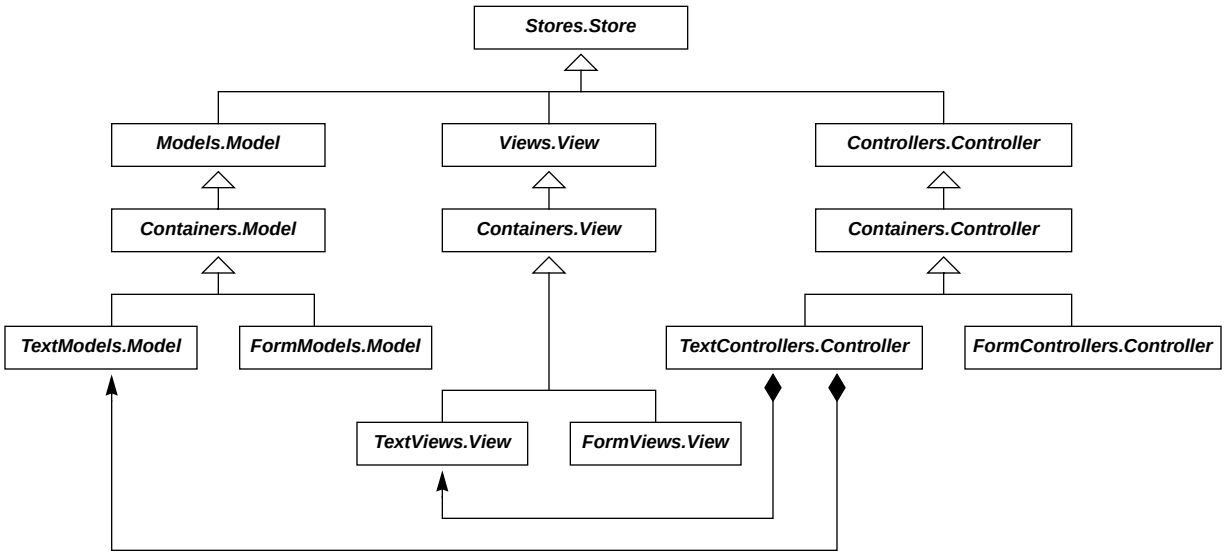
The BlackBox framework lets you open two views of one text model. You can do this by selecting Edit→View In Window (MacOS) or Window→New Window

(MSWindows). This feature is handy when you have written a long module and you are working on a part near the end and you want to see a part near the beginning. With two views, you do not need to continually scroll back and forth from the beginning of your document to the end.

UML class diagrams

Models, views, and controllers are all objects. That is, they are variables that contain data structures and methods (procedures) that operate on the data. The programs in the remainder of this chapter call the methods of several objects within the MVC system. Because the MVC design pattern is an object-oriented system, the objects are related by the two relationships of inheritance and class composition. Both relationships are conveniently represented by a kind of blueprint called a Unified Modeling Language (UML) class diagram. Figure 9.6 is a UML class diagram for most of the classes that are accessed to perform window I/O.

Figure 9.6
A UML diagram for some classes in the MVC system for text and forms.



The figure includes classes for the forms subsystem even though you do not need to access them directly to create the dialog boxes in this book. They are included to show that the MVC design pattern applies to containers other than text. The word before the period in each box is the name of a module, and the word after the period is the name of a class (type). For example, in the box labeled Containers.Model, Containers is a module and Model is a class exported by the module. To perform window I/O you will need to access only four of the modules from Figure 9.6—Views, TextModels, TextViews, and TextControllers—and one module not shown in the figure—PboxMappers.

The triangle is the symbol for inheritance.

In a UML diagram, the triangle △ is the symbol for inheritance. Figure 9.6 shows that classes Models.Model, Views.View, and Controllers.Controller all inherit

from Stores.Store. Stores.Store is called the superclass, and each of the other three is called a subclass. When you inherit a characteristic from a parent, the copy of your parent's genes give you similar characteristics and abilities as your parent. In the same way, when a subclass inherits from its superclass, the superclass gives it a characteristic or ability. The important ability that Stores.Store provides to its subclasses is the ability to be saved on disk, that is, to be stored.

For example, after you design a dialog box and you want to save it in your Rsrc folder, you select File→Save As to store it on disk. The fact that a dialog box is part of the forms subsystem, with its model, view, and controller all being subclasses of Stores.Store, gives it the ability to be stored on disk.

The arrow with diamond tail is the symbol for class composition.

In a UML diagram, the arrow with a diamond tail ◆⟶ is the symbol for class composition. Figure 9.6 shows that class TextControllers.Controller is composed of a link to TextModels.Model and another link to TextViews.View. You can imagine why a controller needs to be composed of links to a model and a view. Recall that a controller is an object that controls the interaction between the user and the view to manipulate the model. Because the controller interacts with both a view and its model, it has links to both.

The iterator design pattern

Besides models, views, and controllers, one additional kind of object is required for window I/O—iterators. Like models, views, and controllers, iterators are objects. That is, they are variables that contain data and methods that operate on the data. Most data structures are storage containers for a collection of values. For example, the list data structure in Chapter 6 stores a collection of strings. An iterator for a data structure is an object that traverses, or iterates over, the collection of values. (The list ADT of Chapter 6 does not have an iterator.)

Iterators

The iterators in BlackBox that are necessary for window I/O are designed to iterate over the character values stored in a text model. There are two kinds of iterators for text—one for inserting text into the model, called a *formatter*, and one for extracting values from the model, called a *scanner*. Formatters are used for window output and scanners are used for window input. Each of these iterators has a relationship with a text model and not with any of its views.

Formatters and scanners

The BlackBox framework supplies powerful iterators for its text model. Unfortunately, the iterators are rather complicated and are difficult for beginning programmers to use. Consequently, the Pbox project supplies its own version of text iterators in the module PboxMappers. Mappers is BlackBox terminology for what are usually known as iterators. Figure 9.7 is the interface for PboxMappers.

In PboxMappers, Formatter and Scanner are both classes (types). The methods that are included in their records have receivers. The first method in each is a procedure called ConnectTo that requires as its parameter a text model. Before you can use a formatter or a scanner you must connect it to a text model. Once it is connected, any operation you perform with that iterator will affect the text model to which it is connected.

Several of the methods for the formatter are similar to the procedures in module StdLog that you use to send output to the Log. WriteInt is similar to StdLog.Int, WriteReal is similar to StdLog.Real, WriteChar is similar to StdLog.Char, WriteString

is similar to StdLog.String, and WriteLn is similar to StdLog.Ln. In each case the method of the formatter does the same thing as the corresponding procedure in StdLog, but it sends the output to a text model instead of to the Log. The only other difference is that WriteReal allows you to specify the number of digits to insert past the decimal point, and StdLog.Real does not.

```
DEFINITION PboxMappers;

   IMPORT TextModels;

   TYPE
      Formatter = EXTENSIBLE RECORD
         (VAR f: Formatter) ConnectTo (text: TextModels.Model), NEW;
         (VAR f: Formatter) WriteInt (n, minWidth: INTEGER), NEW;
         (VAR f: Formatter) WriteReal (x: REAL; minWidth, dec: INTEGER), NEW;
         (VAR f: Formatter) WriteChar (ch: CHAR), NEW;
         (VAR f: Formatter) WriteString (str: ARRAY OF CHAR), NEW
         (VAR f: Formatter) WriteLn, NEW;
         (VAR f: Formatter) WriteIntVector (IN v: ARRAY OF INTEGER; numItm, minWidth: INTEGER), NEW;
         (VAR f: Formatter) WriteRealVector (IN v: ARRAY OF REAL; numItm, minWidth, dec: INTEGER), NEW;
         (VAR f: Formatter) WriteIntMatrix (IN mat: ARRAY OF ARRAY OF INTEGER;
            numR, numC, minWidth: INTEGER), NEW;
         (VAR f: Formatter) WriteRealMatrix (IN mat: ARRAY OF ARRAY OF REAL;
            numR, numC, minWidth, dec: INTEGER), NEW;
      END;

      Scanner = EXTENSIBLE RECORD
         eot-: BOOLEAN;
         (VAR s: Scanner) ConnectTo (text: TextModels.Model), NEW;
         (VAR s: Scanner) Pos (): INTEGER, NEW;
         (VAR s: Scanner) ScanInt (OUT n: INTEGER), NEW;
         (VAR s: Scanner) ScanReal (OUT x: REAL), NEW;
         (VAR s: Scanner) ScanChar (OUT ch: CHAR), NEW;
         (VAR s: Scanner) ScanPrevChar (OUT ch: CHAR), NEW;
         (VAR s: Scanner) ScanString (OUT str: ARRAY OF CHAR), NEW
         (VAR s: Scanner) ScanIntVector (OUT v: ARRAY OF INTEGER; OUT numItm: INTEGER), NEW;
         (VAR s: Scanner) ScanRealVector (OUT v: ARRAY OF REAL; OUT numItm: INTEGER), NEW;
         (VAR s: Scanner) ScanIntMatrix (OUT mat: ARRAY OF ARRAY OF INTEGER;
            OUT numR, numC: INTEGER), NEW;
         (VAR s: Scanner) ScanRealMatrix (OUT mat: ARRAY OF ARRAY OF REAL;
            OUT numR, numC: INTEGER), NEW;
      END;

END PboxMappers.
```

Figure 9.7
The interface for PboxMappers.

The factory design pattern

An interesting characteristic of some objects, particularly those in the MVC design pattern, is how they come into existence. You can think of them as products that are

manufactured in a factory. The analogy to a factory that creates products is so close, that the object-oriented design pattern that does the same thing for objects is called the factory pattern. The *factory design pattern* is a software design technique in which one object, the factory, creates another object. Although the terminology of a factory to describe objects that create other objects is widespread in the OOP community, BlackBox does not use that terminology. Instead, the factories of BlackBox are called *directories*. When you encounter a directory in a BlackBox interface, you should think of it as a factory that produces other objects. When you want to send output to a new window, your program will need to create a new text model and a new text view to display that model in a window. You will use a factory, that is, a BlackBox directory, to create the new model and the new view.

The factory design pattern

The listing in Figure 9.8 shows the interface of module TextModels. The complete interface is quite large. Only those parts are shown that we will need for the programs in this text. If you are interested, you can view the complete interface on-line.

```
DEFINITION TextModels;
   TYPE
      Directory = POINTER TO ABSTRACT RECORD
         (d: Directory) New (): Model, NEW, ABSTRACT;
      END;
      Model = POINTER TO ABSTRACT RECORD (Containers.Model);
   VAR
      dir-: Directory;
END TextModels.
```

Figure 9.8
The interface for TextModels. Many items from the interface are omitted from this listing.

Module TextModels declares two classes (types) that we will use for window output. One is Model, which is declared as

```
Model = POINTER TO ABSTRACT RECORD (Containers.Model);
```

You can see in the UML diagram of Figure 9.6 that TextModels.Model is a subclass of Containers.Model. In Component Pascal, you declare one class to be a subclass of another by enclosing the superclass in parentheses after the word RECORD. That is why Containers.Model is enclosed in parentheses after RECORD. The word ABSTRACT is a record attribute that indicates the nature of the class. It need not concern us at the moment. (You may have noticed that the font for TextModels.Model in Figure 9.6 is bold and slanted. That font style is the UML standard for a class that is abstract.) Also note that a Model is not just a record, but a pointer to a record. The difference between a record and a pointer to a record can be ignored for the time being. Chapter 21 describes pointer types in great detail.

The other class declared in TextModels is Directory, which is defined as

```
Directory = POINTER TO ABSTRACT RECORD
   (d: Directory) New (): Model, NEW, ABSTRACT;
END;
```

It is a class, because it has a method New that is bound to it with a receiver (d: Direc-

tory). New is a function procedure as opposed to a proper procedure, because the last part of its signature is : Model. It is a function that returns a text model. New is the method of the factory that you call to get a newly created text model. So, it makes sense that it would return Model.

TextModels.dir is an abstract data structure (ADS). It is an object in module TextModels and is exported read only. There is only one dir, and any program you write is not allowed to modify it. Its type is Directory. Note that the receiver of method New requires a variable of type Directory. When you call New, you will use dir as the actual parameter for the formal parameter d.

The listing in Figure 9.9 is the interface of module TextViews. As in Figure 9.8, only a small part of the interface is shown here. The interface for TextViews is similar to the interface for TextModels. View is a class that has Containers.View as its superclass. dir is an object ADS that cannot be modified by any module that imports TextViews. As with TextModels, dir will serve as the actual parameter for the formal parameter d.

```
DEFINITION TextViews;
   TYPE
      Directory = POINTER TO ABSTRACT RECORD
         (d: Directory) New (text: TextModels.Model): View, NEW, ABSTRACT;
      END;
      View = POINTER TO ABSTRACT RECORD (Containers.View)
   VAR
      dir-: Directory;
END TextViews.
```

Figure 9.9
The interface for TextViews. Many items from the interface are omitted from this listing.

The signature of method New in TextViews, however, is not quite the mirror image of the signature of method New in TextModels. In module TextModels, New returns a text model and has no formal parameters other than its receiver. In module TextViews, New returns a text view, but it requires a text model for its parameter in addition to its receiver. If you have followed the discussion of the MVC design pattern up until now, this difference should appear reasonable. Remember that a text view cannot exist without a model. Before you can create a text view with NEW, you must have previously created a text model. The model must be supplied to the view factory so the factory can manufacture a new view for that model.

Output to a new window

The programs in this section will show how to use a text model, a text view, and a formatter to create a new window containing output. Besides the factory for the model and the factory for the view, the program will require the services of one more procedure that is exported from module Views. Figure 9.10 shows a small part of its interface.

```
DEFINITION Views;
   TYPE
      View = POINTER TO ABSTRACT RECORD (Stores.Store)
      END;
   PROCEDURE OpenView (view: View);
END Views.
```

Figure 9.10
The interface for Views. Many items from the interface are omitted from this listing.

The interface shows that type View is a subclass of Stores.Store, because Stores.Store is contained in parentheses after the word RECORD. This is consistent with the UML diagram in Figure 9.6, because the triangle symbol is between Stores.Store and Views.View.

Procedure OpenView is not a method. It does not have a receiver, nor is it contained within the View record. It does take a view as a parameter. What kind of view? Any kind that has Views.View as a superclass. Figure 9.6 shows that TextViews.View is a subclass of Containers.View, which is a subclass of Views.View. So, it is legal to supply an object of class TextViews.View as the actual parameter for formal parameter view in procedure OpenView. Similarly, you could supply an object of class FormViews.View for the actual parameter. When you call OpenView, it makes a new window appear on the screen and renders the view inside the window.

Figure 9.11 shows a window that is created by the program in Figure 9.12. Procedure PrintAddress creates a new text model and inserts text into the model. It then creates a new view for the model and displays the view in the window.

Figure 9.11
The output for the program in Figure 9.12

```
MODULE Hw99Pr0981;
   IMPORT TextModels, TextViews, Views, PboxMappers;

   PROCEDURE PrintAddress*;
      VAR
         md: TextModels.Model;
         vw: TextViews.View;
         fm: PboxMappers.Formatter;
   BEGIN
      md := TextModels.dir.New();
      fm.ConnectTo(md);
      fm.WriteString("Mr. K. Kong"); fm.WriteLn;
      fm.WriteString("Empire State Building"); fm.WriteLn;
      fm.WriteString("350 Fifth Avenue"); fm.WriteLn;
      fm.WriteString("New York, NY 10118-0110"); fm.WriteLn;
      vw := TextViews.dir.New(md);
      Views.OpenView(vw)
   END PrintAddress;

END Hw99Pr0981.
```

Figure 9.12
A program that creates a text model and displays it in a text view.

Procedure PrintAddress has three variables—a model md, a view vw, and a formatter fm. Each of these variables is an object. What follows is a description of the effect of the statements in procedure PrintAddress. Figure 9.13 shows the relation-

ships between the objects as the program executes.

When the module is first loaded and before the first statement executes as shown in part (a), md, vw, and fm are all automatically initialized to a special pointer value called NIL. NIL is a value that represents nothing. The fact that these objects have NIL values means that they are not connected to, or do not refer to, anything.

The statement

```
md := TextModels.dir.New()
```

calls the factory method New to manufacture a new text model. TextModels is a module. TextModels.dir is an object (variable) of type Directory in the module. The receiver for New requires a variable of type Directory. So, TextModels.dir is the actual parameter that corresponds to the formal parameter d. New is a function that returns a newly created model. md gets the new model. Figure 9.13(b) shows the effect of executing the statement.

The statement

```
fm.ConnectTo(md)
```

establishes the relationship of the iterator object fm to the new model. ConnectTo is one of the methods of a formatter shown in Figure 9.7. fm is the actual parameter that corresponds to formal parameter f in the receiver, and md is the actual parameter that corresponds to the formal parameter text in the parameter list. Figure 9.13(c) shows the formatter connected to the model.

The statement

```
fm.WriteString("Mr. K. Kong")
```

uses formatter fm to insert text into the new model. Because fm was connected to md earlier, any text that is inserted with the WriteString method is inserted into text model md. Method WriteLn inserts the line character into the model. Figure 9.13(d) shows the text inserted into the model.

The statement

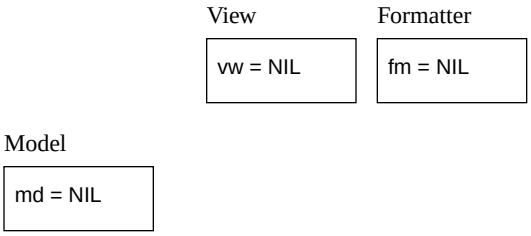
```
vw := TextViews.dir.New(md)
```

calls the factory method New to manufacture a new text view. TextViews.dir is the actual parameter that corresponds to formal parameter d in the receiver of New, and md is the actual parameter that corresponds to formal parameter text in the parameter list. Note how you must supply a previously created text model as a parameter to New. You cannot create a view without a previously created model. Figure 9.13(e) shows the newly created view.

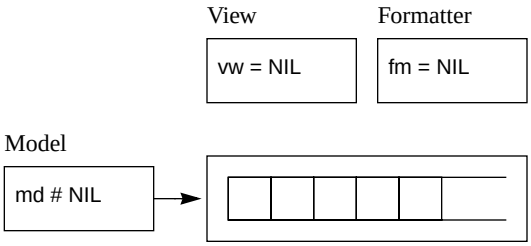
All the processing thus far has gone on behind the scenes. Nothing appears on the screen until the last statement

```
Views.OpenView(vw)
```

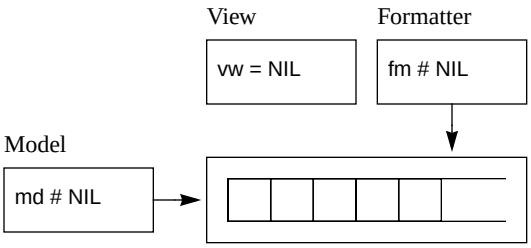
executes. OpenView is the procedure that creates a window for view vw, as Figure



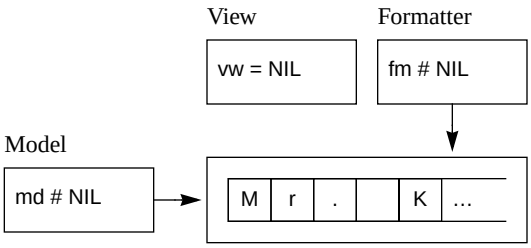
(a) BEGIN



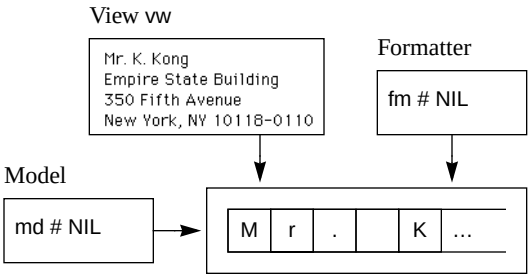
(b) md := TextModels.dir.New()



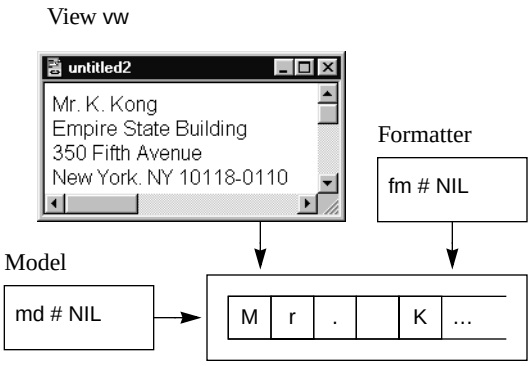
(c) fm.ConnectTo(md)



(d) fm.WriteString("Mr. K. Kong")



(e) vw := TextViews.dir.New(md)



(f) Views.OpenView(vw)

Figure 9.13
The effect of the MVC statements in Figure 9.12

9.13(f) shows.

The program in Figure 9.12 uses a text model, a formatter, which is an iterator, and a text view. It does not require the use of a text controller. The program is quite short. You could simply memorize the first two statements and the last two statements in procedure PrintAddress as the pattern to follow when you want your program to send output to a new window. It would be a mistake, however, to ignore the ideas of the MVC design pattern, the factory pattern, iterators, and the object-oriented concepts that these four statements embody. All these concepts are sound software design principles that are beginning to play a large role in modern software development. It is a strength of the BlackBox framework that these powerful ideas can all be provided in a system with a graphical user interface that is so easy for even beginning programmers to use.

The program in Listing 9.14 inserts the values from two real variables into a text model. The output of the program is shown in Figure 9.15. As in the previous program the procedure has a model object, a view object, and an iterator object for processing with text.

```
MODULE Hw99Pr0982;
   IMPORT TextModels, TextViews, Views, PboxMappers;

   PROCEDURE Rectangle*;
      VAR
         md: TextModels.Model;
         vw: TextViews.View;
         fm: PboxMappers.Formatter;
         width: REAL;
         length: REAL;
   BEGIN
      md := TextModels.dir.New();
      fm.ConnectTo(md);
      width := 3.6;
      length := 12.4;
      fm.WriteString("The width is "); fm.WriteReal(width, 1, 2); fm.WriteLn;
      fm.WriteString("The length is "); fm.WriteReal(length, 1, 2); fm.WriteLn;
      vw := TextViews.dir.New(md);
      Views.OpenView(vw)
   END Rectangle;

END Hw99Pr0982.
```

Figure 9.14
A program that inserts real values into a text model.

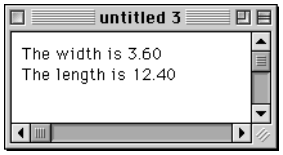


Figure 9.15
The output for the program in Listing 9.14

Besides the objects for the MVC design pattern, procedure Rectangle has local variables width and length. It gives each a real value, then uses method WriteReal to insert their values into the text model. The parameters for PboxMappers.WriteReal are similar to those for PboxStrings.RealToString in Figure 4.12. Namely, in the statement

```
fm.WriteReal(width, 1, 2)
```

width is the variable whose value is inserted into the text model, 1 is the field width which will expand to accommodate the complete value if necessary, and 2 is the number of places displayed past the decimal point.

Input from the focus window

All the programs thus far that required input have taken it from a dialog box. One characteristic of these programs is that little input was required. For example, the problem of computing the wage with overtime required only two numbers to be input—the hours worked and the hourly rate. Many problems require processing larger amounts of data, so much data that it would be impractical to ask the user to enter it every time into a dialog box. These problems are solved by storing the information in a document. Typically, the information is created by selecting the File→New menu option and entering it as text. If there is too much information to enter in a single session the data entry person can save the document in a file and continue entering at a later time. When the information needs to be processed, the document is opened so that the information is visible in the focus window. The procedure that processes the information is then invoked by selecting a menu option.

The next program illustrates the above scenario, although the example is shown with a small amount of data to keep things simple. Figure 9.16 shows the result of running the program. Figure 9.16(a) shows the data to be processed in a window titled Data. It was saved previously in file with that name and consists of text that represents two real values. While this window was focused, the user selected the menu option as shown in Figure 9.16(b), which resulted in the output to the Log shown in Figure 9.16(c). The processing simply consists of interpreting the first value as the number of hours worked and the second value as the hourly rate then computing the resulting wage with a possibility of overtime.

(a) The input window. **(b)** The menu selection **(c)** The output.

Figure 9.16
The input and output of the program in Figure 9.18.

To get the values from the focus window requires the use of module TextControllers, whose partial interface is shown in Figure 9.17.

TextControllers.Controller is a class. The UML diagram in Figure 9.6 shows that TextControllers.Controller is a subclass of Containers.Controller because of the triangle between them. Figure 9.17 shows that TextControllers.Controller is a subclass of Containers.Controller because Containers.Controller is contained in parentheses after the Controller record.

```
DEFINITION TextControllers;
   TYPE
      Controller = POINTER TO ABSTRACT RECORD (Containers.Controller)
         view-: TextViews.View;
         text-: TextModels.Model;
      END;
   PROCEDURE Focus (): Controller;
END TextControllers.
```

Figure 9.17
The interface for TextControllers. Many items from the interface are omitted from this listing.

The UML diagram also shows an arrow originating at a diamond symbol on TextControllers.Controller and pointing to TextViews.View. That arrow represents class composition. It indicates that TextControllers.Controller is composed of, or contains a reference to, TextViews.View. Figure 9.17 shows that TextControllers.Controller has a field named view in its record with type TextViews.View. That is the meaning of class composition. One class is *composed of* a second class if the first class has a field in its record whose class (type) is that of the second class. Furthermore, a UML diagram illustrates class composition by an arrow originating at a diamond from the box of the first class and terminating at the box of the second class. Similarly, Figure 9.17 shows that TextControllers.Controller is composed of TextModels.Model, because it contains a field of that type as well.

Class composition

Both fields TextViews.Views and TextModels.Model are exported read-only. That means that you cannot change their values by assigning something to them. However, if you have a local variable of type TextModels.Model you can assign the server module's exported field to it. Such an assignment does not change the server modules's exported field. The next program uses that technique to extract a model from a controller.

The following program implements the processing illustrated in Figure 9.16. It has six local variables, three of which are variables for the MVC design pattern—a model md, a controller cn, and a scanner (iterator) sc. Figure 9.19 shows the effect of the MVC statements in procedure ComputeWages.

At the beginning of the procedure, the MVC variables, md, cn, and sc, all have values of NIL. Figure 9.19(a) shows that the values are NIL because there are no arrows that link the three variables to anything. Because there is a view in the focus window, there must be a model that already exists before the program executes. Both the model and its view were created previously by the word processor. The arrow from the view to its model indicates that the view contains a reference to its model.

When function procedure TextControllers.Focus executes in the assignment statement

```
cn := TextControllers.Focus()
```

the BlackBox framework detects which window is the focus window. If this window contains the view of a text model, TextControllers.Focus returns a controller for that view and assigns it to cn. Remember that the controller contains a reference (class composition) to both a view and its model. When the controller gets the value from TextControllers.Focus, the field cn.view gets a reference to the focus view, and the

field cn.text gets a reference to the view's text model. Figure 9.19(b) shows these values by the arrows from the controller box.

```
MODULE Hw99Pr0983;
   IMPORT TextModels, TextControllers, PboxMappers, StdLog;

   PROCEDURE ComputeWages*;
      VAR
         md: TextModels.Model;
         cn: TextControllers.Controller;
         sc: PboxMappers.Scanner;
         hours, rate: REAL;
         wages: REAL;
   BEGIN
      cn := TextControllers.Focus();
      IF cn # NIL THEN
         md := cn.text;
         sc.ConnectTo(md);
         sc.ScanReal(hours);
         sc.ScanReal(rate);
         IF hours <= 40.0 THEN
            wages := hours * rate
         ELSE
            wages := 40.0 * rate + (hours - 40.0) * 1.5 * rate
         END;
         StdLog.String("Wages:  "); StdLog.Real(wages); StdLog.Ln
      END
   END ComputeWages;

END Hw99Pr0983.
```

Figure 9.18
A program that gets its input from the focus window.

If the window contains the view of some other kind of model such as a graphic of some kind, TextController.Focus returns NIL. Before proceeding, procedure ComputeWages checks the value of cn to verify that the focus window indeed contains the view of a text model.

Now that cn contains a reference to the model, the statement

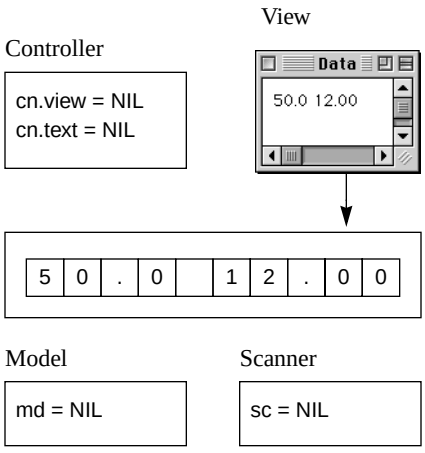
```
md := cn.text
```

extracts the model from cn. After this statement executes, md will be the text model whose view is displayed in the focus window. So, the only purpose of cn is to test if the focus window contains a text view. If so, it gets the view and the model from the focus window. Now that it has given the model to md its job is done. Figure 9.19(c) shows the value given to md by the arrow from the model box.

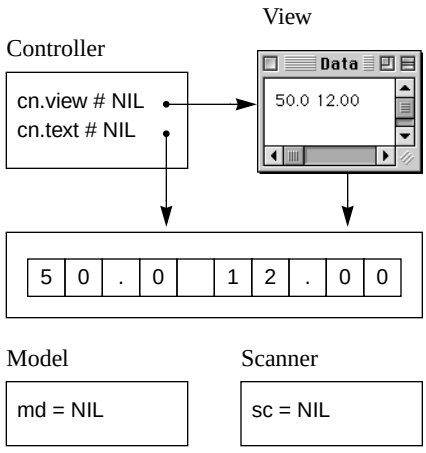
The statement

```
sc.ConnectTo(md)
```

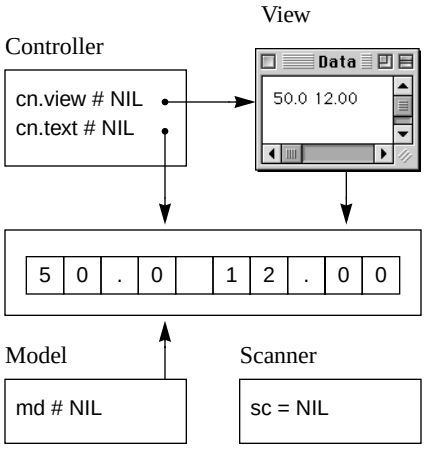
establishes the relationship between the scanner sc and the model md. Now that the



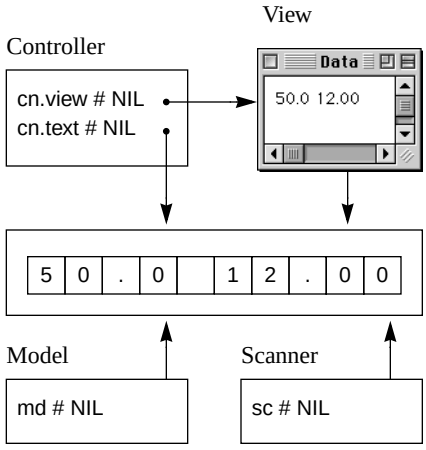
(a) BEGIN



(b) cn := TextControllers.Focus()



(c) md := cn.text



(d) sc.ConnectTo(md)

Figure 9.19
The effect of the MVC statements in Figure 9.18

scanner is connected to the text model, it is ready to scan the values into our variables hours and rate. Figure 9.19(d) shows the connection by the arrow from the scanner box.

When a scanner is connected to a model, it is always positioned at the beginning of the model. Therefore, the statement

```
sc.ScanReal(hours)
```

scans from the beginning of the text model. This procedure assumes that the next characters of text in the model represent a real value. If some other character other

than a digit or a decimal point is encountered by the scanner, a trap will occur. The scanner skips over any leading spaces or tabs until it encounters a string of digits containing a single decimal point. It stops scanning when it reaches the first trailing non digit character such as a space or the end of a line. It gives the real value to variable hours.

The next statement

```
sc.ScanReal(rate)
```

picks up the scan where the previous scan left off. In this scenario as depicted in Figure 9.16(a), the previous scan gives the variable hours the value 50.0 and this scan gives variable rate the value 12.00.

The remaining statements in procedure ComputeWages compute the wage and output it to the Log.

Creating menu selections

Procedure ComputeTotal is activated by a menu selection as shown in Figure 9.16(b). The menu choices that come standard with BlackBox therefore need to be augmented to allow the user to activate the program. A menu is a resource, in the same way that a dialog box is a resource. You create a menu by writing a BlackBox text document and storing it in your project's Rsrc folder. Figure 9.20 shows the document that created the menu selections in Figure 9.16(b).

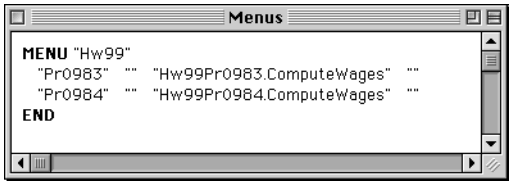


Figure 9.20
The menu document that produced the menu in Figure 9.16(b).

The menu document includes menu selections for the programs in both Figure 9.18 and in Figure 9.22 below. The content of the menu document is

```
MENU "Hw99"
    "Pr0983"    ""    "Hw99Pr0983.ComputeWages"    ""
    "Pr0984"    ""    "Hw99Pr0984.ComputeWages"    ""
END
```

The first line of the menu contains the word MENU followed by the title of the menu enclosed in quotes. In this case, Hw99 is the title that appears on the menu bar at the top of the screen. This title is appropriate if you are using this book in conjunction with a course and your assigned two-digit number is 99. Of course, if you are developing software for another user you would use a title that is more descriptive of the selections that are available for that menu.

Following the menu title is a line for the first selection that contains four strings. The second and fourth strings will not concern us until later. They will always be the empty string for now. The first field is the name of the selection that appears when the user clicks the menu title. The third field is the command that is activated when that selection is made. You can see from Figure 9.16(b) that the user has clicked on the title Hw99 and as a result he may select between Pr0983 and Pr0984. These are the selections that are enumerated in the first strings of the lines in Figure 9.20. When he selects Pr0983, procedure Hw99Pr0983.ComputeWages executes as specified by the third string in the line whose first string is Pr0983.

The last line in the menu document is END. It is possible to have one menu document produce more than one menu on the menu bar. Each menu begins with a line containing MENU and ends with a line containing END.

Your menu document must be saved in your Rsrc folder and must be named Menus. When BlackBox starts up, it scans all the folders named Rsrc contained in all the project folders. If it finds a file named Menus in a Rsrc folder, it interprets the contents as described above and installs the menu in the menu bar at the top of the screen. If you have created or modified a new menu and you wish to install or update it, you do not need to quit BlackBox for the sole purpose of starting it up again to install the menu. After you have saved the menu document in the Rsrc folder simply select Info→ Update All Menus, which will initiate the installation process.

Dialog boxes from programs

Previous chapters showed how to activate a dialog box by providing the user with a commander in the documentation file. Although this is a common technique in the BlackBox environment, commander buttons and documentation files are not common in commercial programs for either MSWindows or MacOS. Sometimes a dialog box is activated by a program to provide information to the user. For example, if you are playing a computer game a dialog box may appear to provide you with information about the progress of the game.

BlackBox provides a way for a program to activate a dialog box. Figure 9.21 shows such a scenario. It is similar to the scenario shown in Figure 9.16, except that the results are displayed in a dialog box instead of on the Log.

(a) The input window. **(b)** The menu selection **(c)** The output.

Figure 9.21
The input and output of the program in Figure 9.22.

The program in Figure 9.22 shows how to activate a dialog box. None of the fields in the dialog box of Figure 9.21(c) allow the user to change the displayed val-

ues. Consequently, the corresponding fields in the interactor d are all exported read-only. As in the previous module, this program takes its input from the focus window. It has the usual MVC parameters to scan the input values—a model md, a controller cn, and a scanner sc. Procedure ComputeWages is activated by the menu selection Hw99→ Pr0984.

```
MODULE Hw99Pr0984;
   IMPORT TextModels, TextControllers, PboxMappers,
      PboxStrings, Dialog, StdCmds;
   VAR
      d*: RECORD
         hours-, rate-: ARRAY 16 OF CHAR;
         wages-: ARRAY 16 OF CHAR
      END;

   PROCEDURE ComputeWages*;
      VAR
         md: TextModels.Model;
         cn: TextControllers.Controller;
         sc: PboxMappers.Scanner;
         hours, rate: REAL;
         wages: REAL;
   BEGIN
      cn := TextControllers.Focus();
      IF cn # NIL THEN
         md := cn.text;
         sc.ConnectTo(md);
         sc.ScanReal(hours);
         sc.ScanReal(rate);
         IF hours <= 40.0 THEN
            wages := hours * rate
         ELSE
            wages := 40.0 * rate + (hours - 40.0) * 1.5 * rate
         END;
         PboxStrings.RealToString(hours, 1, 1, d.hours);
         PboxStrings.RealToString(rate, 1, 2, d.rate);
         PboxStrings.RealToString(wages, 1, 2, d.wages);
         StdCmds.OpenAuxDialog('Hw99/Rsrc/Dlg0984', 'Payroll');
         Dialog.Update(d)
      END
   END ComputeWages;

END Hw99Pr0984.
```

Figure 9.22
A program that gets its input from the focus window and puts its output in a dialog box.

The procedure is identical to the one in the previous module except for its output. There are three local variables—hours, rate, and wages—that all have type REAL. The procedure uses the values of hours and rate that are scanned from the focus window to compute the value for wages. Each of these values is converted to a string with the desired number of places past the decimal point for display in the dialog

box. The same command that is placed after a commander button in a documentation file is executed directly from the program.

```
StdCmds.OpenAuxDialog('Hw99/Rsrc/Dlg0984', 'Payroll')
```

As usual, the first parameter is a string that names the file where the dialog box is stored and the second parameter is a string that gives the title of the dialog box.

When you execute the procedure the first time with given values in the focus window, a new dialog box will appear with the computed values displayed. If you keep the dialog box visible, change the values in the focus window, and select Hw99→ Pr0984 once again, a second dialog box will not appear. Instead, the values in the old dialog box will simply be updated to reflect the new computation.

Exercises

1. Name the two advantages of using a class instead of an ADT.

2. When you program with objects, **(a)** what corresponds to the word type? **(b)** What corresponds to the word procedure? **(c)** What corresponds to the word variable?

3. **(a)** What is a model? **(b)** What is a view? **(c)** What is a controller? **(d)** What is an iterator?

4. What is the primary design concept in the MVC design pattern?

5. In the interface of Figure 9.7, **(a)** is Scanner a module, a type, a constant, a variable, or a procedure? **(b)** Is Scanner a model, a view, a controller, an iterator, or a factory? **(c)** In the line

```
(VAR f: Formatter) WriteReal (x: REAL; minWidth, dec: INTEGER), NEW
```

is f an actual parameter or a formal parameter? **(d)** Is Formatter a model, a view, a controller, an iterator, or a factory?

6. In the interface of Figure 9.8, **(a)** is dir a module, a type, a constant, a variable, or a procedure? **(b)** Is dir a model, a view, a controller, an iterator, or a factory?

7. In the procedure in Figure 9.12 in the line

```
vw := TextViews.dir.New(md)
```

(a) is TextViews a module, a type, a constant, a variable, or a procedure? **(b)** Is dir a module, a type, a constant, a variable, or a procedure? **(c)** Is New a module, a type, a constant, a variable, or a procedure? **(d)** Is md a formal parameter or an actual parameter?

Problems

8. Do Chapter 7, Problem 14, to construct an RPN calculator, but use the stack class from PboxStackObj.

9. Do Chapter 7, Problem 16, to construct a full-featured scientific calculator, but use the stack class from PboxStackObj.

10. Do Chapter 7, Problem 20, to construct a dialog box for two stacks with an "A to B" button but use the stack class from PboxStackObj.

11. Write a Component Pascal program to output the following two-line message on a new window:

```
She said, "Hi there.
What's up?"
```

Use two WriteString procedure calls for the second line to print both the single and the double quote marks. Test your program by inserting a commander button in a documentation file to execute the exported procedure.

12. Write a Component Pascal program to output to a new window your name and address suitable for use as a mailing label. Test your program by inserting a commander button in a documentation file to execute the exported procedure.

13. Using a procedure that is not exported to output a single pattern, write a Component Pascal program to output the following triple pattern on a new window:

```
+
+++
+++++
+++
+
+
+++
+++++
+++
+
+
+++
+++++
+++
+
```

Test your program by inserting a commander button in a documentation file to execute the exported procedure.

14. Write a program that inputs an integer value for the number of feet and a real value for the number of inches from the focus window. When the user selects a choice from a menu item, compute the equivalent length in meters and output the results of the computation to the Log. One inch is exactly 0.0254 meters and one foot is exactly 12 inches. Here is a sample output to the Log.

```
Feet: 4
Inches: 3.8
Meters: 1.31572
```

15. Work Problem 14, but display the number of feet and inches and the computed value for meters in a dialog box.

16. Write a program that inputs two real values for the lengths of two perpendicular sides of a right triangle from the focus window. When the user selects a choice from a menu item, compute the length of the hypotenuse and show all three lengths on the Log with their values identified appropriately as the values are in Problem 14.

17. Work Problem 16, but display the lengths of the two perpendicular sides and the computed value for the length of the hypotenuse in a dialog box.

18. Write a program to input three real numbers from the focus window. When the user selects a choice from a menu item print them in descending order on the Log.

19. Work Problem 18, but output the three real numbers in a dialog box.

20. Write a program to input three integers from the focus window. When the user selects a choice from a menu item output the number that is neither the smallest nor the largest on the Log. Assume that none of the integers are equal.

21. Work Problem 20, but output the number in a dialog box.

22. Write a program to input two integers from the focus window. When the user selects a choice from a menu item output to the Log either the larger integer or a message stating that they are equal.

23. Work Problem 22, but output the number in a dialog box.