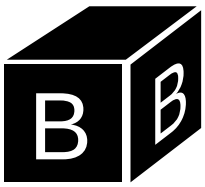


Chapter *23*

Inheritance and Polymorphism

The history of computer science shows a steady progression from lower levels of abstraction to higher levels. When the electronic computer was first invented in the mid twentieth century, there was no assembly language much less the higher level languages with which we are familiar today. It is no accident that the historic evolution is toward progressively higher levels of abstraction instead of the other way around. Human intellectual progress shows that generalities are usually discovered from many specific observations. It is only with hindsight that you can start with the general case and deduce specific consequences from it.

This chapter describes six levels of abstraction.

Six abstraction processes

- Data abstraction, encompassing
  - Type abstraction, and
  - Structure abstraction
- Control abstraction, encompassing
  - Statement abstraction, and
  - Procedure abstraction
- Class abstraction
- Behavior abstraction

Previous chapters show programs that use the first five abstraction processes—type, structure, statement, procedure, and class. This chapter reviews these five abstraction processes and introduces the sixth—behavior abstraction.

Data abstraction

Plato, in his theory of forms, claimed that reality ultimately lies in the abstract form that represents the essence of individual objects we sense in the world. In the *Republic*, written in the form of a dialogue between Socrates and a student, he writes:

Plato's abstraction

> Well then, shall we begin the enquiry in our usual manner: Whenever a number of individuals have a common name, we assume them to have also a corresponding idea or form: do you understand me?
>
> I do.
>
> Let us take any common instance; there are beds and tables in the world—

> plenty of them, are there not?
>
> Yes.
>
> But there are only two ideas or forms of them—one the idea of a bed, the other of a table.
>
> True.
>
> And the maker of either of them makes a bed or he makes a table for our use, in accordance with the idea—that is our way of speaking in this and similar instances—but no artificer makes the ideas themselves: how could he?
>
> Impossible.

Plato's consideration between the specific and the general exemplifies the abstraction process. Another example of the abstraction process is the concept of type in programming languages. Consider all the possible real values, such as 2.0, 5.2, –43.7, 0.8, and so on. In the same way that Plato considered many different instances of a table to be representations of a single abstract table, from a computation point of view the collection of all possible real values defines a single abstract type REAL. Figure 23.1 shows the abstraction process, known as *type abstraction*, for type REAL. A type is defined by a collection of values. Each value, such as 5.2 in the box on the left, is specific, while the type REAL is general.

Type abstraction

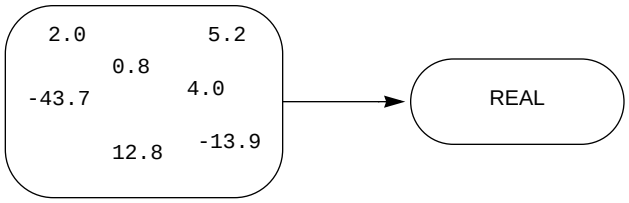


Figure 23.1
Type abstraction for type REAL.

In the history of computing languages, types emerged as one of the first steps toward higher levels of abstraction. At the machine level, which must be programmed with machine language or its equivalent assembly language, there are no types other than the bit patterns of pure binary. With assembly language, you have unlimited freedom to interpret a bit pattern any way you choose. The same bit pattern in a specific memory location can be interpreted as an integer and processed with the addition circuitry of the processor. It can be interpreted as a character and sent to a Web page as such. It can even be interpreted as an instruction and executed.

Every variable has a name, a type, and a value.

In Component Pascal, every variable has a name, a type, and a value. The name is an identifier, defined by the syntax rules of the language. The type is supplied by the language. Both the name and the type of a variable are determined when the software designer writes and compiles the program. The value of a variable, on the other hand, is stored in the main memory of the computer as the program is executing. The value stored is one of the values that defines the type.

The compiler enforces type compatibility, which is a restriction on the freedom of programmers that they do not have with assembly language. The abstraction process frequently imposes a loss of freedom because the nature of abstraction is the

Advantages and disadvantages of abstraction

hiding of detail. Programmers then have no access to the details that are hidden. With the advent of types to restrict the value that a variable can have to some mathematical entity like a real number comes the inability to consider the bit pattern behind the value. But the restriction of freedom to access low-level details is also liberation from the necessity to do so. Abstraction is powerful because the limitation it places on the programmer's ability to access low level details at the same time frees the programmer from that requirement.

The abstraction process permits the grouping together of specific real values into a type because each value shares certain characteristics with all the other values. For example, each value has a sign and a magnitude. Any value can be combined with any other value with the arithmetic operators like multiplication. And any value can be compared with any other value to determine whether the first is less than, equal to, or greater than the second. If it were not for these common properties among individual values, the grouping together of them to define a type would not be useful.

Furthermore, the collection of many specific numeric values to make a general type is useful in a programming language because it models the same process in the real world. For example, the type REAL in Component Pascal corresponds to the notion of a real number in mathematics. All computer applications exist to solve problems in the human world. The first step toward solving any problem is to model it with the machine. There are usually approximations to the model, which may make the solution approximate. For example, there are only a finite number of real values that a computer can store while there are an infinite number of real values in mathematics. Nevertheless, one source of power of the abstraction process in computing is that it can mirror the same process in the human world and so serve as a model to compute the desired solution.

The next step toward higher levels of abstraction in programming languages occurred when languages gave programmers the ability to create new types as combinations of primitive types. Collections of primitive types are known as records or structures in most programming languages. The corresponding abstraction process is called *structure abstraction*.

Structure abstraction

For example, suppose you need to process several different shapes—rectangles and circles. Figure 23.2 shows geometrically how the collection of all possible rectangles and circles define a single shape type. The abstraction process parallels the process of defining a type as a collection of values. An individual rectangle is characterized by its length, say 2.0, and width, say 5.2. An individual circle is characterized by its diameter, say 4.1. A shape is a collection of values—each one having a type—to store its dimensions.

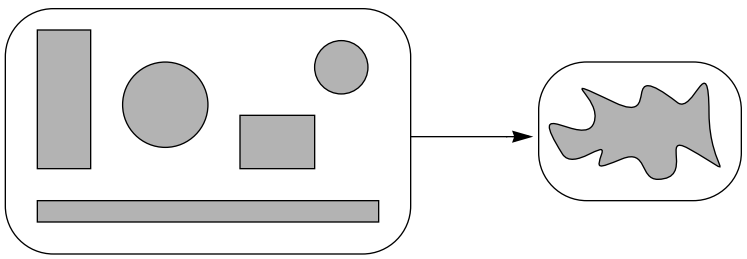


Figure 23.2
Structure abstraction to abstract from specific shapes of many different sizes to a single shape with a general size.

In Component Pascal, you declare a new type as a record structure, which is a collection of fields, each one of which is a primitive type supplied by the language or a previously declared type. To store the information about a shape you need to distinguish between rectangles and circles. You can do that with an integer field called kind. If kind has value 0 then the shape is a rectangle. If kind has value 1 then the shape is a circle. To store the dimensions of a rectangle you need real fields named length and width. To store the dimensions of a circle you need a real field named diameter. The type could be declared

```
MODULE ...ShapeADT;
   TYPE
      Shape* = POINTER TO RECORD
         kind: INTEGER;
         length, width: REAL;
         diameter: REAL
      END
```

You could then declare an individual shape as a variable of type Shape.

```
VAR
   myShape: ...ShapeADT.Shape
```

To initialize myShape to be a 2 × 3 rectangle you use the usual period notation to separate the variable name from the record field name as follows.

```
myShape.kind := 0;
myShape.length := 2.0;
myShape.width := 3.0
```

Programmer-defined types are powerful because they allow the programmer to conveniently model the problem to mirror the situation in the problem domain. For example, an airline reservation system might need to store a collection of information for each ticket it sells, say the passenger's name, address, flight date, flight number, and price of the ticket. Collecting all these types into a single programmer-defined type allows the program to process a ticket variable as a single entity.

Computation abstraction

Statement abstraction

Abstraction of data is only one side of a two-sided coin. The other side is abstraction of computation. At the lowest level between programming languages and the machine is *statement abstraction*.

All computers consist of a central processing unit (CPU) that has a set of instructions wired into it. The instruction set varies from one computer chip maker to another, but all commercial CPUs have similar instructions. CPUs contain cells called registers that store values and perform operations on them. The collection of the operations specifies a computation.

Typical instructions are load, add, mul, and store. The load instruction gets a value from main memory and stores it in a register of the CPU. The add instruction

adds the content of two registers. The mul instruction multiplies the content of two registers. The store instruction puts a value from a register of the CPU into main memory.

Before the advent of high-level languages, programmers wrote their programs using the individual instructions of the instruction set of the particular CPU on which the program was designed to run. Figure 23.3 shows an example of a sequence of instructions for some hypothetical CPU that computes the perimeter of a rectangle. The first two instructions load the value of length into register r1 and the value of width into register r2. The next instruction adds the content of r1 to r2 and puts the sum in register r3. Then, 2.0 is multiplied by the content of r3 with the result placed back in r3, after which it is stored in main memory in the location reserved for variable perim.

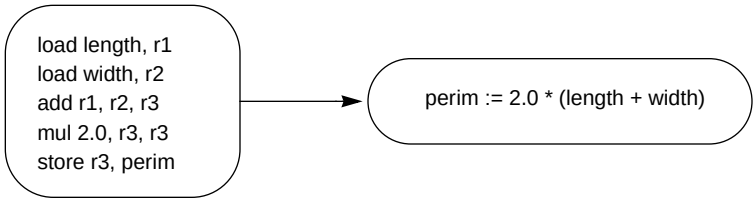


Figure 23.3
Statement abstraction for the assignment statement.

The language illustrated by this sequence of instructions is called assembly language. When you program in assembly language you must consider the details of the CPU—how many registers it has, how to access them, and which values you want to store in which registers. In a high-level language, however, all those details are hidden. The compiler abstracts them away from the view of the programmer, so that the programmer need only write the single assignment statement

```
perim := 2.0 * (length + width)
```

With statement abstraction, even the structure of the CPU is hidden. The programmer does not need to know about registers or hardware instruction sets. A single assignment statement in Component Pascal is a collection of several instructions in assembly language. One statement in a high-level language is defined by many statements at the machine level, in the same way that one type in a high-level language is defined by many possible values at the machine level.

Corresponding to structure abstraction on the data side of the coin is procedure abstraction on the computation side. In the same way that high-level languages allow you to collect variables into structures to create a new data type, they allow you collect statements into procedures to create a new computation. The corresponding abstraction process is *procedure abstraction*.

Procedure abstraction

Figure 23.4 shows procedure abstraction for the computation of the perimeter of a shape. The Component Pascal computation of the perimeter of an arbitrary shape is encapsulated in a function with formal parameter s whose type is Shape. Any time the programmer needs to compute the perimeter, for example to output it with StdLog.Real, a simple call to the function is all that is required. The computation need only be done once, freeing the programmer from having to remember those details whenever the computation is required. For example, if you have two vari-

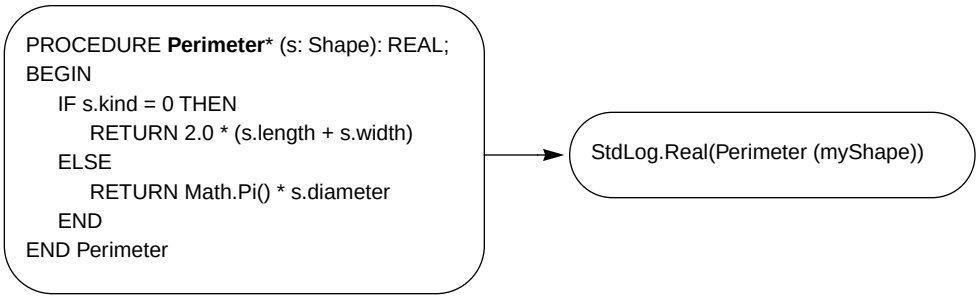


Figure 23.4
Procedure abstraction for the computation of the perimeter of a rectangle

ables—myShape and yourShape—both of type Shape, you can output their perimeters with

```
StdLog.Real(Perimeter (myShape)); StdLog.Ln;
StdLog.Real(Perimeter (yourShape)); StdLog.Ln
```

It would not matter if myShape is a circle and yourShape is a rectangle. The procedure takes care of determining what kind of shape the parameter is and returns the appropriate value. The details of the computation are hidden in the function procedure calls. As with statement abstraction in Figure 23.3, one procedure call at a high level causes the execution of several statements at a low level.

Class abstraction

The next step in the evolution of programming languages toward higher levels of abstraction was the combination of data abstraction with computation abstraction to produce class abstraction. Consider again the shapes in Figure 23.2 and imagine what sort of processing might be required for such geometric figures. A rectangle might represent part of a building like the interior wall of a room or a door. If the walls and doors are to be painted your program would need to compute the area of each rectangle to determine the amount of paint required. Or a circle might represent a corral around which a fence is to be erected. Your program would then need to compute the perimeter to determine the amount of material required for the fence.

Before the advent of object-oriented programming, the function to compute the area or the perimeter of a shape would exist separately from its dimensions. For example, you might have functions to compute the area and perimeter of a shape, which is passed as a parameter in the parameter list of the function. The interface for the server module providing ShapeADT might look something like

```
DEFINITION ...ShapeADT;
   TYPE
      Shape = POINTER TO RECORD END;
   PROCEDURE Area (s: Shape): REAL;
   PROCEDURE Perimeter (s: Shape): REAL;
```

where the kind field and the dimensions are not exported, and so do not appear in the interface.

With class abstraction, however, you bind the procedures with the type resulting in type-bound procedures, also called methods, having the interface

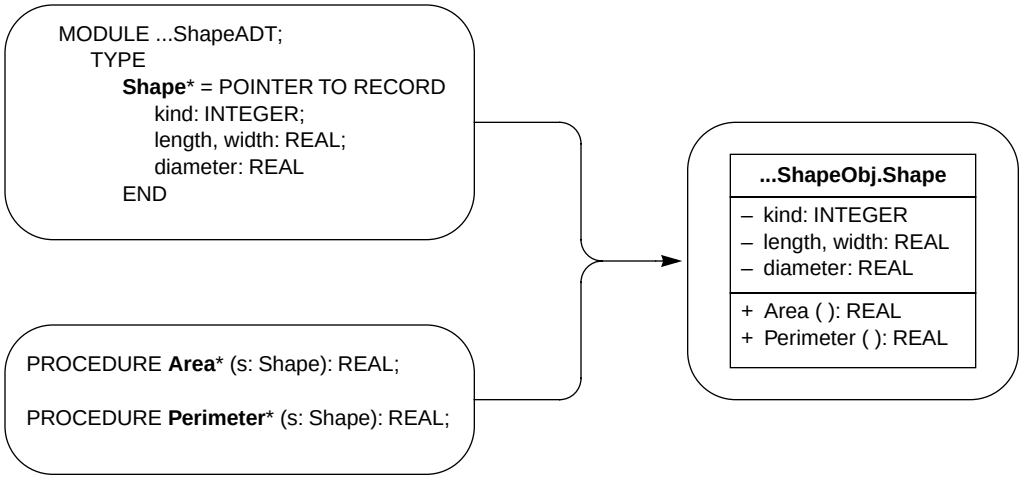
```
DEFINITION ...ShapeObj;
   TYPE
      Shape = POINTER TO RECORD
         (s: Shape) Area (): REAL, NEW;
         (s: Shape) Perimeter (): REAL, NEW
      END;
```

The procedures belong to the type rather than belonging to the module. As before, values for the kind field and the dimensions are not exported and have an implementation like

```
MODULE ...ShapeObj;
   TYPE
      Shape* = POINTER TO RECORD
         kind: INTEGER;
         length, width: REAL;
         diameter: REAL
      END
```

Figure 23.5 shows the process of class abstraction with this shape example. In the figure, the data part on the top left combines with the control part on the bottom left to produce the class on the right. The box in the right part of the figure is the Unified Modeling Language (UML) class diagram for the class named Shape. It shows all the fields of the record, whether exported or not, and also includes the head of each method.

Figure 23.5
Class abstraction that combines the structure abstraction of Figure 23.2 with the procedure abstraction of Figure 23.4.



A class diagram has three parts. The top box contains the name of the class, in this diagram …ShapeObj.Shape. The UML standard is for the class name to be in a

Attributes

bold typeface. The middle box contains the data fields of the record, which are known as *attributes*. In this diagram the attributes are kind, length, width, and diameter. In UML, an item that is not exported is preceded by a – sign and an item that is exported is preceded by a + sign. All of the attributes are preceded by a – sign because none of them are exported. The bottom box contains the methods, which are known as *operations*. There are slight differences in syntax between Component Pascal and UML. The receiver of the methods is not shown before the name of the operation, because it can be inferred by the name of the class in the top box of the UML diagram.

Operations

Example 23.1 The heading for the Perimeter procedure would be written in Component Pascal as

```
PROCEDURE (s: Shape) Perimeter* (): REAL, NEW
```

The corresponding UML entry in the class diagram for the operation is

```
+ Perimeter ( ): REAL
```

The object orientation shift in focus

Object orientation is a viewpoint that shifts the focus from an external operation that requires the input of data about the shape, to an internal operation that is part of the shape itself. This is a significant shift in focus. Computing the perimeter is no longer something that you do to a shape. It is something the shape does for you.

In Figure 23.2, each individual shape on the left has an area and a perimeter in addition to its dimensions. The area and perimeter are not data values that are independent from the dimensions. So, their values should not be stored the same way the dimensions are stored, but they should be computed from the dimensions. In object-oriented design, the functions to compute the area and perimeter are no longer external to the type, but are internal. They literally become part of the type.

To emphasize the shift in focus when a function is bound to a type, object-oriented designers established a new set of terminology. Roughly speaking, in object-oriented terminology

- *class* corresponds to *type*
- *object* corresponds to *variable*
- *method* corresponds to *procedure* or *function*

That is, an object has a class, like a variable has a type. It is more usual to state that an object is an instance of a class rather than to state that an object has a class.

A shape class

Although the true power of object orientation requires behavior abstraction, this section presents a program that illustrates class abstraction without it. The purpose of the program is to show how to solve a problem without behavior abstraction, so it can be contrasted with the program in the following section, which does use it. Figure 23.6 shows the abstraction process for class abstraction without behavior abstraction.

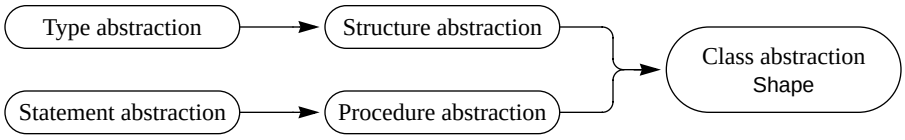


Figure 23.6
Using class abstraction without behavior abstraction to process data for several different kinds of shapes.

Figure 23.7 is the complete interface of the Shape class alluded to in the previous section. Like type Book in Figure 21.17, page 482, type Shape is defined to be a pointer to a record instead of a record. That way, an instance of a Shape can be stored in a circular list CList, which is a list whose nodes have value parts that are pointers to ANYREC.

```
DEFINITION PboxShapeObj;

   TYPE
      Shape = POINTER TO RECORD
         (s: Shape) GetIDString (OUT str: ARRAY OF CHAR), NEW;
         (s: Shape) GetDimensionString (OUT str: ARRAY OF CHAR), NEW;
         (s: Shape) Area (): REAL, NEW;
         (s: Shape) Perimeter (): REAL, NEW;
         (s: Shape) SetRectangleState (length, width: REAL), NEW;
         (s: Shape) SetCircleState (diameter: REAL), NEW
      END;

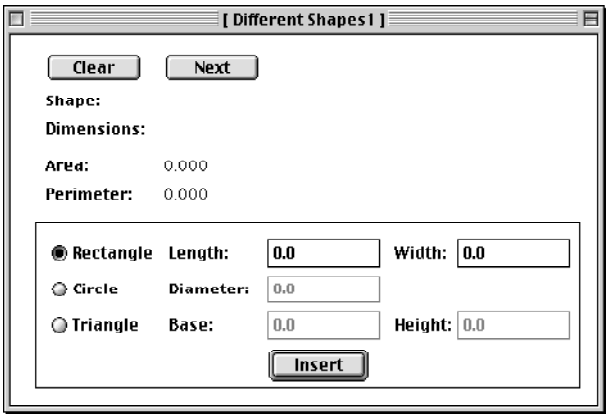
END PboxShapeObj.
```

Figure 23.7
The interface for a Shape class without behavior abstraction.

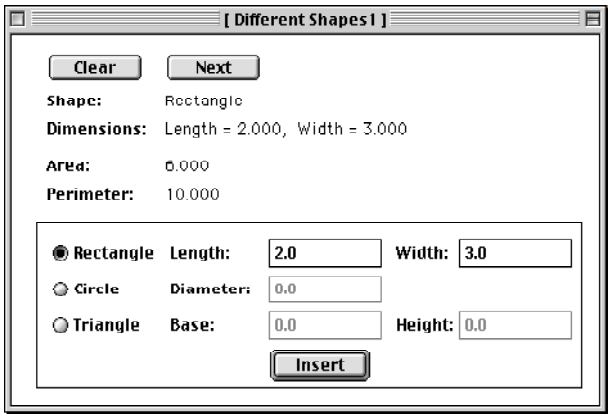
Method GetIdString sets parameter str to a string that can be displayed in a dialog box. For example, if s is a rectangle the method sets str to "Rectangle". Method GetDimensionString sets parameter str to a string that displays the dimensions of the shape to three places past the decimal point. For example, if s is a rectangle with a length of 2 and a width of 3 the method sets str to "Length = 2.000, Width = 3.000". Method Area is a function procedure that returns the area of shape s, and method Perimeter returns the perimeter of s.

The first four methods do not change the state of s. They simply report back information about its state. The last two methods, however, change the state of s. If you supply SetRectangleState with actual parameters 2.0 and 3.0 corresponding to formal parameters length and width, the method will change the state of s to be a rectangle having length 2.0 and width 3.0 regardless of the kind of shape that it was before. Similarly, if you supply SetCircleState with actual parameter 6.0 corresponding to formal parameter diameter, the method will change the state of s to be a circle having diameter 6.0.

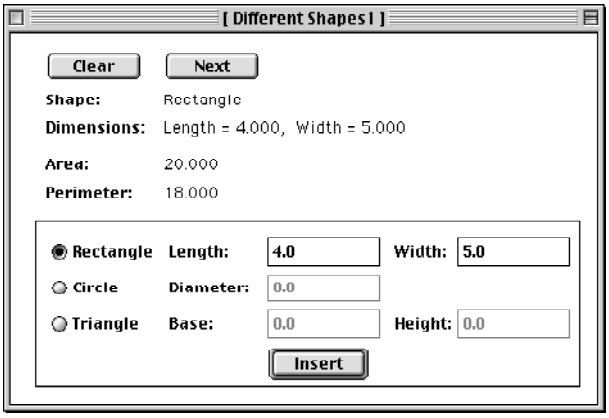
Figure 23.8 shows a sequence of screen shots of a user manipulating a list of shapes. Part (a) shows the dialog box for the first time. The bottom part of the dialog box gives the user the option to enter data for a rectangle, circle, or triangle. As with the circular list for books in Figure 21.15, page 481, the dialog box allows the user to enter data about a shape and store the shape in a circular list. Figure 23.9 is a listing of the program that implements the dialog box of Figure 23.8.



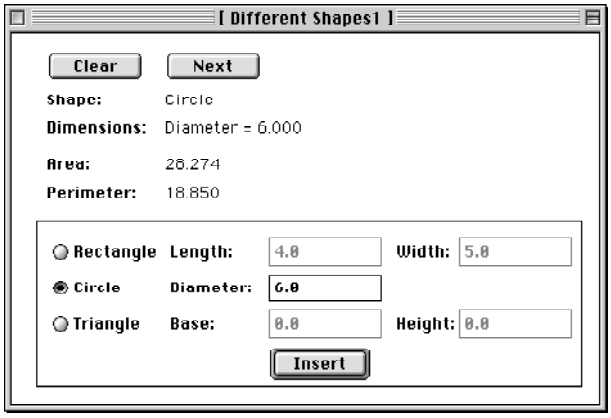
(a) Initial.



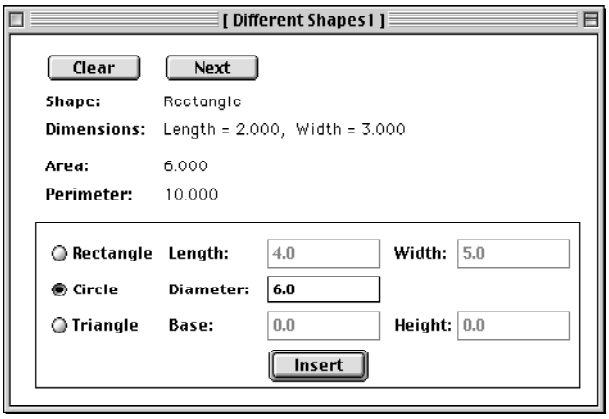
(b) Insert a rectangle.



(c) Insert a rectangle.



(d) Insert a circle.



(e) Press Next.

Figure 23.8
A sequence of screen shots for execution of a program to process shapes.

Figure 23.9
A program that produces the sequence of screen shots in Figure 23.8.

```
MODULE Pbox23A;
   IMPORT Dialog, C := PboxCListADT, S := PboxShapeObj;

   VAR
      d*: RECORD
         idString-, dimensionString- : ARRAY 64 OF CHAR;
         area-, perimeter-: REAL;
         shapeNumber*: INTEGER;
         length*, width*: REAL; (* for rectangle *)
         diameter*: REAL; (* for circle *)
         base*, height*: REAL (* for triangle *)
      END;
      cList: C.CList;

   PROCEDURE ClearDialog;
   BEGIN
      d.idString := ""; d.dimensionString := "";
      d.area := 0.0; d.perimeter := 0.0;
      d.length := 0.0; d.width := 0.0;
      d.diameter := 0.0;
      d.base := 0.0; d.height := 0.0
   END ClearDialog;

   PROCEDURE SetDialog (s: S.Shape);
   BEGIN
      s.GetIDString(d.idString);
      s.GetDimensionString(d.dimensionString);
      d.area := s.Area();
      d.perimeter := s.Perimeter()
   END SetDialog;

   PROCEDURE Clear*;
   BEGIN
      ClearDialog;
      C.Clear(cList);
      Dialog.Update(d)
   END Clear;

   PROCEDURE Next*;
      VAR
         shape: S.Shape;
   BEGIN
      IF ~C.Empty(cList) THEN
         C.GoNext(cList);
         shape := C.NodeContent(cList) (S.Shape);
         SetDialog(shape);
         Dialog.Update(d)
      END
   END Next;
```

```
    PROCEDURE Insert*;
        VAR
            shape: S.Shape;
    BEGIN
        NEW(shape);
        CASE d.shapeNumber OF
        0:
            shape.SetRectangleState(MAX(0.0, d.length), MAX(0.0, d.width)) |
        1:
            shape.SetCircleState(MAX(0.0, d.diameter)) |
        2:
            (* Problem for the student *)
            HALT(100)
        END;
        C.Insert(cList, shape);
        SetDialog(shape);
        Dialog.Update(d)
    END Insert;

    PROCEDURE RectangleGuard* (VAR par: Dialog.Par);
    BEGIN
        par.disabled := d.shapeNumber # 0
    END RectangleGuard;

    PROCEDURE CircleGuard* (VAR par: Dialog.Par);
    BEGIN
        par.disabled := d.shapeNumber # 1
    END CircleGuard;

    PROCEDURE TriangleGuard* (VAR par: Dialog.Par);
    BEGIN
        par.disabled := d.shapeNumber # 2
    END TriangleGuard;

BEGIN
    Clear
END Pbox23A.
```

Figure 23.9
Continued.

The import list

```
IMPORT Dialog, C := PboxCListADT, S := PboxShapeObj;
```

The import abbreviation

sets up a convenient abbreviation scheme. Component Pascal allows you to rename an imported module using the alias symbol :=. This import list renames module PboxCListADT as simply C. Everywhere in this module that you would normally place the name PboxCListADT, you can now place the abbreviation C. The same substitution applies to PboxShapeObj with its abbreviation S. Once you redefine imported module names like this you cannot revert back to the long form.

Example 23.2 In Figure 23.9, the procedure heading

```
PROCEDURE SetDialog (s: S.Shape);
```

would be written

```
PROCEDURE SetDialog (s: PboxShapeObj.Shape);
```

if the import list had not defined the abbreviation S. If you write the procedure heading the second way with the abbreviation defined, however, the module will not compile. ∎

Procedure Clear

Buttons Clear, Next, and Insert are obviously linked to exported procedures Clear, Next, and Insert. Procedure Clear operates the same way the corresponding procedure does in Figure 21.17.

Procedure Next

Procedure Next has a local variable shape of type PboxShapeObj.Shape. The procedure checks if cList is empty, and if it is not calls GoNext to advance it to the next entry. Then it sets shape to the content of the current node by calling NodeContent. Now that shape has the content of the current node the procedure calls SetDialog, passing shape as the actual parameter that corresponds to formal parameter s. The first statement in SetDialog

```
s.GetIDString(d.idString)
```

changes d.IdString to the name of shape s. The dimensions, area, and perimeter fields get set similarly. Finally, after the return to procedure Next, the dialog is updated so the changes in the d interactor will be made visible.

Procedure Insert

Procedure Insert also has a local variable shape of type PboxShapeObj.Shape. To insert a new shape into the linked list based on the user's request, the procedure executes

```
NEW(shape)
```

to allocate a new shape from the heap. The interactor field d.shapeNumber is linked to the set of radio buttons in the dialog box. The level of the button for a rectangle is set to 0, for a circle is set to 1, and for a triangle is set to 2. Procedure Insert uses a CASE statement, testing the value of d.shapeNumber, to determine what shape the user wants to store. If the user wants to store a rectangle as in Figure 23.8(b) d.shapeNumber will have the value 0 and procedure Insert will execute

```
shape.SetRectangleState(MAX(0.0, d.length), MAX(0.0, d.width))
```

The first MAX function returns the maximum of the two real values 0.0 and d.length. Method SetRectangleState has a precondition that neither of its formal parameters can be negative. The purpose of MAX is guarantee that the precondition will be met. If the user enters a negative value for the length or width of the rectangle, zero will be stored instead. Procedure Insert concludes by calling the Insert procedure for the

circular list, setting the output fields in the d interactor, and updating the dialog box to make the changes visible.

Control guards

In Figure 23.8(c), the radio button for the rectangle is on, and fields to input the length and width are available, while those to input the diameter of a circle or the base and height of a triangle are not. An important user-interface design principle is that the user should have a visual cue of those actions which can and cannot be performed on a dialog box. The cue is usually the dimming of those elements that cannot be selected. Figure 23.8(c) has the circle and triangle input fields dimmed while the user can input information about a rectangle. Figure 23.8(d) has the rectangle and triangle fields dimmed while the user can input information about a circle.

A user-interface design principle

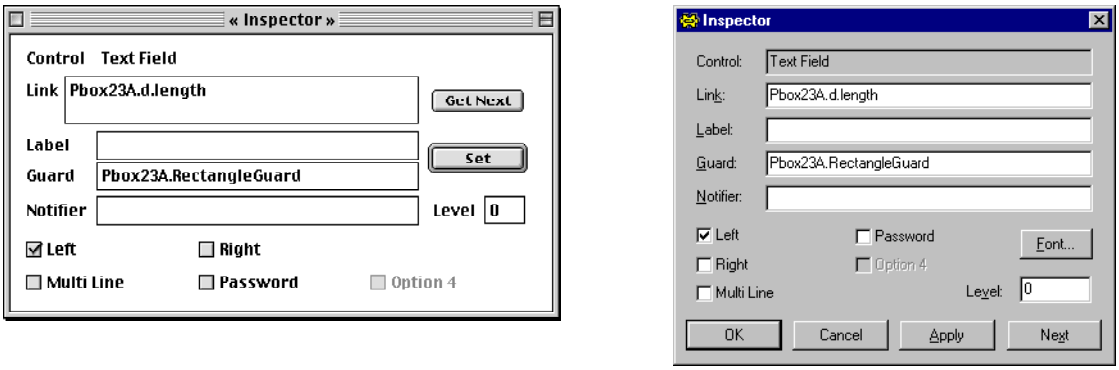
The ease with which the programmer can implement this user-interface design principle is a testament to the power of the BlackBox framework. The technique is based on the fact that BlackBox is a true framework and not just a collection of server modules. Consider the dialog box in Figure 23.8(c). There are three buttons labeled Clear, Next, and Insert. The program to implement the dialog box has three corresponding procedures that are executed when the buttons are pressed. The radio buttons correspond to d.shapeNumber, which will have the value 0 with the Rectangle button pressed. Furthermore, the input and output fields correspond to fields in the d interactor record.

If the user presses the Circle radio button, the effect on the dialog box will be to dim the rectangle input and allow input for the circle as is done in Figure 23.8(d). But, it seems that the only effect of pressing the Circle button is to change the value of d.shapeNumber from 0 to 1. How can you program the dialog box to dim the rectangle input when d.shapeNumber gets the value 1? After all, pressing the radio button does not cause any of your procedures to execute.

The answer is that the BlackBox framework continually monitors the appearance of the dialog box. It can automatically detect the change in d.shapeNumber the instant it occurs. Somehow you need to inform the framework when a field should be dimmed. But you cannot do that by calling a procedure, because the event of pressing a radio button does not cause any of your procedures to execute. The solution is for you to write a procedure that the *framework* calls. Such a procedure is known as a control guard.

Every forms control has a potential control guard that you can link to with the Inspector. You write a guard, which is an exported procedure, in your module. Then use the Inspector to link to the guard. For example, in Figure 23.8(c) one of the forms controls is the input field for the length of a rectangle. The user has entered 4.0 in the field. This control is linked to d.length in the d interactor. Figure 23.10 shows the Inspector for this control. You can see from the Link field of the Inspector that this is indeed the Inspector for d.length. There is a Guard field in the Inspector, in which the programmer has entered Pbox23A.RectangleGuard. This is the control guard, which is an exported procedure in Figure 23.9.

The presence of procedure RectangleGuard may seem strange in Figure 23.9, because nowhere in the module is a call to it. You do not call procedure RectangleGuard; the framework does. When does the framework call the control guard?



(a) MacOS.

(b) MSWindows.

Figure 23.10
Links that must be entered in the Inspector to enable the guards for dimming controls in the dialog box.

Whenever it needs to, namely when you call Dialog.Update and when the user changes the state of the dialog box, like when she presses a radio button. You, as an applications programmer, do not need to concern yourself with those details of when and how the framework calls your guard. All you need to do is provide the entry in the Inspector for the control guard. The framework will see to it that the guard is called at the appropriate times. This arrangement is not that different from the programs you have been writing in BlackBox from the beginning. Most programs implement a dialog box with buttons for the user to click on to initiate an action. Your programs simply supply exported procedures that the framework calls at the appropriate time.

The guard for the rectangle's length input field is

```
PROCEDURE RectangleGuard* (VAR par: Dialog.Par);
BEGIN
   par.disabled := d.shapeNumber # 0
END RectangleGuard;
```

The rectangle's width input field is linked to the same guard. The guard procedure has one formal parameter par whose type is Dialog.Par. It is called by reference, meaning that if this procedure modifies par, the corresponding actual parameter in the calling program will be modified. What is the actual parameter? Some variable in the framework that you do not see, because the framework calls RectangleGuard instead of you calling it.

To investigate the type Dialog.Par you can consult the documentation of Dialog, part of which is shown in Figure 23.11. From the documentation, a variable of type Par has five fields, the first four of which are boolean. The first boolean field is named disabled. If par.disabled is set to false, the corresponding control in the dialog box will not be disabled. That is, it will be available for the user to access. If par.disabled is set to true, the control will be dimmed to show that the user cannot access it. Any attempt at accessing the control will result in no action. The rectangle guard sets par.disabled to the boolean expression

```
d.shapeNumber # 0
```

which will disable the rectangle's width input field if d.shapeNumber is not equal to 0. Because the rectangle's width input field is linked to the same guard it will be disabled under the same circumstances.

```
DEFINITION Dialog;

   TYPE
      Par = RECORD
         disabled: BOOLEAN;
         checked: BOOLEAN;
         undef: BOOLEAN;
         readOnly: BOOLEAN;
         label: String
      END;
```

Figure 23.11
A partial listing of the documentation for the Dialog module.

The Hollywood Principle

The arrangement between BlackBox and the application programmer where the framework calls the programmer's procedures is what distinguishes a true framework from a library of server modules. It is known as the Hollywood Principle, "Don't call us. We'll call you." The ability to program a graphical user interface with so much power and yet so much simplicity from the application programmer's perspective is due to the BlackBox framework as much as it is due to the power and simplicity of the Component Pascal language.

A shape class implementation

Figure 23.13 is the implementation of the shape class whose interface is in Figure 23.7. The exported type Shape is a pointer to a record with four fields as shown in Figure 23.12. The first field is an integer kind, which indicates what kind of shape is stored—0 for rectangle and 1 for circle. The second and third fields are length and width for storing the length and width of a rectangle when kind has value 0. The fourth field is diameter for storing the diameter of a circle when kind has value 1. Modification of the Shape structure to accommodate triangles is left as a problem for the student.

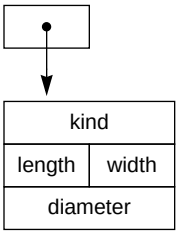


Figure 23.12
The structure of type Shape in Figure 23.13.

```
MODULE  PboxShapeObj;
   IMPORT PboxStrings, Math;

   TYPE
      Shape* = POINTER TO RECORD
         kind: INTEGER;
         length, width: REAL;
         diameter: REAL
      END;
```

Figure 23.13
Implementation of the shape class with class abstraction only. Figure 23.7 shows the interface for this class.

Figure 23.13
Continued.

```
PROCEDURE (s: Shape) GetIDString* (OUT str: ARRAY OF CHAR), NEW;
BEGIN
   CASE s.kind OF
   0:
      str := "Rectangle" |
   1:
      str := "Circle"
   END
END GetIDString;

PROCEDURE (s: Shape) GetDimensionString* (OUT str: ARRAY OF CHAR), NEW;
   VAR
      temp: ARRAY 16 OF CHAR;
BEGIN
   CASE s.kind OF
   0:
      PboxStrings.RealToString(s.length, 1, 3, temp);
      str := "Length = " + temp + ",  ";
      PboxStrings.RealToString(s.width, 1, 3, temp);
      str := str + "Width = " + temp |
   1:
      PboxStrings.RealToString(s.diameter, 1, 3, temp);
      str := "Diameter = " + temp
   END
END GetDimensionString;

PROCEDURE (s: Shape) Area* (): REAL, NEW;
BEGIN
   CASE s.kind OF
   0:
      RETURN s.length * s.width |
   1:
      RETURN Math.Pi() * s.diameter * s.diameter / 4.0
   END
END Area;

PROCEDURE (s: Shape) Perimeter* (): REAL, NEW;
BEGIN
   CASE s.kind OF
   0:
      RETURN 2.0 * (s.length + s.width) |
   1:
      RETURN Math.Pi() * s.diameter
   END
END Perimeter;
```

```
    PROCEDURE (s: Shape) SetRectangleState* (length, width: REAL), NEW;
    BEGIN
        ASSERT((length >= 0.0) & (width >= 0.0), 20);
        s.kind := 0;
        s.length := length;
        s.width := width
    END SetRectangleState;

    PROCEDURE (s: Shape) SetCircleState* (diameter: REAL), NEW;
    BEGIN
        ASSERT(diameter >= 0.0, 20);
        s.kind := 1;
        s.diameter := diameter
    END SetCircleState;

END PboxShapeObj.
```

Figure 23.13
Continued.

Methods SetIdString, GetDimensionString, Area, and Perimeter all have a similar control structure. Each method first determines what kind of shape is stored using a CASE statement on s.kind. If s.kind has value 0, the method processes the data assuming that a rectangle is stored and uses s.length and s.width accordingly. If s.kind has value 1, the method processes the data assuming that a circle is stored and uses s.diameter accordingly.

Methods SetRectangleState and SetCircleState, unlike the previous methods, change the state of s. Each method has a precondition that does not allow the dimensions of the state to be negative. The precondition is implemented with the usual ASSERT statement. The input of the set state methods are values of the dimensions of the shape. The method simply sets the kind field to the integer code for that state and transfers the dimension values to the corresponding fields in the shape's record.

The program of Figure 23.9 stores the shapes in the circular list provided by PboxCListADT. The dialog box in Figure 23.8(e) shows a dialog box where the user has entered a circle of diameter 6.0 and pressed the Next button so that the current shape is a rectangle of length 2.0 and width 3.0. Figure 23.14 shows the corresponding data structure.

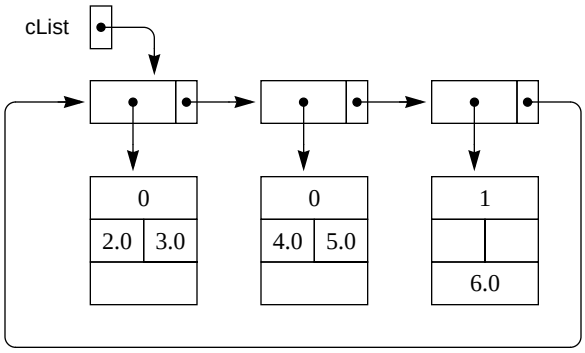


Figure 23.14
The data structure that corresponds to the screen shot of Figure 23.8(e) with the shape of PboxShapeObj.

The difference in syntax for defining and calling a method with class abstraction compared to a procedure with procedure abstraction does not illustrate the power of object-oriented design. After all, there is no inherent benefit to putting an actual parameter in front of a method name instead of enclosing it in parentheses after a function name. The only thing the object-oriented syntax does is to emphasize that functions are bound to classes along with the data. The real power of object-orientation comes with yet another level of abstraction—behavior abstraction.

Behavior abstraction

The program in the previous section processed different shapes using class abstraction. The class Shape in PboxShapeObj is a pointer to a record that contains a kind field to specify what kind of shape is stored in the record. The methods for Shape test the kind field with a CASE statement to determine the appropriate processing to perform. Instead of adopting the viewpoint of class abstraction, where a shape is simply a collection of data and methods that correspond to some specific shape, suppose you take a further step towards abstraction and collect several different shapes together to form an abstract shape. What is common that can be abstracted out?

That is, what do rectangles, circles, and right triangles have in common? They are certainly not all specified by length and width as is the rectangle. A circle, for example, is specified by its diameter. Because dimensions for different objects are specified differently, you cannot include the dimensions in the abstract shape. However, all closed shapes have an area and a perimeter. So, you can at least include those. You must be careful, however, because the algorithm for computing the area of a circle is not the same as the algorithm for computing the area of a right triangle. Even though the abstract shape will specify a method for computing the area and perimeter, it cannot implement it because the algorithm depends on the specific object. Furthermore, each shape has a method to set its ID string and its dimension string.

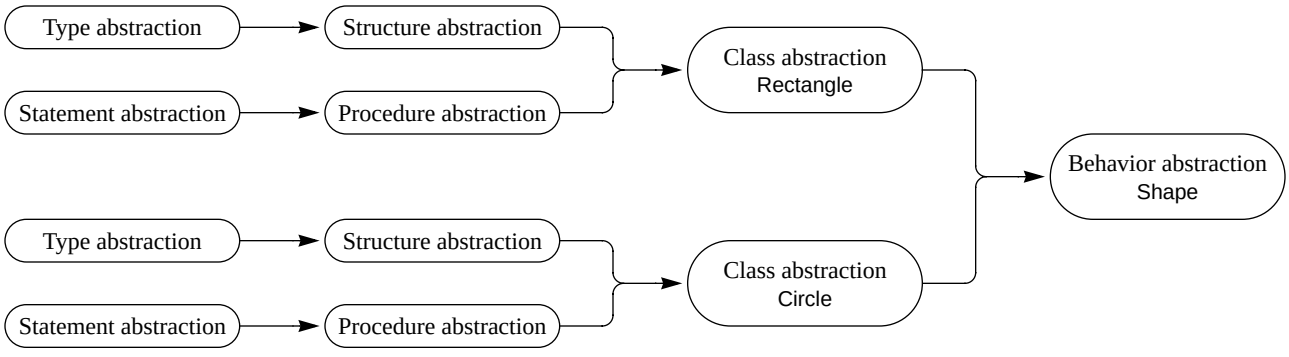


Figure 23.15
Behavior abstraction that combines class abstraction for two different classes into a single abstract class.

Figure 23.15 is a representation of the behavior abstraction process.The figure shows the abstraction processes for data and control culminating in class abstraction for Rectangle and the same abstraction processes culminating in class abstraction

for Circle. Behavior abstraction combines the specific shapes Rectangle and Circle into the abstract class Shape.

Compare Figure 23.15, which shows the abstraction process for a shape using behavior abstraction, with Figure 23.6, which shows the abstraction process using only class abstraction. In Figure 23.6, the concepts of rectangle and circle are merged with the concept of shape. Class Shape is a kind of hybrid, whose data is a combination of fields that must accommodate information for both rectangles and circles and a field kind to tell them apart. However, Figure 23.15 uses class abstraction for each individual shape and behavior abstraction for the abstract shape. This is a significant difference that has major consequences in the program.

Inheritance

In Figure 23.15, the object-oriented relation between class Rectangle and class Shape is that of inheritance. Rectangle inherits from Shape. Similarly, Circle inherits from Shape. The Component Pascal terminology for the inheritance relation is type extension. Type Rectangle is an extension of Shape, which is called the base type. Circle is also an extension of Shape. Figure 23.16 is the interface for PboxShapeAbs, which implements the inheritance relationship between the classes.

Type extension and the base type

```
DEFINITION PboxShapeAbs;

   TYPE
      Shape = POINTER TO ABSTRACT RECORD
         (s: Shape) GetIDString (OUT str: ARRAY OF CHAR), NEW, ABSTRACT;
         (s: Shape) GetDimensionString (OUT str: ARRAY OF CHAR), NEW, ABSTRACT;
         (s: Shape) Area (): REAL, NEW, ABSTRACT;
         (s: Shape) Perimeter (): REAL, NEW, ABSTRACT
      END;

      Rectangle = POINTER TO RECORD (Shape)
         (r: Rectangle) GetIDString (OUT str: ARRAY OF CHAR);
         (r: Rectangle) GetDimensionString (OUT str: ARRAY OF CHAR);
         (r: Rectangle) Area (): REAL;
         (r: Rectangle) Perimeter (): REAL;
         (r: Rectangle) SetState (length, width: REAL), NEW
      END;

      Circle = POINTER TO RECORD (Shape)
         (c: Circle) GetIDString (OUT str: ARRAY OF CHAR);
         (c: Circle) GetDimensionString (OUT str: ARRAY OF CHAR);
         (c: Circle) Area (): REAL;
         (c: Circle) Perimeter (): REAL;
         (c: Circle) SetState (diameter: REAL), NEW
      END;

END PboxShapeAbs.
```

Figure 23.16
The interface for a general geometric shape. that uses behavior abstraction.

In the declaration of the Shape type

```
Shape = POINTER TO ABSTRACT RECORD
```

the word ABSTRACT is a record attribute. It indicates that the class Shape is an abstract class, which means it cannot be instantiated. No variables or fields of such a record can ever exist.

Example 23.3 Suppose Shape is declared as in Figure 23.16 and you have a local variable myShape declared as

```
VAR
   myShape: Shape;
```

The instantiation

```
NEW(myShape)
```

is illegal and will not compile, because Shape is abstract. ∎

Why declare a type if you can never instantiate it? Because an abstract type is not used by itself. Instead, it is a form to be used as a guide for creating concrete types that are extensions of it. You can think of a superclass as a blueprint for the subclasses that inherit from it.

Superclass and subclass

The first method in the abstract shape is

```
(s: Shape) GetIDString (OUT str: ARRAY OF CHAR), NEW, ABSTRACT
```

It has two method attributes, NEW and ABSTRACT. In the same way that an abstract class can never be instantiated, an abstract method can never contain any executable statements and can never be called. Why declare a method if it can never be called? Again, because it is not used by itself. Instead, it is a blueprint for the corresponding method of the subclass. In this case, the blueprint says that the method of the subclass must be named GetIDString and must have one parameter called by result of type ARRAY OF CHAR. A record containing abstract methods must be abstract. The attribute NEW must be used on all newly introduced methods.

Rules for abstract records and abstract methods

The remaining methods of Shape—GetDimensionString, Area, and Perimeter—each have method attributes NEW and ABSTRACT. It is the responsibility of the subclasses to implement the methods using the same method names and signatures, that is, the same number and types of the formal parameters.

In the declaration of type Rectangle

```
Rectangle = POINTER TO RECORD (Shape)
```

the base type is enclosed in parentheses after the reserved word RECORD. The declaration states that Rectangle inherits from Shape; or, Rectangle is the subclass and Shape is the superclass; or, Rectangle is an extension of Shape. The idea of inheritance is that the specific inherits from the general. Shape is general, and Rectangle is

specific. The fundamental class assignment rule is that you can assign the specific to the general, but you cannot assign the general to the specific.

The fundamental class assignment rule

Example 23.4 If myShape is a formal paramter of type Shape and myRectangle is a local variable of type Rectangle, then the assignment

```
myShape := myRectangle
```

is legal, but the assignment

```
myRectangle := myShape
```

is not legal. ∎

The first method of class Rectangle is

```
(r: Rectangle) GetIDString (OUT str: ARRAY OF CHAR)
```

It has the same name as the first method of class Shape and the same signature. The only difference is that the type Rectangle in the receiver (r: Rectangle) is a subclass of the type Shape in the receiver (s: Shape). Furthermore, the method GetIDString for Rectangle is neither ABSTRACT nor NEW. Because it is not ABSTRACT, it has statements and can be called. Because it is not NEW, it is based on a previously declared method. All these characteristics indicate that GetIDString for Rectangle is a concrete implementation of GetIDString for Shape. Methods GetDimensionString, Area, and Perimeter have the same characteristics as GetIDString. They are all concrete implementations of the corresponding methods of Shape.

The last method of class Rectangle, however,

```
(r: Rectangle) SetState (length, width: REAL), NEW
```

is not an implementation of a previously declared method. It is NEW but not ABSTRACT. There is no corresponding method in the superclass of Rectangle.

The declaration of class Circle mirrors that of class Rectangle. Circle is a subclass of Shape. It implements the four methods declared in Shape—GetIDString, GetDimensionString, Area, Perimeter—and declares its own fifth method SetState that is not an implementation of a previously declared abstract method.

Comparing the interface in Figure 23.16 with that in Figure 23.7 it should be obvious that each method of PboxShapeAbs does the same processing as that in the corresponding method of PboxShapeObj. Namely, GetIDString gives str a string value that describes the name of the shape, GetDimensionString gives str a string value that describes the dimensions of the shape with three places past the decimal, Area returns the area of the shape, and Perimeter returns the perimeter of the shape. SetState for Rectangle sets the state to a rectangle and SetState for Circle sets the state for a circle. You can see that the SetState methods cannot be specified by the abstract class Shape, because the parameter lists are different for rectangles and circles. Signatures must be identical between superclass and subclass methods, which would be impossible for SetState.

Polymorphism

Figure 23.17 is a program that implements the same dialog box as that shown in Figure 23.8. From the user's perspective, there is no difference in the behavior of the dialog box between the two versions. However, the program of Figure 23.17 uses behavior abstraction with polymorphism.

```
MODULE Pbox23B;
   IMPORT Dialog, C := PboxCListADT, S := PboxShapeAbs;

   VAR
      d*: RECORD
         idString-, dimensionString- : ARRAY 64 OF CHAR;
         area-, perimeter-: REAL;
         shapeNumber*: INTEGER;
         length*, width*: REAL; (* for rectangle *)
         diameter*: REAL; (* for circle *)
         base*, height*: REAL (* for triangle *)
      END;
      cList: C.CList;

   PROCEDURE ClearDialog;
   BEGIN
      d.idString := ""; d.dimensionString := "";
      d.area := 0.0; d.perimeter := 0.0;
      d.length := 0.0; d.width := 0.0;
      d.diameter := 0.0;
      d.base := 0.0; d.height := 0.0
   END ClearDialog;

   PROCEDURE SetDialog (s: S.Shape);
   BEGIN
      s.GetIDString(d.idString);
      s.GetDimensionString(d.dimensionString);
      d.area := s.Area();
      d.perimeter := s.Perimeter()
   END SetDialog;

   PROCEDURE Clear*;
   BEGIN
      ClearDialog;
      C.Clear(cList);
      Dialog.Update(d)
   END Clear;
```

Figure 23.17
A program that produces the same output as the one in Figure 23.9 but that uses behavior abstraction.

Figure 23.17
Continued.

```
PROCEDURE Next*;
   VAR
      shape: S.Shape;
BEGIN
   IF ~C.Empty(cList) THEN
      C.GoNext(cList);
      shape := C.NodeContent(cList) (S.Shape);
      SetDialog(shape);
      Dialog.Update(d)
   END
END Next;

PROCEDURE Insert*;
   VAR
      rectangle: S.Rectangle;
      circle: S.Circle;
BEGIN
   CASE d.shapeNumber OF
   0:
      NEW(rectangle);
      rectangle.SetState(MAX(0.0, d.length), MAX(0.0, d.width));
      C.Insert(cList, rectangle);
      SetDialog(rectangle) |
   1:
      NEW(circle);
      circle.SetState(MAX(0.0, d.diameter));
      C.Insert(cList, circle);
      SetDialog(circle) |
   2:
      (* Problem for the student *)
   END;
   Dialog.Update(d)
END Insert;

PROCEDURE RectangleGuard* (VAR par: Dialog.Par);
BEGIN
   par.disabled := d.shapeNumber # 0
END RectangleGuard;

PROCEDURE CircleGuard* (VAR par: Dialog.Par);
BEGIN
   par.disabled := d.shapeNumber # 1
END CircleGuard;

PROCEDURE TriangleGuard* (VAR par: Dialog.Par);
BEGIN
   par.disabled := d.shapeNumber # 2
END TriangleGuard;
```

```
BEGIN
   Clear
END Pbox23B.
```

Figure 23.17
Continued.

A comparison of modules Pbox23A in Figure 23.9 and Pbox23B in Figure 23.17 shows little apparent difference except for procedure Insert. Pbox23B.Insert has two local variables—rectangle with type PboxShapeABS.Rectangle and circle with type PboxShapeAbs.Circle. Suppose the user enters information about a rectangle and clicks the Insert button. The CASE statement determines that the value of d.ShapeNumber is 0 and executes

```
NEW(rectangle)
```

In Pbox23A, the corresponding statement is NEW(shape) where shape has type PboxShapeObj.Shape. But in Pbox23B, that would be impossible because PboxShapeAbs.Shape is abstract and you cannot instantiate an abstract class. That is why Pbox23B.Insert needs two local variables each with a concrete type instead of one local variable with an abstract type.

The next statement

```
rectangle.SetState(MAX(0.0, d.length), MAX(0.0, d.width))
```

calls method SetState. But two SetState methods are imported from PboxShapeAbs—one for a rectangle and one for a circle. How does Component Pascal know which one to call? By the type of the receiver. With this call, the actual parameter rectangle has type Rectangle. So, Component Pascal calls the SetState method whose receiver has the same type. The effect of the call is to set the dimensions of rectangle according to the user input.

Then the call

```
C.Insert(cList, rectangle)
```

executes procedure PboxCListADT.Insert. Actual parameter rectangle is a pointer to a record extended from PboxShapeAbs.Shape while the corresponding formal parameter val is a pointer to ANYREC. This is an example of the fundamental class assignment rule applied to parameters. The formal parameter can be general and the actual parameter specific, but the formal parameter cannot be specific and the actual parameter general. In this example, Shape inherits from ANYREC, and Rectangle inherits from Shape. Therefore, Rectangle, which is specific, inherits from ANYREC, which is general.

The fundamental class assignment rule applied to parameters

The next statement in Pbox23B.Insert

```
SetDialog(rectangle)
```

calls procedure Pbox23B.SetDialog. Here again the fundamental class assignment rule applied to parameters comes into play. The formal parameter s has type Shape, which is general, and the actual parameter rectangle has type Rectangle, which is

specific. As it is with class assignments, formal parameter s now has two types. Its static type is Shape, while its dynamic type is Rectangle.

The first statement in Pbox23B.SetDialog is

```
s.GetIDString(d.idString)
```

Polymorphism

which illustrates behavior abstraction with polymorphism. The question is, How does Component Pascal know which GetIDString to call? The situation is different from the call to SetState. In that case, the actual parameter is rectangle, which has type Rectangle. Component Pascal can determine from the interface in Figure 23.16 that there is a method with a Rectangle receiver and call that one. But in this case, the actual parameter is s, which has type Shape. The interface shows that the GetIDString with a Shape receiver is abstract.

```
(s: Shape) GetIDString (OUT str: ARRAY OF CHAR), NEW, ABSTRACT
```

Therefore, it has no statements and cannot be called.

The solution to this problem is at the heart of polymorphism. The only methods that can be called are the concrete ones

```
(r: Rectangle) GetIDString (OUT str: ARRAY OF CHAR)
```

and

```
(c: Circle) GetIDString (OUT str: ARRAY OF CHAR)
```

which are the methods that implement the corresponding abstract method. But the question remains, How does Component Pascal know which of the concrete methods to call? It knows, not from the *static* type of s at compile time, but from its *dynamic* type at execution time. In this scenario, because the dynamic type of s is Rectangle it calls the GetIDString whose receiver has type Rectangle. The selection of one method among several identically named methods based on the dynamic type of the actual parameter for the receiver is called polymorphic dispatch.

Polymorphic dispatch

The remaining method calls in PBox23B.SetDialog are all based on polymorphic dispatch. The formal parameter s has static type Shape, but dynamic type Rectangle. Therefore, the corresponding method implemented for Rectangle gets called. Suppose the user were entering data for a circle. In that case Pbox23B.Insert would execute

```
NEW(circle)
```

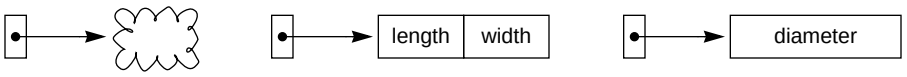
followed by setting the state of circle and inserting it into cList. Then, the procedure call

```
SetDialog(circle)
```

would give formal parameter s in procedure SetDialog the dynamic type Circle. The method calls would all be to the ones with a Circle receiver.

An abstract shape class implementation

Figure 23.19 is the implementation of the shape class whose interface is in Figure 23.16. The exported type Shape is a pointer to an abstract record with no fields. Figure 23.18(a) depicts the abstraction as a cloud. Class Rectangle inherits from Shape. Its record has two fields, length and width, for storing the length and width of a rectangle as Figure 23.18(b) shows. Class Circle also inherits from Shape. Its record has one field, diameter, for storing the diameter of a circle. Compare Figure 23.18 with Figure 23.12 where only class abstraction is used without behavior abstraction. With behavior abstraction there is no need for the kind field to determine what kind of shape is being processed.



(a) Shape. **(b)** Rectangle. **(c)** Circle.

Figure 23.18
The structure of type Shape, Rectangle, and Circle in Figure 23.19.

```
MODULE PboxShapeAbs;
   IMPORT PboxStrings, Math;

   TYPE
      Shape* = POINTER TO ABSTRACT RECORD END;

      Rectangle* = POINTER TO RECORD (Shape)
         length, width: REAL
      END;

      Circle* = POINTER TO RECORD (Shape)
         diameter: REAL
      END;

(* -------------------- *)
   PROCEDURE (s: Shape) GetIDString* (OUT str: ARRAY OF CHAR), NEW, ABSTRACT;

   PROCEDURE (r: Rectangle) GetIDString* (OUT str: ARRAY OF CHAR);
   BEGIN
      str := "Rectangle"
   END GetIDString;

   PROCEDURE (c: Circle) GetIDString* (OUT str: ARRAY OF CHAR);
   BEGIN
      str := "Circle"
   END GetIDString;
```

Figure 23.19
Implementation of the abstract class Shape whose interface is in Figure 23.16.

```
(* -------------------- *)
   PROCEDURE (s: Shape) GetDimensionString* (OUT str: ARRAY OF CHAR), NEW, ABSTRACT;

   PROCEDURE (r: Rectangle) GetDimensionString* (OUT str: ARRAY OF CHAR);
      VAR
         temp: ARRAY 16 OF CHAR;
   BEGIN
      PboxStrings.RealToString(r.length, 1, 3, temp);
      str := "Length = " + temp + ",  ";
      PboxStrings.RealToString(r.width, 1, 3, temp);
      str := str + "Width = " + temp
   END GetDimensionString;

   PROCEDURE (c: Circle) GetDimensionString* (OUT str: ARRAY OF CHAR);
      VAR
         temp: ARRAY 16 OF CHAR;
   BEGIN
      PboxStrings.RealToString(c.diameter, 1, 4, temp);
      str := "Diameter = " + temp
   END GetDimensionString;

(* -------------------- *)
   PROCEDURE (s: Shape) Area* (): REAL, NEW, ABSTRACT;

   PROCEDURE (r: Rectangle) Area* (): REAL;
   BEGIN
      RETURN r.length * r.width
   END Area;

   PROCEDURE (c: Circle) Area* (): REAL;
   BEGIN
      RETURN Math.Pi() * c.diameter * c.diameter / 4.0
   END Area;

(* -------------------- *)
   PROCEDURE (s: Shape) Perimeter* (): REAL, NEW, ABSTRACT;

   PROCEDURE (r: Rectangle) Perimeter* (): REAL;
   BEGIN
      RETURN 2.0 * (r.length + r.width)
   END Perimeter;

   PROCEDURE (c: Circle) Perimeter* (): REAL;
   BEGIN
      RETURN Math.Pi() * c.diameter
   END Perimeter;
```

Figure 23.19
Continued.

```
(* -------------------- *)
    PROCEDURE (r: Rectangle) SetState* (length, width: REAL), NEW;
    BEGIN
        ASSERT((length >= 0.0) & (width >= 0.0), 20);
        r.length := length;
        r.width := width
    END SetState;

    PROCEDURE (c: Circle) SetState* (diameter: REAL), NEW;
    BEGIN
        ASSERT(diameter >= 0.0, 20);
        c.diameter := diameter
    END SetState;

END PboxShapeAbs.
```

Figure 23.19
Continued.

Method GetIDString for PboxShapeAbs.Rectangle has only one assignment statement

```
str := "Rectangle"
```

and GetIDString for PboxShapeAbs.Circle also has only one assignment statement

```
str := "Circle"
```

Contrast this state of affairs with the GetIDString for PboxShapeObj.Shape in Figure 23.13

```
CASE s.kind OF
0:
    str := "Rectangle" |
1:
    str := "Circle"
END
```

which uses the kind field to determine which shape is being processed. Without behavior abstraction, you need a CASE statement to process a shape. With behavior abstraction you do not.

The basic characteristic of abstraction is hidden detail. With behavior abstraction, the details of selecting what kind of shape to process are hidden. Rather than use a CASE statement to select the processing within a single method, the processing for specific shapes is separated into different methods, one for each kind of shape, which are then called with polymorphic dispatch. Of course, hiding the detail in a lower level of abstraction does not eliminate the detail. Component Pascal must maintain the equivalent of a kind field behind the scenes. It stores data to identify the specific concrete class each time the program executes the NEW procedure to instantiate an object. The object's internal "kind" data is consulted during execution time to determine the dynamic type of the object.

Now, consider the implications of polymorphism in a software project with doz-

ens of modules and thousands of lines of code. The software is always in a state of flux with updates and revisions carried out continuously to satisfy the customers and keep up with the competition. Without behavior abstraction in PboxShapeObj, what does it take to add another shape like a triangle? The answer is that you must modify every method in PboxShapeObj that processes a general shape by adding an additional case for the triangle to the CASE statement. In a large software project, the required modifications for a similar revision could be extensive.

With behavior abstraction in PboxShapeAbs, what does it take to add another shape like a triangle? The answer is, You do not need to modify PboxShapeAbs at all! You can simply package your additional shape in its own module, which imports PboxShapeAbs. There is nothing in Component Pascal to prevent you from declaring a subclass in one module whose superclass is in another module. So, with behavior abstraction you do not modify existing code. You simply add code for additional features. The ability to extend an application by adding code instead of modifying existing code is probably the most important benefit of object-oriented programming. It permits a company to design an abstract class with a few concrete classes and have third-party developers write their own concrete classes to enhance the product. The plug-ins for Web browsers are implemented with this idea. For this approach to software development to be effective, the original abstract classes must be well designed.

The most important benefit of object-oriented programming

Another benefit of behavior abstraction over simple class abstraction is the savings in space for the attributes of a object. Figure 23.20 shows the data structure for the circular linked list that corresponds to Figure 23.14. With class abstraction only, you must allocate unused space for a diameter even if the shape is a rectangle, or you must allocate unused space for a width and length even if the shape is a circle. With behavior abstraction, you allocate only enough space for the attributes that are required for the object.

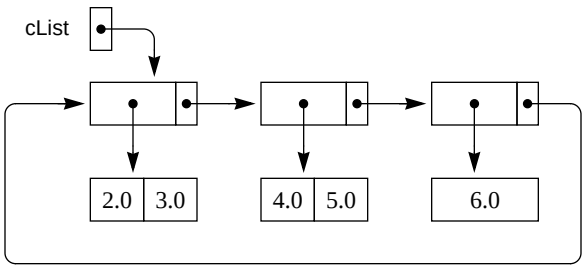


Figure 23.20
The data structure that corresponds to Figure 23.14, but with the abstract shape of PboxShapeABS.

Unified Modeling Language

Figure 23.21 is a Unified Modeling Language (UML) class diagram of the classes declared in PboxShapeAbs. The UML standard specifies several diagrams other than class diagrams. Each box in a class diagram represents a class. A box has three compartments. The name of the class is always in the top compartment in a bold typeface as in **Rectangle**. The name of an abstract class is slanted as in ***Shape***. The

second compartment contains the attributes, and the third compartment contains the operations. The names of abstract methods are slanted as in *GetIDString* in class Shape.

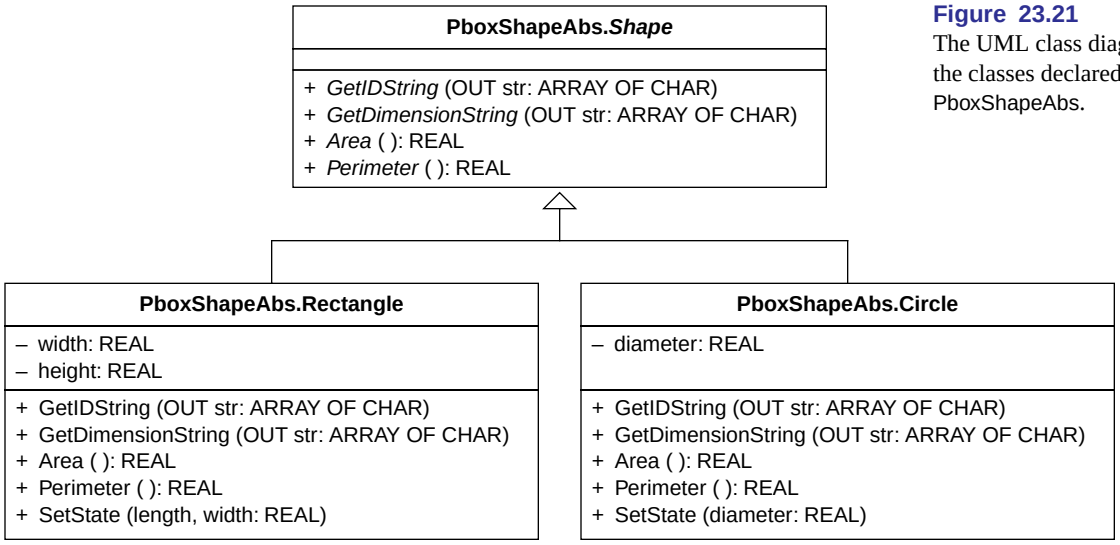
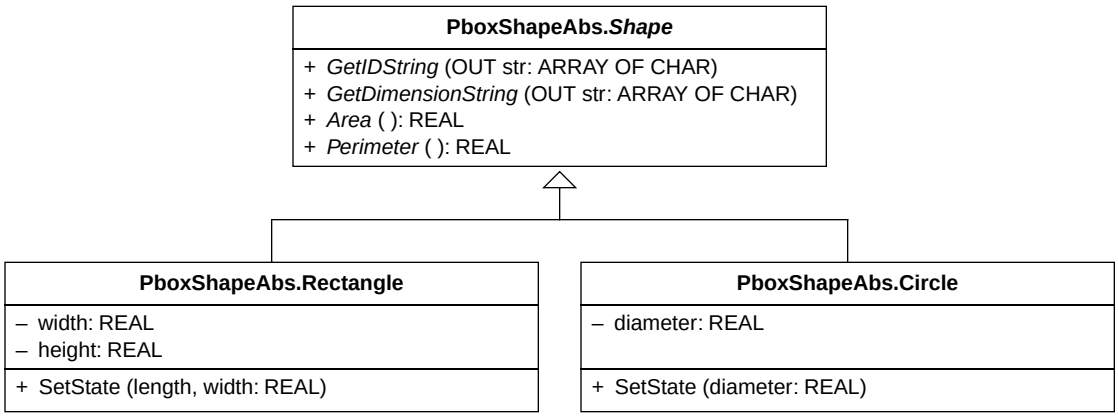


Figure 23.21
The UML class diagram for the classes declared in PboxShapeAbs.

(a) The full version of the class diagram.



(b) The abbreviated version of the class diagram.

In UML, the receiver of a method is not shown, because it can be inferred from the class. Nor are the method attributes listed. The ABSTRACT attribute can be inferred from the slanted type. In UML terminology, items that are exported read/write are called public and are preceded by the plus symbol +. Items that are not exported are called private and are preceded by the minus symbol –. There is no UML standard for items that are exported read-only.

Example 23.5 The method heading for GetIDString in Component Pascal is

```
PROCEDURE (s: Shape) GetIDString* (OUT str: ARRAY OF CHAR), NEW, ABSTRACT;
```

The receiver is (s: Shape). The method attributes are NEW and ABSTRACT. The corresponding heading in the UML class diagram is

+ *GetIDString* (OUT str: ARRAY OF CHAR)

The receiver is missing, but its type can be inferred in Figure 23.21 because it is in the Shape class box. The plus sign indicates that the method is exported. The slanted type for the name implies that the method is abstract. ∎

The triangle symbol  is the UML notation for inheritance. The tip of the triangle points to the superclass and the other end of the triangle is connected to the subclasses. In this book, the superclass will always be an abstract class with abstract methods. Each concrete class will implement its own version of each abstract method. Figure 23.21(a) shows the full UML class diagram. The abstract class Shape has four abstract methods—GetIDString, GetDimensionString, Area, and Perimeter. Both class Rectangle and Circle implement each of these methods. Rather than repeat the abstract methods in each concrete class, this book will use an abbreviated version of the UML class diagram where the corresponding concrete methods are omitted as in Figure 23.21(b). It will be assumed that each abstract method of the superclass is implemented by each concrete subclass.

Class composition

Object-oriented design consists of defining several objects and establishing the relationships between them. Inheritance is one way that objects can be related and class composition is another. Inheritance is frequently described as the "is-a" relationship because the subclass "is a" superclass. For example, in the previous section a circle is a shape. In contrast to inheritance, class composition is described as the "has-a" relationship.

The is-a relationship

The has-a relationship

The program in this section illustrates class composition with a Pizza class. There are two kinds of pizzas, rectangular and circular. Because Shape is a class, and Pizza is a class, and a pizza has a shape, the relationship between the two is one of class composition. The Pizza class is composed of the Shape class.

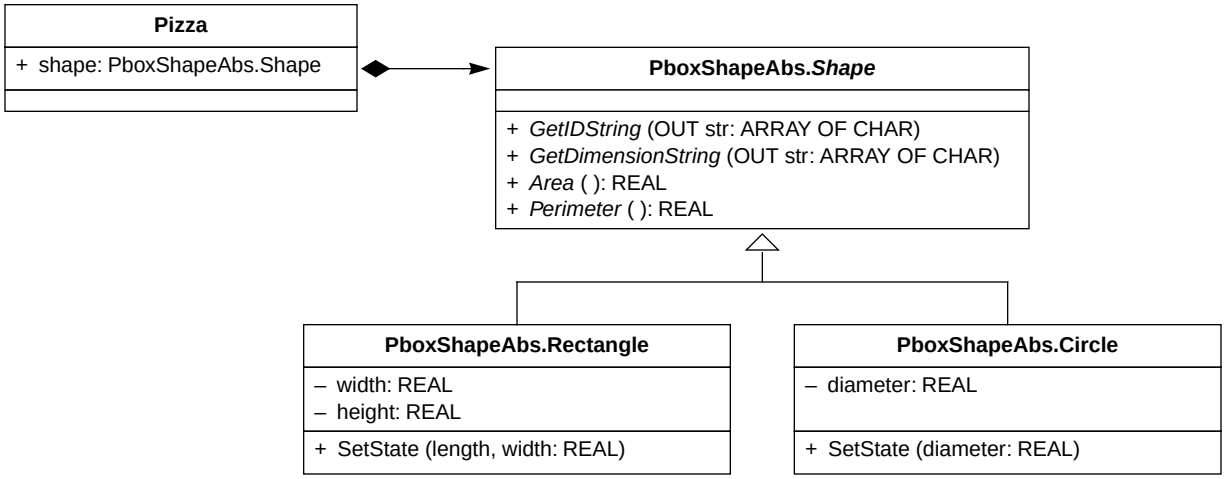
An arrow with a diamond tail  is the UML symbol for class composition. In a UML class diagram, the diamond tail touches the containing class, and the arrowhead touches the class that it contains. There is no new Component Pascal syntax to learn for class composition. Because a class is a record with various fields, to use class composition you simply put the contained class in the field of the class that you want to contain it.

Example 23.6 Suppose you want to define a class named Pizza that contains a Shape class. In Component Pascal, you would declare

```
Pizza* = POINTER TO RECORD
   shape*: PboxShapeAbs.Shape
END;
```

Figure 23.22 shows the corresponding UML class diagram. The shape field of class Pizza is in the attribute box of the Pizza class. The Pizza class contains the Shape class. So, the diamond tail in the UML class diagram touches the Pizza class box, and the arrowhead touches the Shape class box.

Figure 23.22
The UML class diagram for a Pizza class that has a Shape.



To access an element of a composed class, use the standard period "." notation for accessing the field of a record.

Example 23.7 Suppose you have a variable myPizza of type Pizza as in Figure 23.22 and an output text field in your dialog box linked to d.dimensionString. You want to display the dimensions of your pizza in your dialog box. Then, myPizza.shape is a Shape with method GetDimensionString. The statement

```
myPizza.shape.GetIDString(d.dimensionString)
```

calls the GetIDString method polymorphically to set d.dimensionString to the dimensions of the shape of myPizza.

A Pizza class

In real life, a pizza has more than just a shape. It also has a crust, a topping, and a price. The crust for the Pizza class in this section can be thick or thin. The topping can be vegetarian or pepperoni, and either topping can have extra cheese. The price of the pizza depends on all these characteristics. Figure 23.23 shows a dialog box for

a program that implements a Pizza class that has a shape, a topping with possible extra cheese, and a crust. It is for a restaurant where you can order a custom pizza with any shape, any size, any topping, and any crust. The program computes the price according to the pizza specification. The figure shows an order that has been entered for a rectangular pizza, 20 × 30 cm, vegetarian, thick crust. The price is computed as 7.63. The user is about to enter an order for a circular pizza, 25 cm diameter, pepperoni, thick crust with extra cheese.

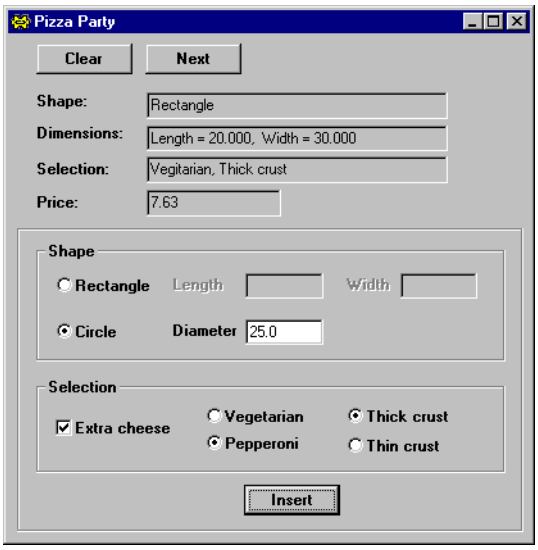


Figure 23.23
The dialog box for a program that uses the Pizza class.

The price of a pizza is determined by a fixed cost of 2.50 regardless of the shape or selection plus a variable cost that depends on the ingredients. A vegetarian pizza has a variable cost of 0.0045 per square cm, and a pepperoni pizza has a variable cost of 0.0065 per square cm. An order with extra cheese adds 0.0010 per square cm, so that the vegetarian and pepperoni toppings are 0.0055 and 0.0075 respectively. Thick crust costs 0.0030 and thin crust 0.0020. The final price is determined by adding a tax of 0.09 to the fixed plus variable cost.

Example 23.8 In Figure 23.23, the area of the rectangular pizza is

$$20 \times 30 \;=\; 600 \text{ cm}^2$$

So, the price is computed as

$$(2.50 + 0.0045 \times 600 + 0.0030 \times 600) \times 1.09 \;=\; 7.63$$

If the order had been with extra cheese the Selection would display “Vegetarian, Extra cheese, Thick crust”, and the price would be computed as

$$(2.50 + 0.0055 \times 600 + 0.0030 \times 600) \times 1.09 \;=\; 8.28$$ ∎

Figure 23.24 is the UML class diagram for class Pizza in module PboxPizza. Because a pizza "has a" shape, and a pizza has a topping, and a pizza has a crust, class composition is used to include each of these three constituents in class Pizza. In the same way that Shape is an abstract class with concrete subclasses Rectangle and Circle, Topping is an abstract class with concrete subclasses Vegetarian and Pepperoni. Inheritance is the relation between class Pepperoni and class Topping, because pepperoni "is a" topping. Similarly, Crust is an abstract class with concrete subclasses Thick and Thin.

The abstract class Topping has boolean attribute extraCheese, which is true if the customer wants extra cheese on his topping and false otherwise. It is possible to duplicate the extraCheese attribute in classes Vegetarian and Pepperoni and not have it in Topping. It is usually best, however, to have those characteristics that are common to a set of classes appear only once in a more general class. Accordingly, the extraCheese attribute appears only once in class Topping.

A subclass inherits the attributes of its superclass.

When any subclass inherits from any superclass, the subclass inherits all the attributes of the superclass. An object of the subclass accesses the attributes of the superclass as if they were all declared as fields of the subclass. That is, you use the period "." between the name of the object and the field to access the field.

Example 23.9 Suppose you declare

```
VAR
   myPepperoni: Pepperoni;
```

where the classes are declared as in the UML class diagram of Figure 23.24. You have allocated myPepperoni from the heap with

```
NEW(myPepperoni)
```

and you want to set the field extraCheese in the Topping superclass to FALSE. The statement

```
myPepperoni.extraCheese := FALSE
```

performs the assignment. It is as if extraCheese is an attribute of myPepperoni directly, even though it is an attribute of the Topping superclass. ∎

Example 23.10 Suppose you declare

```
VAR
   myPizza: Pizza;
```

where the classes are declared as in the UML class diagram of Figure 23.24. If you want to set extraCheese to FALSE for myPizza, use the usual technique for class composition. Assuming a concrete topping has been allocated from the heap, the following statement performs the assignment.

```
myPizza.topping.extraCheese := FALSE
```∎

Figure 23.24
The UML class diagram for class PboxPizza.Pizza.

Classes Vegetarian and Pepperoni each implement abstract method GetToppingString. The Vegetarian version of GetToppingString sets str to "Vegetarian" if extraCheese is false, and to "Vegetarian, Extra cheese" otherwise. Similarly, the Pepperoni version sets str to "Pepperoni" or "Pepperoni, Extra cheese". Method ToppingCost returns the variable cost per square cm for each topping, taking into account whether extra cheese is ordered. For the Vegetarian class, ToppingCost

returns 0.0065 without extra cheese or 0.0075 with extra cheese.

The Thick and Thin classes are simpler because their methods do not depend on any attributes. The Thick version of GetCrustString always sets str to "Thick crust" and the Thin version always sets it to "Thin crust". The Thick version of CrustCost always returns 0.0030 and the Thin version always returns 0.0020.

Figure 23.25 shows the implementation of the PboxPizza.Pizza class. Most of the implementation is left as a problem for the student. Your solution should translate the UML design of Figure 23.24 into Component Pascal code.

```
MODULE PboxPizza;
   IMPORT PboxShapeAbs;

   TYPE
      Topping* = POINTER TO ABSTRACT RECORD
         (* Problem for the student. *)
      END;

      (* Topping subclasses, Problem for the student. *)
      (* Crust class and subclasses, Problem for the student. *)

      Pizza* = POINTER TO RECORD
         shape*: PboxShapeAbs.Shape;
         topping*: Topping;
         (* Problem for the student *)
      END;

(* ---------------------- *)
   PROCEDURE (t: Topping) GetToppingString* (OUT str: ARRAY OF CHAR), NEW, ABSTRACT;

   (* GetToppingString, Problem for the student. *)

(* ---------------------- *)
   PROCEDURE (t: Topping) ToppingCost* (): REAL, NEW, ABSTRACT;

   (* ToppingCost, Problem for the student. *)

(* ---------------------- *)

   (* GetCrustString, Problem for the student *)

(* ---------------------- *)

   (* CrustCost, Problem for the student *)

END PboxPizza.
```

Figure 23.25
Implementation of the Pizza class whose UML class diagram is in Figure 23.24.

Figure 23.26 shows the program for the dialog box of Figure 23.23. It imports, among other modules, PboxPizza. As with PboxPizza, parts of the module are left as a problem for the student at the end of the chapter. Procedure setDialog requires a

local temporary array of characters to set the selection string, because it is the concatenation of the topping string, the string “, “, and the crust string. In the same way that procedure Insert requires local concrete classes for a rectangle and a circle, it requires local concrete classes for vegetarian and pepperoni toppings, and thick and thin crusts.

```
MODULE Pbox23C;
   IMPORT Dialog, C := PboxCListADT, S := PboxShapeAbs, P := PboxPizza;

   CONST
      basePrice = 2.50;
      tax = 0.09;

   VAR
      d*: RECORD
         shapeString-, dimensionString-, selectionString-: ARRAY 64 OF CHAR;
         price-: REAL;
         shapeNumber*: INTEGER;
         length*, width*: REAL; (* for rectangle *)
         diameter*: REAL; (* for circle *)
         extraCheese*: BOOLEAN;
         toppingNumber*, crustNumber*: INTEGER
      END;
      cList: C.CList;

   PROCEDURE ClearDialog;
   BEGIN
      d.shapeString := ""; d.dimensionString := ""; d.selectionString := "";
      d.price := 0.0;
      d.shapeNumber := 0;
      d.length := 0.0; d.width := 0.0;
      d.diameter := 0.0;
      d.extraCheese := FALSE;
      d.toppingNumber := 0; d.crustNumber := 0
   END ClearDialog;

   PROCEDURE SetDialog (pz: P.Pizza);
      VAR
         tempStr: ARRAY 32 OF CHAR;
   BEGIN
      pz.shape.GetIDString(d.shapeString);
      pz.shape.GetDimensionString(d.dimensionString);
      (* Problem for the student *)
   END SetDialog;

   PROCEDURE Clear*;
   BEGIN
      ClearDialog;
      C.Clear(cList);
      Dialog.Update(d)
   END Clear;
```

Figure 23.26
The program for the dialog box of Figure 23.23.

Figure 23.26
Continued.

```
PROCEDURE Next*;
    VAR
        pizza: P.Pizza;
BEGIN
    IF ~C.Empty(cList) THEN
        C.GoNext(cList);
        pizza := C.NodeContent(cList) (P.Pizza);
        SetDialog(pizza);
        Dialog.Update(d)
    END
END Next;

PROCEDURE Insert*;
    VAR
        pizza: P.Pizza;
        rectangle: S.Rectangle;
        circle: S.Circle;
        (* Problem for the student. *)
BEGIN
    NEW(pizza);
    CASE d.shapeNumber OF
    0:
        NEW(rectangle);
        rectangle.SetState(MAX(0.0, d.length), MAX(0.0, d.width));
        pizza.shape := rectangle |
    1:
        NEW(circle);
        circle.SetState(MAX(0.0, d.diameter));
        pizza.shape := circle
    END;
    (* CASE d.toppingNumber, Problem for the student *)
    (* d.extraCheese, Problem for the student *)
    (* CASE d.crustNumber, Problem for the student *)
    C.Insert(cList, pizza);
    SetDialog(pizza);
    Dialog.Update(d)
END Insert;

PROCEDURE RectangleGuard* (VAR par: Dialog.Par);
BEGIN
    par.disabled := d.shapeNumber # 0
END RectangleGuard;

PROCEDURE CircleGuard* (VAR par: Dialog.Par);
BEGIN
    par.disabled := d.shapeNumber # 1
END CircleGuard;

PROCEDURE TriangleGuard* (VAR par: Dialog.Par);
BEGIN
    par.disabled := d.shapeNumber # 1
END TriangleGuard;
```

```
BEGIN
   Clear
END Pbox23C .
```

Figure 23.26
Continued.

An alternate design of the Pizza class

Behavior abstraction with polymorphic dispatch allows you to eliminate IF or CASE statements. The design of the Pizza class in the previous section requires an IF statement in the implementation of GetToppingString, because the topping string depends on whether the boolean field extraCheese is true or false. The design also requires an IF statement in the implementation of ToppingCost for the same reason. The cost of the topping depends on the value of the extraCheese attribute. The design in Figure 23.27 uses behavior abstraction to eliminate all IF statements in the methods for the Pizza class.

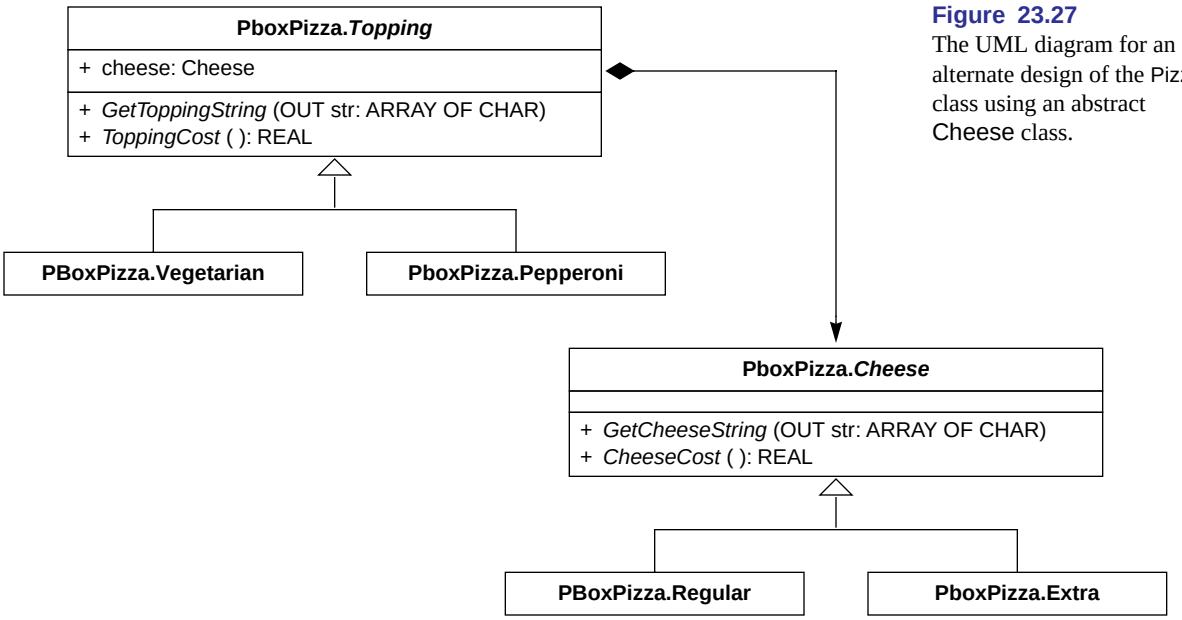


Figure 23.27
The UML diagram for an alternate design of the Pizza class using an abstract Cheese class.

In this design, extraCheese is not a boolean attribute of Topping. Instead, class Topping is composed of class Cheese, which is abstract. The Cheese class specifies methods GetCheeseString and CheeseCost, which are implemented by the concrete subclasses Regular and Extra.

The Regular version of GetCheeseString sets str to the empty string “”. The Extra version of GetCheeseString sets str to the string “, Extra cheese” with a leading comma and space. The implementation of the Vegetarian version of GetToppingString simply concatenates “Vegetarian” with the string for the cheese. If cheese is instantiated as Regular, then “Vegetarian” concatenated with the empty string is sim-

ply “Vegetarian”. If cheese is instantiated as Extra, then “Vegetarian” concatenated with “, Extra cheese” is “Vegetarian, Extra cheese”.

The same idea is used to eliminate the IF statements from the implementation of ToppingCost. The Regular version of CheeseCost returns 0.0, and the Extra version returns the price difference between regular and extra cheese. ToppingCost can simply add the price difference to the cost for the topping without extra cheese.

Class composition versus inheritance

When object-oriented (OO) design was first invented there was no history of design experience on which to draw to develop programs. In the early days, programmers concentrated on inheritance and the power of polymorphism. With the hindsight that comes with experience, many designs from that era are now known to be less than sound because the designers did not have an appreciation of the utility of class composition. Most of OO design consists of modeling the problem to be solved with an optimum mixture of class composition and inheritance. UML class diagrams are useful because they capture these two aspects of OO structure in a standard form that does not depend on the programming language used for the implementation.

As with any design process, there is always more than one way to solve a problem. The choice of a particular solution depends on the trade-offs that the designer makes according to the goals of the project. Figure 23.28 shows another way to model the relationship between a pizza and a crust. Suppose the application is for a bakery where the crust is the important object. The bakery may make crusts for pies as well as pizzas. You might choose to have classes Pie and Pizza inherit from Crust, reasoning that crust is common to both pies and pizzas and should therefore be abstracted out to the superclass.

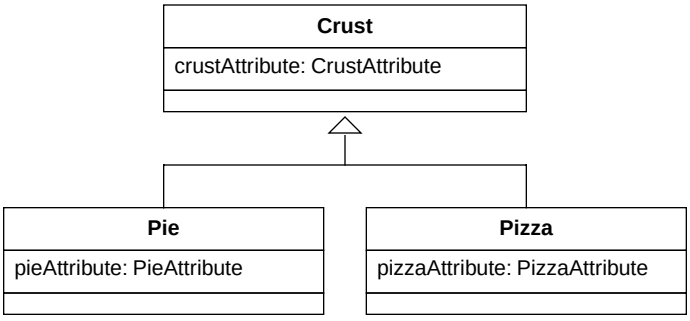


Figure 23.28
A possible design with a different relationship between Pizza and Crust.

Most OO designers would object (!) to this design. Remember that inheritance is the “is-a” relationship and class composition is the “has-a” relationship. The design of Figure 23.28 implies that a pizza *is* a crust, whereas the design of Figure 23.24 implies that a pizza *has* a crust. In this example, the real world nature of pizza provides a guide for the proper model to use. In some situations the problem is not so clear even after considering the so-called real world. For example, what is the relationship between a square and a rectangle? Mathematically, a square is a rectangle with equal sides. Would you therefore make a square a subclass of a rectangle? The problem with that implementation is that a square object would inherit both the

length and width of the rectangle when it only needs the length of one side. The square versus rectangle design problem has provoked much debate in OO circles. The upshot is that many solutions to any given problem are possible, and good OO design can be difficult.

Records versus pointers

Figure 23.19 shows that Shape, Rectangle, and Circle are all declared to be pointers to records.

```
Shape* = POINTER TO ABSTRACT RECORD END;
Rectangle* = POINTER TO RECORD (Shape)
   length, width: REAL
END;
Circle* = POINTER TO RECORD (Shape)
   diameter: REAL
END;
```

With these declarations, you can allocate local pointer variables on the run-time stack with

```
myShape: Shape;
myRectangle: Rectangle;
myCircle: Circle;
```

You can allocate rectangle and circle records from the heap with

```
NEW(myRectangle);
NEW(myCircle)
```

because myRectangle and myCircle are pointers. It is the records that are allocated from the heap, and the pointers on the stack that point to them. However, you cannot allocate a shape from the heap with

```
NEW(myShape)
```

because Shape is abstract. The usual class assignment rule applies. The assignment

```
myShape := myRectangle
```

is legal, but the assignment

```
myRectangle := myShape
```

is not.

There is nothing in Component Pascal to prevent you from declaring classes and subclasses to be records instead of pointers to records. For example, Component Pascal permits the following declarations, which differ from the previous declarations only by the omission of POINTER TO.

```
ShapeR* = ABSTRACT RECORD END;
RectangleR* = RECORD (Shape)
   length, width: REAL
END;
CircleR* = RECORD (Shape)
   diameter: REAL
END;
```

You can allocate local variables for the rectangle and circle records on the run-time stack, such as

```
yourRectangle: RectangleR;
yourCircle: CircleR
```

But, the declaration of a local variable of type Shape such as

```
yourShape: ShapeR;
```

is not allowed, because it attempts to allocate an abstract record. The class assignment rule

```
yourShape := yourRectangle
```

cannot apply here, because it is impossible to have yourShape in the first place.

Extensible records

Component Pascal provides the record attribute EXTENSIBLE to allow the programmer to declare a superclass that is not abstract as follows.

```
ShapeE* = EXTENSIBLE RECORD END;
RectangleE* = RECORD (Shape)
   length, width: REAL
END;
CircleE* = RECORD (Shape)
   diameter: REAL
END;
```

The local variable allocations on the run-time stack

```
herShape: ShapeE;
herRectangle: RectangleE;
herCircle: CircleE;
```

are all legal. The allocation for herShape is legal, because herShape is not abstract. Now that you have a superclass with subclasses you might think that the class assignment rule would permit the assignment

```
herShape := herRectangle
```

Class assignment rule

But, it does not! These variables are not assignment compatible, even though you can assign myRectangle to myShape in the pointer version.

Object-oriented programming languages in general and Component Pascal in

particular rely on the characteristics of pointers and allocation from the heap to provide polymorphism. Some pure OO languages do not have explicit pointers at all. In these languages, every variable is automatically a pointer to a record, even though the pointer is hidden. The only assignment that is possible is a pointer assignment. The phrase "pointer to" is usually not part of the terminology in these languages. Instead, a variable is said to be a "reference to" an object. But apart from the terminology, such languages are identical to Component Pascal in their OO capabilities and their underlying structure. Most pure OO languages provide automatic garbage collection because of the prevalence of heap allocation.

Other OO languages are similar to Component Pascal in that they are not pure OO. These languages provide procedure abstraction as well as class and behavior abstraction and usually have pointers as an explicit primitive type. An advantage of such mixed-paradigm languages is that you are not forced to use OO techniques when they are not appropriate. These languages also tend to be more efficient than pure OO languages. Component Pascal is rather unique in that pointers are an explicit primitive type, yet the language still provides automatic garbage collection.

Extensible records have a place in OO design. However, the most important OO design patterns are based on abstract records instead. An abstract record cannot be instantiated. Its purpose is to be a kind of blueprint for the subclasses that are extended from it. The design patterns presented in this book use abstract records for inheritance together with class composition.

Private versus public

Items that are not exported are called private, and items that are exported are called public. An important OO design issue is whether to export an attribute, making it public, and if so, whether it should be exported read/write or read-only.

Consider class Rectangle in Figure 23.21 where attributes length and width are private. Because they are private, they are not accessible to any client module, including module Pbox23C in Figure 23.26. But, procedure Pbox23C.Insert needs to give values to length and width from the input dialog box. It does so by executing the call

```
rectangle.SetState(MAX(0.0, d.length), MAX(0.0, d.width))
```

Method SetState is public, and so can be called from Pbox23C.Insert. An alternate design would be to make length and width public. Then, you would not even need the SetState method. To set the length and width of the rectangle, Pbox23C.Insert would simply make the assignments

```
rectangle.length := MAX(0.0, d.length);
rectangle.width := MAX(0.0, d.width)
```

Now, consider attribute shape in class Pizza in Figure 23.24. Because it is public, Pbox23C.Insert can access it directly with the assignment

```
pizza.shape := rectangle
```

An alternate design would be to make shape private and supply the public method

```
PROCEDURE (p: PboxPizza) SetShape* (s: PboxShape.Shape), NEW;
BEGIN
   p.shape := s
END SetShape;
```

Procedure Pbox23C.Insert would then make the call

```
pizza.SetShape(rectangle)
```

to set the shape attribute of pizza to rectangle.

What is the difference between these two design decisions? Why are Rectangle.length and Rectangle.width private, which requires a public method to change them, while Pizza.shape is public, which requires no such method? Why not make every attribute public and dispense with methods to change their values? After all, your program would be shorter and would also run faster because of the time it takes to call a method.

The programs in this book are small enough to be written by a single individual. You typically write both the client module, like that in Figure 23.26, and the server module, like that in Figure 23.25. It is common, however, in a large project for the programming effort to include a team of programmers, so that the person who writes the client module is not the person who writes the server module. Indeed, it is even possible for the server programmer to provide the module to many different customers who write their own clients. In such an environment, protection is the key concept. If you write a server and you do not know who will write the client you should program defensively, making sure that the data in your data structures are consistent and meaningful. You should not allow clients to corrupt your data structures.

Design by contract

BlackBox provides design by contract to ensure that clients cannot violate the preconditions of any methods. The Component Pascal ASSERT statement enforces the preconditions stated in the specification of each procedure. Rectangle.length and Rectangle.width are private to enforce the invariant that they cannot be negative. If one of these dimensions were set negative, then its value would be meaningless as would be the computations of the area and perimeter of the rectangle. The public method to set the state of the rectangle ensures with an ASSERT statement that the values will never be set negative. This implementation is consistent with the design-by-contract rule, which states

The design-by-contract rule

- IF in the client.
- ASSERT in the server.

There is no corresponding reason to protect Pizza.shape beyond the protection provided by the language itself. Component Pascal is a strongly typed language. The compiler will allow an assignment to pizza.shape only if the right side of the assignment has the same type or an extension of the same type as the left side. Method SetShape above adds no protection value to the server. Whatever damage a client could do with a direct assignment to the public attribute it could do with a call to the public method that changes the private attribute.

You should be aware that some OO designers adhere to a blanket rule that

attributes should always be private and only accessed through public methods. The philosophy in this book, however, is to not provide superfluous methods. If a method to access the state of a private variable does not add protection value to the server, then the method can be dispensed with and the attribute made public.

The read-only export feature of public attributes is unique to Component Pascal. For example, class PboxMappers.Scanner has the attribute

```
eot-: BOOLEAN
```

exported read-only. When you write a statement like

```
WHILE ~sc.eot DO
```

where sc is an instance of class Scanner, you are accessing the value of a public attribute. Because it is not exported read/write, however, Component Pascal does not allow you to change its value with an assignment like

```
sc := FALSE
```

Such an assignment would corrupt the scanner's data structure. Most OO languages do not provide public read-only attributes. They would maintain eot as a private attribute and provide the function

```
PROCEDURE (s: PboxMappers.Scanner) Eot* (): B OOLEAN, NEW;
BEGIN
   RETURN s.eot
END Eot;
```

You would then include the function call in the WHILE statement as

```
WHILE ~sc.Eot() DO
```

This design is less efficient than the one permitted by Component Pascal, because a function must be called with each execution of the loop. It is necessary, however, when the language does not provide read-only access.

Abstract objects and methods

A curious restriction on abstract objects and methods are the following two rules.

Restrictions on abstract objects and methods

- You cannot instantiate an abstract object with NEW.
- You cannot implement an abstract method with BEGIN..END.

If you cannot allocate a new abstract object from the heap, to what use could you ever put such an object? Similarly, if you cannot give any instructions to an abstract method, you certainly can never call it. So, why have an abstract method at all if it can never be called?

The answer is that abstract objects and methods are necessary blueprints for the implementation of behavior abstraction with polymorphism. Procedure SetDialog in

Figure 23.17 shows an example of how an abstract object and method can be useful. Formal parameter s is an abstract object. When the compiler translates the SetDialog procedure, it cannot determine the dynamic type of s. That is, the compiler only knows that the static type of s is an abstract Shape. The statement

```
NEW(s)
```

would be a compile error because of the restriction that you cannot instantiate an abstract object with NEW. During execution, however, formal parameter s might correspond to actual parameter rectangle as in the call to SetDialog from procedure Insert. In that situation, the dynamic type of s would be Rectangle. On the other hand, the dynamic type could just as easily be circle as in another call to SetDialog from the same procedure. The general object s can morph between these two specific classes during execution. When the compiler translates SetDialog it must take into account that s could be either. It is the specific objects that are instantiated with NEW in procedure Insert.

When the compiler translates

```
s.GetDimensionString(d.dimensionString)
```

in procedure SetDialog, it must translate the method call for the general case, because s is general. There are three headings for GetDimensionString in Figure 23.19. Only the concrete versions for a Rectangle and a Circle are implemented with BEGIN..END. The version of GetDimensionString for a Shape cannot be implemented. Its purpose is for the compiler to verify that any specific object that inherits the general method will have the same signature, that is, the same number and types of parameters. It also allows the compiler to verify that the signature of the above call to GetDimensionString from SetDialog matches the signature in the heading for the general case.

Exercises

1. **(a)** What is the fundamental class assignment rule? **(b)** What is the fundamental class assignment rule applied to parameters?

2. What is the most important benefit of object-oriented design?

3. **(a)** What object-oriented relationship is the "has-a" relationship? **(b)** What object-oriented relationship is the "is-a" relationship?

4. What is the Hollywood Principle? What does it have to do with BlackBox?

5. Draw the abbreviated version of the UML class diagram for the following classes.

```
TYPE
    Alpha* = RECORD
        rho: POINTER TO Beta
    END;

    Beta = ABSTRACT RECORD END;
    Gamma = RECORD (Beta) END;
    Delta = RECORD (Beta)
        value: T;
        omega: Alpha
    END;

PROCEDURE (IN b: Beta) Phi (n: INTEGER; OUT val: T), NEW, ABSTRACT;
PROCEDURE (IN a: Alpha) Phi* (n: INTEGER; OUT val: T), NEW;
PROCEDURE (IN g: Gamma) Phi (n: INTEGER; OUT val: T);
PROCEDURE (IN d: Delta) Phi (n: INTEGER; OUT val: T);
```

Problems

6. Modify PboxShapesObj in Figure 23.13 to include a right triangle shape containing two fields named base and height. Test your program by modifying the program in Figure 23.9.

7. Modify the program in Figure 23.17 to include a right triangle shape containing two fields named base and height. Do not modify PboxShapesAbs in Figure 23.19. Instead, implement the Triangle class in a new module without changing any code in module PboxShapeAbs.

8. Complete the PboxPizza implementation of Figure 23.25 according to the design of the UML class diagram of Figure 23.24. Test your implementation by completing the program of Figure 23.26.

9. Complete the PboxPizza implementation of Figure 23.25 according to the design of the UML class diagram of Figure 23.24 with the modification of Figure 23.27 where class Cheese is abstract. None of the methods of any of the classes are allowed to have IF or CASE statements or any local variables. Test your implementation by completing the program of Figure 23.26, which should be unchanged from that of Problem 8.