CHAPTER FOURTEEN

Object-oriented Programming

14.1 Introduction

In this chapter we will primarily be concerned with mixing together two ideas we've already presented: generic operations (from Chapter 9) and object-based abstractions (from Chapter 13). This combination, in which abstract data types with state can have diverse implementations that are operated on through a uniform interface, is the core of the cluster of ideas known as *object-oriented programming*.

Although there is general agreement on these two core concepts, and although there is considerable enthusiasm for object-oriented programming, there is no general consensus on the remainder of the ideas in the cluster—which are essential or inessential, central or peripheral. Nonetheless, we'll cover a few of the more common "extras." The most significant is that rather than clearly distinguishing abstract data types from their implementation, the two are fused together into a single notion of a *class* and organized into a single *class hierarchy* in which the hierarchichal class relationship can represent the sharing of interface between abstract types, the provision of an abstract type's interface by a concrete implementation, or even the sharing of common portions of concrete implementations.

Rather than go into further detail here about object-oriented programming, we'll do so in the next section in the context of a specific example application program. After that, we'll explore some extensions and variations in a separate section and then peek behind the scenes to see how the object-oriented programming system we presume in the earlier sections can be efficiently implemented. Finally, we'll give you a chance to apply the object-oriented programming techniques imaginatively by building your own world in an adventure game.

14.2 An Object-oriented Program

To illustrate object-oriented programming, we'll look at a program for an on-line clothing catalog company. Ideally, users (or customers) would browse through pictures of the various items of clothing, selecting those they wanted. To keep things simple, we'll initially only offer oxford-cloth shirts and chino pants. As the customers browse, they fill in an order form, which is essentially a list of items. They do this by adding items to the list and inputting the specifics of those items, such as size and color. Customers can also, at any time, decide to see what's on their list, get the total price of all their items, delete an item from the list, or revise the specifics of an item on their list. When they are finished, if our system were real, they would pay for their order and, in due time, the items would be sent to them.

To illustrate the ideas of object-oriented programming, our program will concentrate on the items of clothing and on the order form, or list. Because our emphasis is not on the user interface, we'll stick with a style of interaction that is simple to program but not likely to win many customers. Similarly, in place of payment and shipping, we'll simply provide a mechanism for exiting the program.

We can use the preceding description to get started on an *object-oriented design*. First, we can look through this description for important nouns; these serve as clues regarding what classes of objects we'll need. These nouns are items of clothing and a list of items. Thus, we will tentatively assume that we have a class called *item* and a class called *item-list*. We also mentioned two specific kinds of items of clothing, which suggests having *oxford-shirt* and *chinos* classes. There are some other nouns that are less clear-cut. For example, the description mentions "prices," both for individual items and for the total. Should there be a class for prices, or should they simply be numbers? There is no one right answer to this question. For now, we'll use numbers for the prices. However, if there are a variety of interesting things to do with prices, we might want to reconsider this decision.

The next thing we need to do is identify the operations that we'll need to perform on objects of each class. Again, we can get some guidance from our description of the program by looking carefully both at what is explicitly said (for example, that items can be deleted from the list) and what is implicit (for example, to calculate a total price, it must be possible to find the price of each item). This combination will give us an initial list of operations. As we look in more detail at how each operation can be implemented, and as we give more careful consideration to how the operations are used to produce the overall user-visible behavior of the program, we may come up with some additions to our "wish lists."

Let's start by considering the item-list class. What do we need to be able to do to an item-list? The following operations come from the program's description; in each case we've italicized one or more words to serve as the name of the operation:

- *Add* a specified item to the list
- *Display* the list for the user
- Find the *total price* of the items on the list
- *Delete* a specified item from the list
- Allow the user to *choose* an item from the list (to delete or revise)

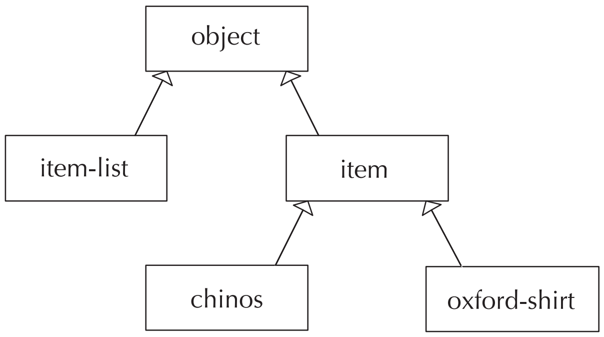
Once we started to program with these operations, we'd quickly discover we had a problem if the user decides to choose an item from an empty item list. The best way to handle this is to not even present choosing as an option when the list is empty; for the program to do that, it needs to be able to tell if the list is empty. Thus we are forced to add an *empty?* predicate to our catalog of operations.

The item class is somewhat simpler; the operations that appear (at least implicitly) in the program's description are as follows:

- Allow the user to *input specifics* (such as size and color) for an item
- *Display* the description and price of an item to the user
- Allow the user to *revise specifics* (such as size and color) for an item
- Find the *price* of the item

In the introduction to the chapter we indicated that one of the principal ideas in object-oriented programming is the use of generic operations to allow a uniform interface to diverse implementations of an abstract data type. How does this idea fit into shopping for clothing? The key observation is that although the two kinds of clothing in our catalog (oxford-cloth shirts and chinos) have some properties in common, there are considerable differences between them. For example, if you called a catalog company and asked for the price of any item, you would get an answer, but if you ordered a pair of chinos with a sleeve length of 32, the operator would be pretty puzzled. In our program, we'll use *subclasses* of the item class to represent different kinds of clothing. For example, the oxford-shirt class will be a subclass of the item class. Any oxford-cloth shirt is an item of clothing, or, in object-oriented jargon, an object that is an instance of the oxford-shirt class is also implicitly an instance of the item class. That way the oxford-shirt object will support all the operations of the item class, such as finding its price. However, because it is not simply an instance of the item class, but also of the oxford-shirt subclass, it can support additional operations, such as finding the sleeve length, as well.

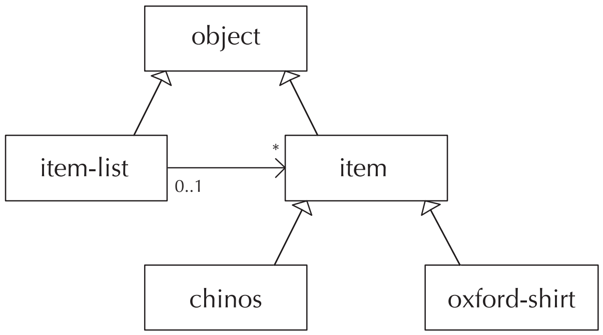
Now that we have a rough idea of what classes we'll need, let's take a look at some common object-oriented jargon. We start with the word *class*. When we speak generally of items of clothing, we're really talking about the abstract idea of such an item rather than a particular item or items. In object-oriented jargon, this idea makes "item" an example of a class. A class is an abstract grouping of objects that are similar to each other; the fundamental commonality between the objects of a class is that they all can be operated on in the same way. For example, we can find the price of any item in the same way and display the description of any item in the same way, although the results are likely to be different for different items. Just as a set can be a subset of another set, a class can be a *subclass* of a more general *superclass*, Thus, the oxford-shirt class is a subclass of the item class, and the item class is the superclass of the oxford-shirt class. We also say that the oxford-shirt class is *derived* from the *base class* item. The *ancestry* of a class consists of all the classes it is derived from, whether directly or indirectly. We choose to define a class's ancestry as including the class itself, as well as the class's superclass, the superclass's superclass, etc. In our example, the ancestry of the oxford-shirt class consists of the oxford-shirt class, the item class, and the object class. The object class is the ultimate ancestor of all classes in our system. This organization of our program's world into classes that are subclasses and superclasses is called the *class hierarchy*. We can represent this class hierarchy as a tree, with the object class at the root:



This diagram is our first example of a class diagram in a standard notation known as the *Unified Modeling Language*, or *UML*. This notation also provides means for expressing many other aspects of object-oriented design, not just the class hierarchy. We'll gradually explain more and more of the notation, as the need arises. (Even so, we'll only see the tip of the iceberg.)

The oxford-shirt class is really an abstract notion; when we talk about a particular oxford cloth shirt, this particular concrete shirt is a specific example of the general class. In object-oriented jargon, we say that a particular object is an *instance* of a class. So, for example, if the first item we wanted to order was a blue oxford-cloth shirt with a 32-inch sleeve length and a 16-inch neck size, this particular shirt would be an instance of the oxford-shirt class. One very important principle in object-oriented programming is that a particular object is an instance of all of the ancestor classes of the class it belongs to, which means that our blue oxford-cloth shirt is not only an instance of the oxford-shirt class but also an instance of the item class and an instance of the object class.

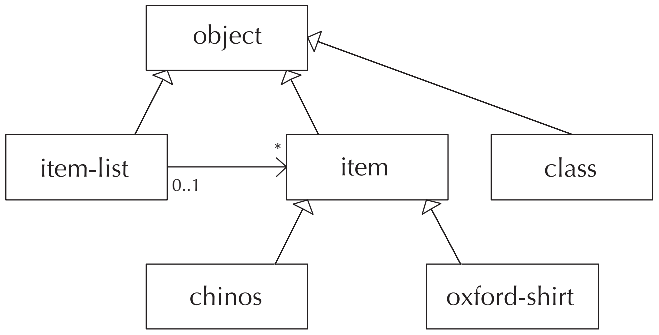
The UML class diagram can show information about the individual instances as well as about the overall classes. Our first diagram showed only the hierarchical relationship among the classes. However, an important relationship also exists between instances of the item-list class and instances of the item class. Each item-list is associated with arbitrarily many items. From the item-list, we can find the items. We can add this association to our diagram as follows:



The line between item-list and item represents the association. You can tell it is an association rather than a subclass/superclass relationship because it doesn't have the triangular arrowhead. Instead, it has an arrowhead formed out of two lines. This arrowhead points to the item class, which tells us that from the item-list, we can find the items. If each item also could tell us what item-list it belonged to, we'd have an arrowhead on the other end of the association as well. There is some additional information near the two ends of the association. An asterisk (*) is on the end pointing at the item class. This is a special UML notation indicating that each item-list can have any number of items. If every item-list had to have exactly five items, the number 5 would appear in place of the asterisk. If each item-list was constrained to have somewhere between three and seven items, the notation would be 3..7, and if the requirement was any number from 2 on up, the notation would be 2..*. These notations are called *multiplicities*. On the other end of the association, we see the multiplicity 0..1. As we just indicated, this notation is the UML way of indicating the range from 0 to 1. In other words, each item is associated either with no item-list or with one item-list.

This multiplicity of 0..1 documents a design decision we made. At the moment we only foresee having one item-list for the customer's choices. But what if we changed the program so that there could be multiple customers, each with an individual item-list? We'd still want each item to be on at most one list. Even though an item object is actually just a description of the clothing, and not the real cloth garment, sharing would be a bad idea because we earlier decided that the item class supported an operation for revising the specifics (such as size and color). If two customers had the same item on their item-lists, and one of them changed the size or color, the other customer would be surprised to see the same change. Therefore, we insist that each item appear on at most one item-list. Allowing an item to be on no list at all doesn't seem to do any harm and may come in handy as we explore the little "world" we are constructing, so we left this option open. Another designer might well have chosen to insist that each item be on an item-list, by using a multiplicity of 1 where we used 0..1. This illustrates one of the nice features of putting the associations on the UML diagrams, complete with their multiplicities: it forces us to think about these design decisions.

We've stressed that classes are abstract notions, whereas the objects that are instances of the classes are concrete. However, our object-oriented programming system is designed in such a way that each class has a concrete *class object* to represent it. For example, the oxford-shirt class has an object called `oxford-shirt-class` that represents it. These class objects are themselves instances of a class, namely, the class class. The class class is a class that is used just for the concrete representations of classes. For example, not only `oxford-shirt-class` but also `object-class`, `item-class`, `item-list-class`, `chinos-class`, and even `class-class` are all instances of the class class. If we add this class to our hierarchy, the full diagram looks as follows:



Each instance of a class contains certain pieces of information, stored in *instance variables*. Each instance of that class has the same assortment of instance variables: for example, every oxford shirt has a color, a neck size, a sleeve length, and a price. However, the specific values of those instance variables are stored independently in each instance—each shirt gets to have its own color, size, and price. Moreover, the instance variables are *state variables* that can change over time—if the customer remembers having put on a few pounds, the size of the chinos ordered can be changed.

Because an instance of one class is also an instance of all the ancestor classes, it will have the instance variables of each of these classes. For example, every item has a price, and so if the first thing on the item-list is an item, it has a price. The fact that it is not just a plain item, but rather an oxford-cloth shirt, is irrelevant. (However, oxford-cloth shirts can and do have some additional instance variables, beyond those that all items have.) The way we describe this situation is by saying that a class's instance variables are *inherited* by its subclasses. The oxford-shirt class gets its price instance variable by inheritance from the item class.

Objects do more than just store information, however. They can also perform operations. For example, an item can display its description and an item-list can add another item to itself. The operations are traditionally known as *methods*. We speak of general *method names*, such as `item/display`, which is the name used to make any item display itself, no matter how it chooses to do so. (We include the class name in the method name.) These method names are inherited by subclasses, so any item, even if it is an oxford-cloth shirt, can be operated on using `item/display`. However, the specific *method implementation* that gets used may or may not be inherited; in the case of oxford-cloth shirts and displaying, the implementation gets *overridden* with an oxford-shirt-specific way of displaying. If the method implementation is not overridden, the superclass's implementation is inherited. For example, oxford-cloth shirts report their price exactly the same way as plain items do because this method's implementation is inherited.

Now, having completed our jargon lesson, and having earlier sketched out the classes we need and their interfaces, we turn our attention to how the classes can be implemented. We'll use an object-oriented programming system that allows us to write object-oriented programs in Scheme. (A later section addresses how the object-oriented programming system itself is implemented.) This object-oriented programming system is available from the web site for this book.

The first thing we need to do is to implement the three classes we previously described. We start by defining the item-list class, using a procedure called `define-class` :

```
(define-class
  'item-list     ; the class is named item-list
  object-class   ; it has the object-class as its superclass
  '(item-vector  ; it has instance variables named item-vector
    num-items)   ;  and num-items
  '(add          ; and methods with these names
    display
    total-price
    delete
    choose
    empty?))
```

Most of the preceding arguments to `define-class` follow directly from our earlier discussion of the class and its interface. In particular, the method names come from our "wish list" of operations, and the superclass is `object-class` because we had no more specific superclass in mind. The only genuinely new revelation is the instance variables. As you might guess from their names, we are choosing to use a vector to hold the items, and to allow us to leave some extra unused space in the vector, we use another instance variable to explicitly record the number of items. That way we can make the vector big enough to accommodate some addition of items; if it still turns out to be too small, we can replace it with a larger one.

The `define-class` procedure is an abbreviation for a long list of other definitions. For example, by defining the item-list class as shown, we have defined `item-list-class` to be the actual class object (representing the class) but have also defined `make-item-list` to be the constructor, `item-list?` to be a predicate that tests whether an object is an item-list or not, etc. (We'll have more to say about constructors and predicates later.) Thus, you should not be surprised when we use names that you haven't seen explicitly defined. For example, we'll use `make-item-list` without ever saying `(define make-item-list ...)`. The definition was provided by `define-class`.

Next we need to implement the methods. For an example of how this implementation is done, let's first look at the simplest one, `empty?`, which only needs to check whether there are 0 items:

```
(class/set-method!
 item-list-class 'empty?
 (lambda (this)
   (= (item-list/get-num-items this) 0)))
```

Not only does this implementation of the `empty?` method illustrate how `class/set-method!` is used to provide method implementations, it also provides an example of how the value of an instance variable is retrieved. The `item-list/get-num-items` procedure can be applied to any instance of the item-list class to retrieve the current value of its `num-items` instance variable; we call it a *getter* procedure. The particular item-list it is applied to in the foregoing is the one passed in to the `empty?` method as that method's one argument, called `this`. Every method must have at least one argument, the object to operate on. No matter what other arguments a particular method might need, its first argument must be this object. (Here it was the only argument.) By convention this first argument is called `this` because it is "this object." (Another popular convention is to call it `self` because from the object's perspective it is "myself." We'll stick with `this`.) To summarize in the anthropomorphic language that is commonly used by object-oriented programmers, an item-list answers the question of whether it is empty by getting its own number of items and checking to see whether that equals 0.

The `num-items` instance variable will clearly need to be incremented in the `add` method and decremented in the `delete` method. A more subtle question is how it gets set to 0 in the first place. The most logical time to do this would be when the item-list is first created. Our object-oriented system lets us create an item-list as follows:

```
(define example-item-list (make-item-list))
```

What happens when we create this item-list is that a new instance of the item-list class is created and then a special object-class method, called `init`, is automatically invoked on this newly created object. The normal implementation of the `init` method, provided in the object class, does nothing interesting. However, it can be overridden like any other method. Thus we can arrange for `num-items` to start at zero by providing a suitable `init` method in the item-list class.

The item-list class's `init` method should set the `num-items` instance variable using that instance variable's *setter* procedure, which is called `item-list/set-num-items!`. We'll also have it set the `item-vector` to a 10-element vector initially because 10 seems like a large enough number that many customers will stay within it, but it is still a small enough number to not waste a great deal of the computer's memory. (Remember, if a particular customer wants to order more than 10 items, we can always switch to a bigger vector because `item-vector` is a state variable that can change.) Putting these two together, we get the following:

```
(class/set-method!
 item-list-class 'init
 (lambda (this)
   (item-list/set-item-vector! this (make-vector 10))
   (item-list/set-num-items! this 0)))
```

To see this `init` method in action, we can now do the above definition of `example-item-list`, which invokes `make-item-list` and gives a name to the result. Now we can check that it really behaved as expected:

```
(item-list/get-num-items example-item-list)
0
(item-list/empty? example-item-list)
#t
```

Another useful way to see that the item-list was constructed as expected is by using the `object/describe` method, which is provided by our object-oriented programming system for the sake of debugging and exploration:

```
(object/describe example-item-list)

An instance of the class item-list
with the following instance variable values:
   num-items: 0
   item-vector: [a 10 element vector]
   class: [an object of class class]
```

You can see from the preceding that the `num-items` and `item-vector` instance variables are as expected. You can also see that there is a third instance variable called `class`; every object has one of these. It records the most specific class this object is an instance of, which plays an important role in the underlying implementation of the object-oriented programming system. Until we examine that implementation in Section 14.4, you can safely ignore this instance variable.

Incidentally, we sometimes call procedures like `make-item-list` *instantiators* rather than constructors because the way they construct objects is by making instances of classes. From the object's viewpoint it is a constructor, whereas from the class's viewpoint, it is an instantiator. We say that `make-item-list` is `item-list-class`'s instantiator.

We now can continue implementing the methods of `item-list-class` one by one, starting with `add`, which adds an item to the list. It needs to check to see whether there is still room in the vector for the additional item. If not, the vector needs to be replaced with a bigger one, and then the addition can continue. Otherwise, the item can be inserted into the vector and the `num-items` counter increased by 1. For now let's leave out the details of how we upgrade to a bigger vector; a comment indicates where this upgrade will need to go:

```
(class/set-method!
 item-list-class 'add
 (lambda (this item)
   (let ((num-items (item-list/get-num-items this))
         (item-vector (item-list/get-item-vector this)))
      (if (= num-items (vector-length item-vector))
          (begin ; some code (yet to be determined) goes here to
                 ; replace the vector with a bigger one somehow
                 (item-list/add this item))
          (begin (vector-set! item-vector num-items item)
                 (item-list/set-num-items! this (+ num-items 1))
                 'added)))))
```

Exercise 14.1

Suppose that we forgot to replace the comment with the actual code to upgrade to a bigger vector and therefore used the method exactly as shown. Under what circumstances would users not notice our mistake? When they did notice our mistake, what would the symptoms be? Explain.

Rather than complicate the above method with the details of "growing" into a bigger vector, a better option is to use procedural abstraction to advantage: We concern ourselves here with *what* we want done (growing) and elsewhere with *how* that should be accomplished. That is, here we simply invoke a `grow` procedure and elsewhere define what that procedure does. In the object-oriented programming context, this `grow` procedure actually needs to be a method of the item-list class. Thus, in addition to all the methods from our original wish-list, which constitute the " public interface" of the class, we'll have an additional `grow` method that is for " private" internal use. The `define-class` needs to be revised to show the one additional method name; we'll also put some comments in to distinguish the two categories of methods:

```
(define-class
  'item-list
  object-class
  '(item-vector
  num-items)
 '(
   ;; intended for public consumption:
   add
   display
   total-price
   delete
   choose
   empty?
   ;; intended for private, internal use:
   grow
   ))
```

Having added this method name, we can now replace the comment in our `add` method with real Scheme code:

```
(class/set-method!
 item-list-class 'add
 (lambda (this item)
   (let ((num-items (item-list/get-num-items this))
         (item-vector (item-list/get-item-vector this)))
     (if (= num-items (vector-length item-vector))
         (begin (item-list/grow this)
                (item-list/add this item))
         (begin (vector-set! item-vector num-items item)
                (item-list/set-num-items! this (+ num-items 1))
                'added)))))
```

Now we no longer have the worry that we might forget to replace the comment. Of course, we could forget to implement the `grow` method; if that happens, we'll get an "unimplemented method" error message when growth first becomes necessary, which at least points us right to the problem.

The `grow` method itself can be written quite straightforwardly, particularly if we allow ourself the use of a `vector-copy!` procedure for copying the contents of the old vector into the new, larger vector.

Exercise 14.2

The `vector-copy!` procedure takes two vectors as its arguments. The second vector must be at least as long as the first. The `vector-copy!` procedure copies each element of the first vector into the corresponding position in the second vector. Write this procedure.

Exercise 14.3

Now write the `grow` method for `item-list-class`. It should make a new vector that is twice as long as the current `item-vector`. It should then copy the contents of the old vector into the new one. Finally, it should set the `item-vector` instance variable to be the new vector.

One method commonly invokes another method on the same object, as the `add` method did with the `grow` method. By the same token, a method will often also invoke some methods on other, related objects. For example, to display an item-list, each item in the list needs to be displayed:

```
(class/set-method!
 item-list-class 'display
 (lambda (this)
   (let ((num-items (item-list/get-num-items this))
         (item-vector (item-list/get-item-vector this)))
     ;;(continued)
```

```
      (from-to-do
       0 (- num-items 1)
       (lambda (index)
         (display (+ index 1))
         (display ") ")
         (item/display (vector-ref item-vector index))
         (newline))))
    (display "Total: ")
    (display-price (item-list/total-price this))
    (newline)
    'displayed))
```

As you can see, when `item-list/display` is applied to an item-list, it winds up applying `item/display` to each of the items stored in its vector. Of course, it does some other things as well, namely, decorating each item's display with an item number in front, such as `1)`, separating the items with newlines, and adding a display of the total price at the end. We assume that there is a procedure `display-price` that can take a numerical price and suitably display it; for example, in the United States there would be a dollar sign at the front and a decimal point before the last two digits.

Exercise 14.4

Write this (United States) `display-price` procedure. Assume that prices are represented within the program as an integer number of cents, such as 1950 for 19 dollars and 50 cents. You can use `quotient` and `remainder` to calculate the number of dollars and remaining cents. (Representing monetary amounts as an integer number of cents is a common practice in business programs because it can eliminate round-off errors. For example, adding together 100 one-cent items should yield an exact dollar, whereas adding 100 copies of .01 together on our computer yields 1.0000000000000007, due to the inexact way in which .01 is represented in the computer.)

Exercise 14.5

Write the `total-price` method for the `item-list` class; you can use `item/price` to get the price of each item.

To delete an item from an item list, we need to decrement the `num-items` instance variable. If the item wasn't the last one in the list, the items after it should be shifted one position closer to the beginning of the vector, to close up the gap.

Exercise 14.6

Write the `delete` method for the `item-list` class in this way. It should take the item to delete as an argument (after the `this` argument) and search the vector to find it, using `eq?` to compare the specified item with each one in the vector. It is an error if the item is not in the item-list.

There is one very subtle problem with a `delete` method that works as we just described. As the simplest illustration, consider starting with a new, empty item-list, inserting a single item into it, and then deleting it again. When the item is inserted into the item-list, position 0 in the vector will be set to the item and `num-items` will be increased to 1. When the item is deleted, `num-items` is decreased to 0 and the vector is left unchanged. Thus, position 0 of the vector still holds the item. From a logical standpoint, this doesn't matter because it will never be accessed; only the first `num-items` elements of the vector ever get looked at. However, the underlying Scheme system won't have any way of knowing this, so it will keep the item around just in case it is accessed, even though it never will be. If the item is not accessible in any way other than through the vector, its needless appearance in the vector will be all that prevents the Scheme system from reusing (*garbage collecting*) the portion of the computer's memory that the item occupies. Thus, although the program will work correctly, it may wind up using more memory than it needs to, or even unnecessarily running out of memory, because deleted items are still in the `item-vector`.

Exercise 14.7

Fix this problem by modifying your `delete` method so that after shifting the items down in the vector to fill the gap, it uses `vector-set!` to store the symbol `empty` (or any other arbitrary value) into the newly unused vector location. This vector location is the one that prior to the deletion held the last item.

At this point, the only method that remains to be implemented for the `item-list` class is the `choose` method, which will get us for the first time into the user interface of this program because we need to provide the user with some way of choosing an item from the list. As a rather crude approach to the user interface, we'll display the whole item-list (using `item-list/display`) and then ask the user to select an item. Because the items are numbered consecutively from 1 by the display method, we can input the user's choice as an integer in the range from 1 to the number of items. Notice that we've got a tight coupling between the `display` method and the `choose` method; this coupling may not be good for the long-term maintainability of our program, particularly if we don't explicitly document it in the requirements for the `display` method. At any rate, our `choose` method is as follows; we use an `input-integer-in-range` procedure, which we'll need to write later.

```
(class/set-method!
 item-list-class 'choose
 (lambda (this)
   (if (item-list/empty? this)
       (error "Can't choose an item when there aren't any.")
       (begin
         (display "Which item?")
         (newline)
         (item-list/display this)
         (vector-ref (item-list/get-item-vector this)
                     (- (input-integer-in-range
                         1 (item-list/get-num-items this))
                        1))))))
```

The `input-integer-in-range` procedure simply needs to display a prompt, read an input from the user, and if it isn't appropriate, complain and try again:

```
(define input-integer-in-range
  (lambda (min max)
    (display "(enter ")
    (display min)
    (display "-")
    (display max)
    (display ")")
    (newline)
    (let ((input (read)))
      (cond ((not (integer? input))
             (display "input must be an integer and wasn't")
             (newline)
             (input-integer-in-range min max))
            ((or (< input min)
                 (> input max))
             (display "input out of range")
             (newline)
             (input-integer-in-range min max))
            (else
             input)))))
```

At this point we're done implementing the item-list class (although we'll look at variations on it in the next section), and we can move on to our other principal class, item. The only piece of information we can be sure is associated with any item is its price—not all items have distinguishing colors or sizes, for example. Thus we'll put a single instance variable, `price`, in the item class, leaving the rest (color, size, ...) for more specialized subclasses. Recalling our earlier wish list of method names for the item class, we get the following definition:

```
(define-class
  'item
  object-class
  '(price)
  '(
    ;; intended for public consumption:
    price
    display
    input-specifics
    revise-specifics))
```

One decision we'll have to make is how an item's price gets set in the first place. A straightforward possibility is to have it supplied to the `make-item` constructor as an argument, as in

```
(define example-item (make-item 1950)) ; 1950 cents = $19.50
```

Our object-oriented programming system handles constructor arguments such as this one by passing them as additional arguments to the `init` method, after the `this` argument. Thus the item class needs an `init` method that accepts the price as its second argument:

```
(class/set-method!
 item-class 'init
 (lambda (this price)
   (item/set-price! this price)))
```

Most of the remaining methods for the item class are easy. The `price` method needs only to get the value of the price instance variable and return it, the `display` method needs only to display the price (because there is no other information), and the `input-specifics` method doesn't need to do anything at all because there are no "specifics" (such as color or size) to get.

Exercise 14.8

Write these three methods.

Exercise 14.9

Play around with the item and item-list classes. Make some items with various prices and add them to and delete them from an item-list. Display the items and the item-list, redisplaying the item-list after each addition or deletion. Also use `object/describe` to look at the items and at how the item-list evolves. Make sure the item-list successfully outgrows its original vector.

The only remaining method for the item class is `revise-specifics`. At first glance, it would appear that we should simply make this a do-nothing method, like `input-specifics`, because plain items have no specifics. However, there is another, better alternative. We can write a `revise-specifics` method that does nothing except invoke the `input-specifics` method. For a plain item, this will of course still do nothing. For an instance of some subclass of item, such as oxford-shirt, if the `revise-specifics` method isn't overriden, the one from the item class will be used, but it will wind up invoking the subclass's more specialized `input-specifics` method, and all the specifics of color, size, etc., will wind up being input. Finally, if some subclass wishes to override not only `input-specifics` but also `revise-specifics`, it can provide a more sophisticated means of revision that allows some specifics to be left unchanged and others revised, rather than requiring them to all be input over again.

Exercise 14.10

Write the `revise-specifics` method for the item class.

To demonstrate more concretely our point about `revise-specifics`, consider the following example:

```
(define-class
  'special-item
  item-class
  '()
  '())
```

```
(class/set-method!
 special-item-class 'input-specifics
 (lambda (this)
   (newline)
   (display " *** doing special input ***")
   (newline)))

(define a-normal-item (make-item 1950))

(define a-special-item (make-special-item 1000000))

(item/input-specifics a-normal-item)

(item/input-specifics a-special-item)

 *** doing special input ***

(item/revise-specifics a-normal-item)

(item/revise-specifics a-special-item)

 *** doing special input ***
```

As you can see, although we only provided a specialized version of the `input-specifics` method, both `item/input-specifics` and `item/ revise-specifics` now behave differently for the normal item than for the special item. (For the normal item, both are silent, whereas for the special item, both print out the message "doing special input.") This is an important consequence of using generic operations: When one operation invokes another, making a specialized version of the called one effectively specializes the behavior of the calling one as well.

Speaking of specialized subclasses of `item-class`, we should get around to creating one. Oxford-cloth shirts have colors, neck sizes, and sleeve lengths, in addition to the inherited instance variables (`price` from `item-class` and `class` from `object-class`). There is also one other, less obvious, instance variable that we need to add, one to indicate whether the specifics have been input yet or not. That way, if the item is displayed before the specifics have been input, a less detailed description can be produced. For this purpose we'll use a boolean-valued instance variable called `specified-yet`. There are no additional method names beyond those inherited from `item-class`, so the definition is as follows:

```
(define-class
  'oxford-shirt
  item-class
  '(color
    neck
```

```
     sleeve
     specified-yet)
   '(
     ))
```

We'll set up the `init` method so that we can use `make-oxford-shirt` as shown below:

```
(define an-oxford-shirt (make-oxford-shirt))

(item/price an-oxford-shirt)
1950

(oxford-shirt/get-specified-yet an-oxford-shirt)
#f
```

As you can see, we didn't need to specify any arguments to the `make-oxford-shirt` constructor, and we wound up with a price of 1950 and the `specified-yet` instance variable set to `#f`. The question is, how shall we arrange for this result? The most obvious possibility, but not the best, would simply be to have the `init` method set both the `price` instance variable and the `specified-yet` instance variable:

```
(class/set-method!
 oxford-shirt-class 'init
 (lambda (this)
   (item/set-price! this 1950)
   (oxford-shirt/set-specified-yet! this #f)))
```

The problem with this method implementation is that although it works, it contains a nearly verbatim copy of what `item-class`'s version of the `init` method does, which could result in a maintainability problem. Right now all `item-class`'s `init` does is to set the `price` instance variable, but what if we changed it to do more? Would we remember to also make the additions in `oxford-shirt-class`'s `init`? The program would be more maintainable if rather than repeating what `item-class`'s `init` does, we could simply invoke that `init` method and let it do whatever it needs to do. At first, you might think that the following will do that:

```
(class/set-method!
 oxford-shirt-class 'init
 (lambda (this)
   (item/init this 1950)
   (oxford-shirt/set-specified-yet! this #f)))
```

The problem is that when `item/init` is applied to an object, the class of the object determines which implementation of the `init` method is used, just like with any other method invocation. Thus, when it is applied to an instance of the oxford-shirt class, as here, we would wind up using the `init` method from `oxford-shirt-class`. As such, this `init` method is simply reinvoking itself, resulting in an infinite recursion. (Actually, it isn't even an error-free infinite recursion because the recursive call passes in two arguments, but this `init` method only accepts one.)

What we need is to say "please ignore the fact that this object is more than just a plain item; I still want you to invoke the plain item `init` method." In our object-oriented programming system, we can do this by using `item^init` instead of `item/init`:

```
(class/set-method!
 oxford-shirt-class 'init
 (lambda (this)
   (item^init this 1950)
   (oxford-shirt/set-specified-yet! this #f)))
```

This use of `^` rather than `/` is a general feature of our object-oriented programming system. It always means to use the method implementation corresponding to the class that explicitly appears before the `^`, rather than using the method implementation corresponding to the class of the object being operated on. For example, returning to the earlier example of the special-item class, if we were to say `(item^input-specifics a-special-item)`, we wouldn't get the special message we get from `(item/input-specifics a-special-item)` because we are asking to have the method invoked that would be used were the special item just a plain item.

The `^` feature is not nearly as commonly used as normal method invocation; the primary use for it is to allow a method (such as the `init` method above) to invoke the method of the same name from the superclass. In particular, normally every `init` method will invoke the superclass's `init` method in this way; therefore each ancestor class winds up with a chance to do its own initialization.

Exercise 14.11

The two `init` methods we previously showed for `item-list-class` and `item-class` violated this guideline. Neither of them gave the superclass's `init` a chance. Luckily, the superclass (`object-class`) doesn't currently do anything in its `init` method. However it would be wiser not to count on that. Revise these two `init` methods so that their first step invokes the superclass's `init` method.

This feature of having a method invoke the like-named method from the superclass provides an intermediate point between the extremes of either inheriting the superclass's method unchanged or totally overriding it with a new method implementation. Instead, an overriding implementation can be provided that *augments* the superclass's implementation by invoking it and also doing some extra work. For example, consider the `display` method. The implementation in `item-class` already does some useful work, namely, displaying the price of the item. If there were ever any additional information that all items had, presumably this `display` method would show it as well. So, when we write a `display` method for the oxford-shirt class, it should add on to that base behavior by taking care of displaying the information specific to that subclass and then using `item^display` as well:

```
(class/set-method!
 oxford-shirt-class 'display
 (lambda (this)
   (if (oxford-shirt/get-specified-yet this)
       (begin
         (display (oxford-shirt/get-color this))
         (display " Oxford-cloth shirt, size ")
         (display (oxford-shirt/get-neck this))
         (display "/")
         (display (oxford-shirt/get-sleeve this))
         (display "; "))
       (display "Oxford-cloth shirt; "))
   (item^display this)))
```

Exercise 14.12

Suppose you do the following:

```
(let ((item-list (make-item-list)))
  (item-list/add item-list (make-item 100))
  (item-list/add item-list (make-oxford-shirt))
  (item-list/add item-list (make-item 200))
  (item-list/add item-list (make-item 300))
  (item-list/add item-list (make-oxford-shirt))
  (item-list/display item-list))
```

What output do you get from this example? Explain why.

The only method we still need to implement for the oxford-shirt class is the `input-specifics` method. (We could also optionally add a `revise-specifics` method.) Our `input-specifics` follows. Notice that it not only sets the instance variables corresponding to the shirt's color and size but also sets the `specified-yet` instance variable to `#t`. To get input from the user, it makes use of the `input-integer-in-range` procedure we saw earlier, as well as an `input-selection` procedure that allows the user to select one of a collection of values, which in this case are colors.

```
(class/set-method!
 oxford-shirt-class 'input-specifics
 (lambda (this)
   (display "What color?")
   (newline)
   (oxford-shirt/set-color!
    this (input-selection '("Ecru" "Pink" "Blue" "Maize" "White")))
   (display "What neck size? ")
   (oxford-shirt/set-neck! this (input-integer-in-range 15 18))
   (display "What sleeve length? ")
   (oxford-shirt/set-sleeve! this (input-integer-in-range 32 37))
   (oxford-shirt/set-specified-yet! this #t)
   'inputted))
```

A simple approach to implementing `input-selection` is to print out each of the choices with a number in front of it and then use `input-integer-in-range` to allow the user to select one. The procedure can then return the element in the list of choices that corresponds to the integer the user entered:

```
(define input-selection
  (lambda (choices)
    (define display-loop
      (lambda (number choices)
        (if (null? choices)
            'done
            (begin
              (display " ")
              (display number)
              (display ") ")
              (display (car choices))
              (newline)
              (display-loop (+ number 1) (cdr choices))))))
    (display-loop 1 choices)
    (list-ref choices
              (- (input-integer-in-range 1 (length choices))
                 1))))
```

At this point we've seen enough of how the classes work that we can turn our attention to the overall program that uses these classes. We call this program `compu-duds`:

```
(define compu-duds
  (lambda ()
    (let ((item-list (make-item-list)))
      (define loop
        (lambda ()
          (newline)
          (display "What would you like to do?")
          (newline)
          (display " 1) Exit this program.")
          (newline)
          (display " 2) Add an item to your selections.")
          (newline)
          (display " 3) List the items you have selected.")
          (newline)
          (display
           " 4) See the total price of the items you selected.")
          (newline)
          (let ((option
                 (if (item-list/empty? item-list)
                     (input-integer-in-range 1 4)
                     (begin
                       (display
                        " 5) Delete one of your selections.")
                       (newline)
                       (display
                        " 6) Revise specifics of a selected item.")
                       (newline)
                       (input-integer-in-range 1 6)))))
            (newline)
            (cond ((= option 2)
                   (let ((item (input-item)))
                     (item-list/add item-list item)
                     (item/input-specifics item)))
                  ((= option 3)
                   (item-list/display item-list))
                  ((= option 4)
                   (display-price (item-list/total-price item-list))
                   (newline))
                  ;;(continued)
                  ((= option 5)
                   (item-list/delete item-list
                                     (item-list/choose item-list)))
                  ((= option 6)
                   (item/revise-specifics
                    (item-list/choose item-list))))
            (if (not (= option 1))
                (loop))))) ;end of the loop procedure
      (loop)))) ;this starts the loop
```

As you can see, nothing thus far has embodied any knowledge of what kinds of clothing our on-line clothing store has to offer. That knowledge is used by the `input-item` procedure, which allows the user to select an item to add. Here is a simple version:

```
(define input-item
  (lambda ()
    (display "What would you like?")
    (newline)
    (display " 1) Chinos")
    (newline)
    (display " 2) Oxford-cloth shirt")
    (newline)
    (if (= (input-integer-in-range 1 2) 1)
        (make-chinos)
        (make-oxford-shirt))))
```

Although our on-line clothing store offers two kinds of clothing, chinos and oxford-cloth shirts, we've only implemented a class for the shirts. We need to implement a class for chinos in a similar manner so that the rest of the program will work. The definitions of `chinos-class` and its `init`, `display`, and `input-specifics` methods follow, completing our `compu-duds` program. However, although the program will now be complete, the next section investigates extensions to and variations on this program.

```
(define-class
  'chinos
  item-class
  '(color
    size   ; waist, in inches
    inseam ; also in inches
    cuffed ; #t = cuffed, #f = hemmed
```

```
      specified-yet)
     '(
      ))

  (class/set-method!
   chinos-class 'init
   (lambda (this)
     (item^init this 3300) ; chinos are priced at $33.00
     (chinos/set-specified-yet! this #f)))

   (class/set-method!
    chinos-class 'display
    (lambda (this)
      (if (chinos/get-specified-yet this)
          (begin
            (display (chinos/get-color this))
            (display " chinos, size ")
            (display (chinos/get-size this))
            (display ", ")
            (display
             (if (chinos/get-cuffed this)
                 "cuffed"
                 "hemmed"))
            (display " to ")
            (display (chinos/get-inseam this))
            (display " inches; "))
          (display "Chinos; "))
      (item^display this)))

  (class/set-method!
   chinos-class 'input-specifics
   (lambda (this)
     (display "What color?")
     (newline)
     (chinos/set-color! this
                        (input-selection
                         '("Charcoal" "Khaki" "Blue")))
     (display "What waist size? ")
     (chinos/set-size! this (input-integer-in-range 30 44))
     (display "Hemmed or cuffed?")
     (newline)
     (display " 1) Hemmed")
     (newline)
     (display " 2) Cuffed")
     (newline)
     (chinos/set-cuffed! this (= (input-integer-in-range 1 2) 2))
     (display "What inseam length? ")
     (chinos/set-inseam! this (input-integer-in-range
                               29
                               (if (chinos/get-cuffed this)
                                   34
                                   36)))
     (chinos/set-specified-yet! this #t)
     'inputted))
```

Exercise 14.13

Explain the role served by each conditional expression (i.e., `if` or `cond`) appearing in `chinos-class`'s methods.

14.3 Extensions and Variations

In this section we'll look at some of the many possible extensions to the `compu-duds` program from the last section. Because many of these extensions and variations will involve extending the class hierarchy or modifying existing classes, we first look at a couple additional tools our object-oriented programming system provides for helping to keep track of these modifications.

The first of these tools is that at any point you can see the complete class hierarchy by using `show-class-hierarchy`, as follows:

```
(show-class-hierarchy)

object
   item-list
   item
      chinos
      oxford-shirt
   class
```

In this case, we see that the object class (which is the "root" of the class hierarchy) has three subclasses: item-list, item, and class. (We'll learn more about the class class, which is used for representing classes, in the next section.) The item class in turn has the chinos and oxford-shirt subclasses. By using `show-class-hierarchy` as you add various new classes or reorganize the existing classes, you'll have an easier time keeping track of what you've done so far. It is also a useful tool if you need to work with someone else's program, in that it provides an overview.

Another useful tool for keeping track of your work as you modify existing classes and create new ones is `object/describe`. We've already seen how `object/describe` can be used to examine the state of a normal object, showing each of its instance variables. However, `object/describe` is also useful when applied to the special class objects, such as `chinos-class`. This is because there is an overriding `describe` method in the class class, which results in a special form of description just for classes. Here is an example:

```
(object/describe chinos-class)
The class chinos has the following ancestry:
   object
   item
   chinos
and the following immediate subclasses:
and the following instance variables (including inherited ones):
   specified-yet (new)
   cuffed (new)
   inseam (new)
   size (new)
   color (new)
   price (from item)
   class (from object)
and the following method names (including inherited ones):
   revise-specifics (name from item, implementation from item)
   input-specifics (name from item, new implementation)
   display (name from item, new implementation)
   price (name from item, implementation from item)
   init (name from object, new implementation)
   describe (name from object, implementation from object)
```

As you can see, this output provides a good overview of the status of the class, which could help you keep track of your work as you modify and create classes.

Before we start in on the actual extensions and variations, one final comment is worth making; it is a warning regarding the ordering restrictions that our object-oriented programming system imposes. When a class is defined, the superclass must already have been defined. When a method is set, the class needs to already have been defined. When an object is created, the class it is an instance of must already exist. (However, the methods don't need to have been set yet.) If you redefine a class, you'll need to redefine all the classes descended from it, reset all the methods in those various redefined classes, and remake any instances of those classes. Otherwise, you wind up with the subclasses still being subclasses of the old superclass, the methods existing only in the old class, and the objects still being instances of the old class. If you just reevaluate a `class/set-method!`, on the other hand, no problems result; all instances of the class and any descendant classes that inherit the implementation will immediately get the new version.

Now let's explore some variations and extensions of the `compu-duds` program. The first variation doesn't use any of the fancier features of object-oriented programming but rather takes advantage of data abstraction (i.e., the separation between a class's interface and its implementation). Consider the `total-price` method within the item-list class. As it stands now, it loops through all the items, adding up their prices. If the total price is repeatedly queried, this adding up would be wastefully repeated. Instead, the item-list could keep track of the total price as it added and deleted items and simply return the current total when queried.

Exercise 14.14

Modify the definition of the item-list class and of its `add`, `delete`, and `total-price` methods in order to keep and use a running total price in this way. Without changing any other part of the `compu-duds` program, it should work just the same as before, except with regard to efficiency.

Next let's look at adding new subclasses of `item-class`. The existing two subclasses allow you to dress like two-thirds of this book's authors, who are frequently seen wearing oxford-cloth shirts and chinos. But perhaps, like most of our students, you have other tastes in clothing.

Exercise 14.15

Add one or more additional kinds of items to `compu-duds` so that it more closely reflects your own taste in clothing.

Exercise 14.16

If you added other kinds of pants than chinos, or other kinds of shirts than Oxford shirts, you may have discovered some commonality between your new classes and the existing classes. Reorganize the class hierarchy so there are subclasses of `item-class` for pants and shirts, with the chinos now being a subclass of `pants-class` and Oxford shirts positioned under shirts. Place your own kinds of pants and shirts under the more general classes as well, and figure out what commonality you can centralize.

Exercise 14.17

How about adding yet another level of hierarchy between `shirt-class` or `pants-class` and the specific kinds of pants and shirts? What classes of an intermediate degree of specialization might you include?

In addition to using subclassing to reflect real-world concerns, such as additional kinds of clothing, we can also use it purely for the program's internal purposes. For example, consider the sizing of the vector in the item-list class. Right now, it grows (by doubling in size) whenever addition of an item requires it to, but it never shrinks, no matter how many items are deleted. For many applications, this is a reasonable design decision. However, if you want to be particularly thrifty with the computer's memory, it might be desirable to have a kind of item-list that shrank as deletions occured. Thus we'll define a subclass of `item-list-class` for thrifty-item-lists that is indistinguishable in terms of what operations it provides and how they outwardly behave but that shrinks down to a smaller vector when appropriate.

As a brief side trip, not particularly related to object-oriented programming, consider what "when appropriate" should mean: When should the vector shrink? At first you might think that because we double the vector's size when it overflows, we should cut the vector's size in half when it becomes only half-full. The problem with this approach is that if the customer happens to repeatedly add, then delete, then add, then delete, etc., we might wind up moving to a new size of vector on every single operation. For example, if we start with a 10-element vector and the customer after adding 10 items adds an eleventh, we move to a 20-element vector. The customer then deletes an item, leaving the 20-element vector only half-full, so we move back to a 10-element vector. The eleventh item is added again, leading us back to a 20-element vector, etc.

A way around this oscillation between neighboring sizes is to demand that the vector get even emptier than half-empty before we shrink down to a vector half the size. For example, we can shrink to a vector half the size when a deletion results in the vector being one-third full or less. However, the vector should never shrink below its original size of 10.

Exercise 14.18

Make a thrifty-item-list subclass of `item-list-class` that behaves in this way. By simply changing the `compu-duds` procedure to use `make-thrifty-item-list` in place of `make-item-list`, you should be able to switch to this new kind of item list. In particular, the rest of the program, which thinks it is operating on an item-list, will still be right because a thrifty-item-list is an item-list. Be sure not only to check that the program still works but also to use `object/describe` to make sure the vector size is growing and shrinking appropriately.

As we've remarked before, one of the most central features of object-oriented programming is its use of generic operations to allow instances of various subclasses of a class to be uniformly operated upon, even if the resulting behavior varies. For example, this allows chinos to be displayed differently from Oxford shirts, even though `item/display` is done to both. As a consequence of this, we rarely need predicates to test whether an object is an instance of a particular class because we can generally get appropriate behavior for each class without needing conditionals that say "if this item is an Oxford shirt, display it this way, whereas if it is a pair of chinos, display it this other way." Nonetheless, there are occasionally circumstances for which class predicates are useful, and so our object-oriented programming system provides them. Each class defined using `define-class` automatically gets a predicate with a name formed by adding a question mark to the end of the class name, such as `chinos?`. The most common situation in which these predicates are useful is when sifting a particular kind of object back out of a mixed collection.

Exercise 14.19

Add to the `compu-duds` program a feature that lets the user change in a single step the waist size of all the items selected that are chinos (and hence have a waist size). If in Exercises 14.15 and 14.16 you added other kinds of pants and introduced a general pants class, your new feature should change the waist size of all pants, rather than merely of all chinos.

Another feature of object-oriented programming that we haven't explicitly illustrated before now is that a class can be useful even without implementations for all its methods, if we don't intend to ever instantiate the class but rather use it only as a framework for subclasses that are instantiated and that provide the missing method implementations. In this case the class is called an *abstract class*. The `compu-duds` program very nearly contained an example of an abstract class, in that the item class really has no reason ever to be instantiated to form "plain items," other than for the sake of some of our pedagogic examples; in the program itself, only specific subclasses of item ever get instantiated. So, for example, we could have left out `item-class`'s implementation of the `input-specifics` method. With that omission, the program would have been unaffected, but we would no longer have the option of "plain items." In other words, the item class would have become abstract.

The idea of an abstract class can be taken to the fullest extent, often termed a *pure abstract class*, if the class provides only the names (and presumably specifications) for methods, without providing any method implementations or instance variables. In this case the class is serving as a true abstract data type, that is, just an interface specification, with its subclasses providing the implementations. For example, there is only a relatively minor amount of implementation sharing between the different kinds of item; they share the `price` instance variable and method, and the `display` method for showing the price, which gets augmented in the more specific `display` methods. Because this implementation sharing is so minor, it might make more sense to eliminate it altogether, leaving the item class purely abstract. As a negative consequence, some repetition would occur among the various subclasses, but as a positive consequence, the subclasses would be free to implement the formerly shared methods in other, perhaps simpler, ways. For example, the `price` method could simply return a constant price rather than having to get the value of a `price` instance variable.

Exercise 14.20

Another common use for pure abstract classes is to make explicit an interface that can be implemented using two or more different data structures. For example, we implemented the abstract interface of the item-list class using a vector but could instead have used a true list representation, in the sense of `cons`, `car`, `cdr`, `null?`, and `set-cdr!`. Change the class item-list to be purely abstract, with two subclasses called item-list-as-vector and item-list-as-list that provide these two implementations. You will have to change `compu-duds` to use one of the `make-item-list-as-`... constructors in place of `make-item-list`.

Finally, we'll conclude this section on extensions and variations with two loose ends left from the preceding section.

Exercise 14.21

We mentioned that the `revise-specifics` method could be overridden to allow more selective revision, rather than requiring the user to input all the specifics over again. Illustrate this possibility.

Exercise 14.22

We mentioned the possibility of using a class for prices. Presumably it would be most convenient if the constructor would take the number of dollars and the number of cents as two arguments, and if we had methods for displaying the price and for adding another price to the price. Design a class with this interface. Consider two possible implementations: one that stores the dollars and cents separately and one that stores them combined into a total number of cents, such as we've used previously. Explain the trade-offs between these implementations, and implement whichever of them you decide is on the whole preferable, or both if you think they should coexist. Test your price class in isolation, and then change the `compu-duds` program to use it.

14.4 Implementing an Object-oriented Programming System

In this section, we'll show how the object-oriented programming system used in the preceding sections can be implemented. The object-oriented programming system is a relatively large and complex piece of software, so we'll explain it incrementally in bite-sized chunks, each occupying a subsection. First we'll provide an overview of the system, and then we'll show how instance variables are represented and their getters and setters implemented. Next we'll show how instantiators (constructors) are implemented, then methods, and then predicates. Once we have explained all these features of classes, we'll show how classes themselves are constructed. In the last two subsections we pull it all together. We make extensive use of vectors to provide a reasonably efficient implementation of object-oriented programming, which is similar to some implementations used for object-oriented languages such as Java. If you'd rather accept object-oriented programming as a "black box" technology instead of looking behind the scenes, this section can be skipped without any loss of continuity.

Throughout this explanation we'll use the object-oriented programming system as our technology for describing itself. We'll assume first that we have constructed, somehow, the object class and the class class and we'll use them to explain what `define-class` does. As we do this, we'll see that `define-class` invokes several methods from the class class and the object class; we'll show how each of these methods is implemented. There is an inherent circularity in what we're doing; after all, how can we make a class for representations of classes before we have such a class to represent itself with? In the penultimate subsection, we show how to use *bootstrapping* to get around this circularity, and in the last subsection, we pull the last (superficial) layer of mystery aside by explaining how `define-class` abbreviates many normal definitions.

Overview

Before we delve into the implementation of the object-oriented programming system, we should make clear what needs implementing. One of the primary interfaces we've been using in the preceding sections is `define-class`; however, `define-class` is really just a convenient abbreviation, not the fundamentals of what we need to implement. In particular, evaluating a definition like

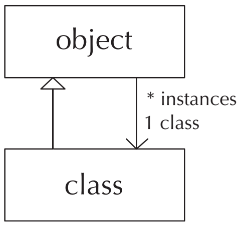
```
(define-class
  'widget
  object-class
  '(size)
  '(activate))
```

is really just a shorthand for evaluating the following sequence of definitions:

```
(define widget-class (make-class 'widget
                                 object-class
                                 '(size)
                                 '(activate)))

(define widget? (class/predicate widget-class))

(define make-widget (class/instantiator widget-class))

(define widget/get-size (class/getter widget-class 'size))

(define widget/set-size! (class/setter widget-class 'size))

(define widget/get-class (class/getter widget-class 'class))

(define widget/set-class! (class/setter widget-class 'class))

(define widget/activate (class/method widget-class 'activate))

(define widget^activate
  (class/non-overridable-method widget-class 'activate))

(define widget/init (class/method widget-class 'init))

(define widget^init
  (class/non-overridable-method widget-class 'init))

(define widget/describe (class/method widget-class 'describe))

(define widget^describe
  (class/non-overridable-method widget-class 'describe))
```

In the ensuing subsections we'll see how each of these pieces works; for example, in looking at instance variables, we'll see how `class/getter` produces the getter for `widget-class`'s `size` instance variable, which winds up being called `widget/get-size`. The machinery we will describe is housed in two fundamental classes: *object* and *class*.

As we've remarked before, the object class is the root of the class hierarchy and so is inherited from, directly or indirectly, by every class. It has one instance variable, `class`, that is used to indicate which class the object belongs to. We can indicate this fact, that each object knows which class it is an instance of, in a UML class diagram as follows:



Notice that there are two relationships between these fundamental classes. Each class object is an object, so we have a subclass/superclass relationship between the class class and the object class. However, each object is also associated with one particular class, the class it is a direct instance of, so we have an association arrow as well. The arrowhead is only on one end, indicating that the object knows its class, but the class doesn't know its instances. Next to the multiplicities, we've given a name to each role in this association to help clarify the meaning of the association. In particular, we show not just that each class object is associated with arbitrarily many objects but also what those objects are: the instances of the class. Similarly, the one class object associated with an arbitrary object is that object's class.

When we earlier expanded the `define-class` to show what it was a shorthand for, you could see that the widget class winds up with a getter and setter for the inherited instance variable named `class` as well as for its own `size` instance variable. This example illustrates a general pattern: Every class has getters and setters for all its instance variables, whether inherited or newly added, and similarly for methods. The object class also has two methods, which are inherited by all classes, namely `init` and `describe`. We've already seen the special role that the `init` method plays when a class is instantiated to construct a new object; we've also seen the use of the `describe` method for debugging and exploration. As you can see from the preceding definitions, the widget class winds up with these two methods and its own `activate` method. Each method also has a non-overridable version, which is given a name with a `^` in it, such as `widget^init`.

The class class is used for representations of classes. For example, `widget-class` is an instance of the class class. These objects of the class class store the information that is shared by the entire class, such as the names of the instance variables and methods. Each class object (i.e., each instance of the class class) is made using `make-class`. Once the class object is made, we use methods of the class class, such as `class/getter`, to obtain the procedures to use with that class.

Instance Variables

Each object is represented simply as a vector containing its instance variable values. To continue our running example, consider the widget class. Each widget has two instance variables: `class` and `size`. Therefore, each widget will be represented as a two-element vector. The vector will contain the widget's `class` instance variable at position 0 and its `size` at position 1. Given this representation, it is easy to see what getters and setters need to do. For example, if we wanted to produce `widget/get-size` and `widget/set-size!` by hand, the definitions would look like this:

```
(define widget/get-size
  (lambda (object)
    (vector-ref object 1)))

(define widget/set-size!
  (lambda (object value)
    (vector-set! object 1 value)))
```

One important aspect of object-oriented programming is that a class's operations (including getters and setters) can be used on any instance of that class *including* instances of subclasses, sub-subclasses, etc. To do this, we will impose a very simple constraint on the layout of the instance variables within the vector: The superclass's instance variables have to come first, in the same order as in the superclass. The newly added instance variables come afterward, at the end of the vector. That is why the `class` instance variable is at position 0 and the `size` instance variable at position 1: The `class` instance variable is inherited from the object class and hence had to come first.

As an example of where this approach would pay off, consider defining a subclass of the widget class, perhaps called colorful-widget. It could have additional instance variables, such as color, that would be stored in vector positions from 2 onward. However, just like any other widget, the size would be stored in position 1. Thus we can use the above `widget/get-size` on any widget, without needing to know whether or not it is a colorful-widget or any other more specialized variety.

Of course, we don't really want to write each getter and setter by hand; instead we want to use `class/setter` and `class/getter` to do it for us. That way, instead of defining `widget/get-size` using `vector-ref` and 1 as in the foregoing, we could just do

```
(define widget/get-size (class/getter widget-class 'size))
```

We have `class/getter` find the correct vector position and then use it to produce the specific getter procedure for us. We'll do that as follows:

```
(class/set-method!
 class-class 'getter
 (lambda (this instvar-name)
    (let ((index (class/ivar-position this instvar-name)))
      (lambda (object)
        (vector-ref object index)))))
```

Notice that we are using object-oriented programming to build the object-oriented programming system itself. In particular, `getter` is a method we are providing for the class class. It in turn uses another method from the class class, `class/ivar-position`, that is in charge of figuring out which vector position each instance variable goes in. For example, it is `class/ivar-position` that figures out that the `size` of a widget should go at position 1. We'll put off defining `class/ivar-position` until after we've seen how classes are created by `make-class`.

The preceding definition would work, so long as nobody ever applied a getter to an object that wasn't of the right class. However, if someone applied `widget/get-size` to an object that wasn't a widget, that person would get whatever instance variable happened to be stored in position 1—perhaps the account number of a bank account or the name associated with an employee record. To catch similar mistakes, we'd prefer that `widget/get-size` were actually defined more like the following:

```
(define widget/get-size
  (lambda (object)
    (if (widget? object)
        (vector-ref object 1)
        (error "Getter applied to object not of correct class:"
               'size 'widget))))
```

To have `class/getter` make such an improved getter for us, it is actually defined as follows:

```
(class/set-method!
 class-class 'getter
 (lambda (this instvar-name)
   (let ((index (class/ivar-position this instvar-name))
         (ok? (class/predicate this)))
     (lambda (object)
       (if (ok? object)
           (vector-ref object index)
           (error
            "Getter applied to object not of correct class:"
            instvar-name (class/get-name this)))))))
```

The definition of `class/setter` is completely analogous, just using `vector-set!` instead of `vector-ref`:

```
(class/set-method!
 class-class 'setter
 (lambda (this instvar-name)
   (let ((index (class/ivar-position this instvar-name))
         (ok? (class/predicate this)))
     (lambda (object value)
       (if (ok? object)
           (begin
             (vector-set! object index value)
             'set-done)
           (error
            "Setter applied to object not of correct class:"
            instvar-name (class/get-name this)))))))
```

To summarize the most important points from this subsection:

1. Each object is represented as a vector storing the instance variable values.
2. The superclass's instance variables come first, in the same order as in the superclass.
3. Aside from error-checking, all a getter or setter does is a single `vector-ref` or `vector-set!`.

Instantiators

As we saw in the preceding subsection, each object is represented as a vector. Thus to a first approximation, we could write the widget class's instantiator as follows:

```
(define make-widget
  (lambda ()
    (make-vector 2)))
```

However, two details are omitted here. First, if the new object is to truly be a widget, its `class` instance variable needs to be set to the `widget-class`. Second, we need to give the `init` method an opportunity to initialize the newly created widget. Thus, a better second attempt at `make-widget` would be as follows:

```
(define make-widget
  (lambda ()
    (let ((instance (make-vector 2)))
      (object/set-class! instance widget-class)
      (object/init instance)
      instance)))
```

At this point, had we stuck with the non-error-checking design for setters in which they only did a `vector-set!`, the `make-widget` procedure would work. But with error-checking setters, `object/set-class!` will first use `object?` to verify that it is really being passed an object (i.e., an instance of the object class). Unfortunately, until the class is set, there is no way this test can succeed. So, our next version of `make-widget` is as follows:

```
(define make-widget
  (lambda ()
    (let ((instance (make-vector 2)))
      (unchecked-object/set-class! instance widget-class)
      (object/init instance)
      instance)))
```

The `unchecked-object/set-class!` procedure that this version uses is like our first, crude `widget/set-size!` procedure. It can be written by hand and simply sets vector location 0.

One final issue is worth addressing before we turn to automating the production of class instantiators. Thus far we have assumed that `make-widget` takes no arguments. But what if, in reality, we want to use it like `(make-widget 1000)`? In this case, the `make-widget` procedure would have to accept the argument and pass it in to `object/init` (after the instance) because, as we've seen in our prior object-oriented programming, the `init` method actually processes any arguments to the instantiator. Rather that adding a single argument in this way, we'll allow any number of arguments to be passed to the instantiator and from there to the `init` method, using the special unparenthesized lambda-parameter notation (Section 10.3) and `apply`:

```
(define make-widget
  (lambda init-args
    (let ((instance (make-vector 2)))
      (unchecked-object/set-class! instance widget-class)
      (apply object/init (cons instance init-args))
      instance)))
```

Now all that remains is to automate the production of such instantiator procedures using `class/instantiator`. We'll assume that the class can find out how many instance variables it has using `class/get-num-ivars`. With this assumption, we can write

```
(class/set-method!
 class-class 'instantiator
 (lambda (this)
   (let ((num-ivars (class/get-num-ivars this)))
     (lambda init-args
       (let ((instance (make-vector num-ivars)))
         (unchecked-object/set-class! instance this)
         (apply object/init (cons instance init-args))
         instance)))))
```

At this point, we've clarified how two features of classes work—instance variables, with their getters and setters, and instantiators, for creating objects that are instances of a class. Of course, we've added some additional items to our agenda—`class/ivar-position` and `class/get-num-ivars` now need explaining too. For now, we'll put these off and continue working through the features we originally identified—methods, predicates, and class creation. Along the way, we're sure to discover yet more supporting machinery we need, and before we can declare ourselves finished, we'll need to have implemented not only the "official interface" methods, like `class/instantiator`, but also the "behind the scenes" methods, like `class/ivar-position`. This process of working from the external interface of a class inward to the supporting mechanisms is a common occurrence in object-oriented programming, no less so when the class we are programming is the class of classes.

Methods

Although at this point we know how to make instances of a class and how to get and set the instance variables contained within those instances, we're still missing one of the most fundamental ingredients of object-oriented programming: methods.

For efficient access, methods can be stored in vectors, just like instance variables are; the primary difference is that one vector of methods is shared by an entire class, whereas the instance variables are unique to each instance. Given a class, we can get its method vector using `class/get-method-vector`. Then we simply need to retrieve the particular method from the vector using `vector-ref` and apply it. Putting it all together, here is an example of how `widget/activate` might be written by hand:

```
(define widget/activate
  (lambda (object)
    (let ((method (vector-ref (class/get-method-vector
                                (object/get-class object))
                               2)))
      (method object))))
```

Notice that we get the method from the method vector of the object's class. That way, if the object wasn't a plain widget, but rather some special kind of widget such as our hypothetical colorful-widget, we'd get the colorful-widget class's `activate` method, which might do something special. This approach contrasts with `widget^activate`, which always gets the method from the `widget-class`'s method vector, even if the particular object turns out to be a colorful-widget:

```
(define widget^activate
  (let ((method-vector (class/get-method-vector widget-class)))
    (lambda (object)
      (let ((method (vector-ref method-vector 2)))
        (method object)))))
```

We wait until `widget^activate` is used to retrieve the method from the method vector so that the method can be changed using `class/set-method!`, which we'll see shortly.

We are making an assumption that underlies our ability to use `widget/activate` on any kind of widget, even special widgets like colorful widgets. Namely, we rely on all subclasses of `widget-class` to still use position 2 of the method vectors to store their `activate` methods. In other words, just as with instance variables, the methods whose names are inherited from the superclass need to be at the beginning of the method vector, in the same order as in the superclass; methods whose names are newly introduced come afterward. This explains how we knew that the `activate` method would be in position 2 of the method vectors: positions 0 and 1 would be occupied by the `init` and `describe` methods because those method names are inherited from `widget-class`'s superclass, `object-class`.

At this point, we've shown how methods can be written by hand. We still need to show how that process can be automated, and we still need to make a few refinements. Also, we'll need to show how methods are installed into the method vectors using `class/set-method!`. Before proceeding with these topics, however, we should point out that procedures such as the one named `widget/activate` aren't the *real* methods but act as though they were. When you apply `widget/activate` to a widget, it retrieves the real method from the method vector and applies the real method to the widget. Thus, the net effect is just as if the real method had been applied in the first place. For this reason we call `widget/activate` a *virtual method*.

Exercise 14.23

How many `vector-ref`s are done between the time when the virtual method `widget/activate` is invoked and the time when the real method gets applied? Explain. Be sure to count those done by procedures that `widget/activate` uses. You may omit any that are used only for error-checking, however.

Our first refinement will be to allow the `activate` method to take any number of additional arguments, rather than just the widget being activated. In the following procedure, `args` is a name for a list of all the additional arguments after the widget being activated:

```
(define widget/activate
  (lambda (object . args)
    (let ((method (vector-ref (class/get-method-vector
                               (object/get-class object))
                              2)))
      (apply method (cons object args)))))
```

Next we'll get rid of the explicit number 2 and instead use `class/method-position` to find it for us:

```
(define widget/activate
  (let ((index (class/method-position widget-class 'activate)))
    (lambda (object . args)
      (let ((method (vector-ref (class/get-method-vector
                                 (object/get-class object))
                                index)))
        (apply method (cons object args))))))
```

Now it is time for you to do some of the work.

Exercise 14.24

As a third refinement to `widget/activate`, make it check that it really is being applied to a widget and signal an error if not, much as with `widget/get-size`.

Exercise 14.25

Rewrite `widget^activate` so it too benefits from these three refinements. That is, it should accept additional arguments, use `class/method-position` rather than the number 2, and verify that it is being applied to a widget.

Exercise 14.26

Automate the production of virtual methods like `widget/activate` and `widget^activate` by writing the `class/method` and `class/non-overridable-method` methods as procedure factories. For an example, you could look at `class/getter`.

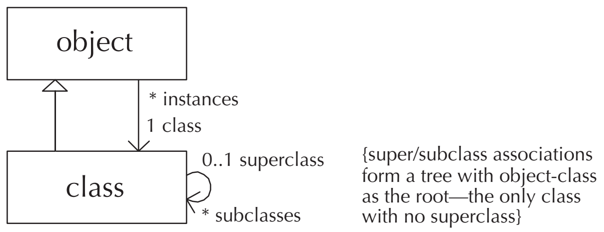
At this point we can retrieve and use methods, but we haven't seen how they get into method vectors in the first place. If you've been following carefully, you might well think it could be done as follows:

```
(class/set-method!
 class-class
 'set-method!
 (lambda (this method-name method)
   (vector-set! (class/get-method-vector this)
                (class/method-position this method-name)
                method)))
```

This code is very nearly right. Before making the few refinements it needs, we should point out that it brings us face-to-face with the bootstrapping problem that we'll have to solve eventually. We used `class/set-method!` to install into the `class-class` its `set-method!` method—in other words, we used `class/set-method!` to install itself. (This circularity is like trying to pull yourself up by your bootstraps, hence the description of the problem as being a bootstrapping problem.) Clearly this circularity will need to be addressed.

However, for now we'll leave the circularity unaddressed and instead clean up a detail. Namely, if we're putting the activate method into the widget class's method vector, and the widget class has subclasses like colorful-widget, and those subclasses don't have their own overriding methods, the same method that is being installed into the widget class's method vector should be installed into the subclasses' method vectors as well. That's how method implementations (as opposed to just method names) get inherited. In fact, the same method might get installed into the method vectors of sub-subclasses, sub-sub-subclasses, etc. Any class descended from `widget-class` will get this method, unless an overriding method intervenes.

Installing method implementation in descendant classes requires several forms of support. Most fundamentally, each class will need to know its subclasses. We'll assume that we can get a list of the subclasses of a class using `class/get-subclasses`. When we look at how classes are created by `make-class`, we'll see how the list of subclasses is kept up to date. For the time being, we'll record the availability of this information in our UML class diagram:


object
* instances
1 class
class
0..1 superclass
* subclasses
{super/subclass associations form a tree with object-class as the root—the only class with no superclass}

This class diagram illustrates another feature of the UML notation, namely, the ability to express a *constraint* that an association must satisfy. The constraint is written next to the association in curly braces. Here we've recorded the fact that the subclass relationship forms a tree. Starting from `object-class` and going to its subclasses, then their subclasses, etc., we should reach every class eventually, without ever reaching the same class a second time.

Another form of support we'll need is some way to keep track of whether a descendant class has its own overriding implementation of the method and hence shouldn't inherit the superclass's. Therefore, we'll give each class a second vector. In addition to the `method-vector`, there will be a `method-set?-vector` containing boolean values: True means the method has been directly set and hence shouldn't be inherited in, whereas false means that the superclass's implementation is the only one available. Using these, and a higher-order procedure for traversing the tree of descendant classes, we arrive at the following:

```
(class/set-method!
 class-class
 'set-method!
 (lambda (this method-name method)
   (let ((index (class/method-position this method-name)))
     (vector-set! (class/get-method-vector this)
                  index
                  method)
     (vector-set! (class/get-method-set?-vector this)
                  index
                  #t)
     (apply-below this
                  (lambda (class)
                    (vector-set! (class/get-method-vector class)
                                 index
                                 method))
                  (lambda (class)
                    (not (vector-ref (class/get-method-set?-vector
                                      class)
                                     index)))))
   method-name))

(define apply-below
  (lambda (class proc apply-to?)
    (for-each (lambda (subclass)
                (if (apply-to? subclass)
                    (begin (proc subclass)
                           (apply-below subclass proc apply-to?))))
              (class/get-subclasses class))))
```

Predicates

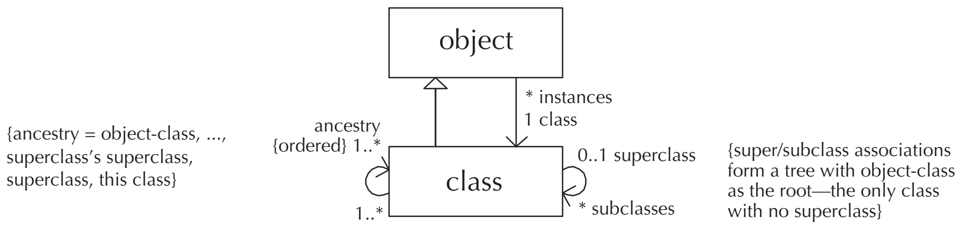
We need to design our class predicates so that, for example, `widget?` returns `#t` not only when applied to a plain widget that is directly an instance of `widget-class` but also when applied to a colorful widget that is an instance of some descendant class of `widget-class`. Thus it will not suffice to simply check to see if the object's class is equal to `widget-class`.

One straightforward solution would be to get the object's class, check for equality with `widget-class`, and if unequal retrieve the class's superclass and check again, iterating up the class hierarchy until an ancestor equal to `widget-class` is found or the top of the class hierarchy is reached. This approach would work, but it would be our first operation in the whole object-oriented programming system that wasn't accomplished with some small fixed number of vector accesses. Instead, the deeper the inheritance hierarchy, the slower the predicates would be. This is intolerably inefficient if we are going to use the predicates on a precautionary basis before each method invocation or instance variable access. Even if predicates were used more sparingly, we should design them to take a constant number of steps, absent any reason to the contrary.

For this design, we'll focus on each class's ancestry (i.e., the sequence of classes starting with `object-class` and working down the hierarchy to the class in question). We'll call `object-class` the "level 0 ancestor," the subclass of `object-class` that is an ancestor of the class in question its "level 1 ancestor," etc. In this terminology, the `widget?` predicate should return `#t` for any object whose class has `widget-class` as its level 1 ancestor. To test this condition efficiently, we simply store the ancestry as a vector, use `vector-ref` to retrieve element 1, and test to see whether it is equal to `widget-class`:

```
(define widget?
  (lambda (object)
    (let ((ancestry (class/get-ancestry (object/get-class object))))
      (and (> (vector-length ancestry) 1)
           (eq? (vector-ref ancestry 1)
                widget-class)))))
```

Of course, we still need to ensure that the ancestry vectors really exist in each class—we'll address that when we turn to the construction of classes. For now, let's update our UML class diagram, as a reminder of this important design decision:


object
* instances
1 class
class
ancestry
{ordered} 1..*
1..*
0..1 superclass
* subclasses
{ancestry = object-class, ..., superclass's superclass, superclass, this class}
{super/subclass associations form a tree with object-class as the root—the only class with no superclass}

The diagram shows a new notation, the special constraint {ordered}. This notation means that rather than each class being associated with an unordered set of one or more ancestors, it has an ordered list of one or more ancestors—one specific ancestor comes first, etc. We've also used the constraint notation to indicate what the ancestors need to be. The first ancestor (level 0) needs to be `object-class`, and the last ancestor needs to be the class whose ancestry this is. Immediately preceding that comes the superclass, and before that comes the superclass's superclass. The implication is that the ancestry needs to be consistent with the superclass/subclass association between classes.

The design of the `widget?` predicate just given has two flaws. First, it invokes `object/get-class` and `class/get-ancestry`, which as precautions invoke `object?` and `class?`, respectively. However, assuming that `object?` and `class?` follow the same design as `widget?`, they will in turn invoke `object/get-class` and `class/get-ancestry`. So, we'll wind up with an infinite recursion. The second flaw in `widget?`'s design is that no provision is made for the possibility that it might be applied to a value that is not an object at all, as in `(widget? 3)`.

Exercise 14.27

Write out the `object?` predicate that corresponds to the above `widget?` predicate. Now trace out the details of the infinite recursion that occurs when it is used.

To provide some measure of foolproofing against predicates being applied to nonobjects, we can test a number of additional consistency conditions. If all of these consistency conditions hold, we can be relatively certain that we have a genuine object and can test the original two conditions: that the ancestry vector is long enough and that it has the `widget-class` as its element at position 1 (i.e., as the level 1 ancestor). The following version of `widget?` embodies the additional consistency checks to guard against nonobjects, although it still has the infinite recursion problem:

```
(define widget?
  (let ((level (- (vector-length (class/get-ancestry widget-class))
                  1))
        (min-length (class/get-num-ivars widget-class))
        (min-class-length (class/get-num-ivars class-class)))
    (lambda (object)
      (and (vector? object)
           (>= (vector-length object) min-length)
           (let ((class (object/get-class object)))
             (and (vector? class)
                  (>= (vector-length class) min-class-length)
                  (let ((a (class/get-ancestry class))
                        (size (class/get-num-ivars class)))
                    (and (number? size)
                         (= size (vector-length object))
                         (vector? a)
                         (eq? (vector-ref a
                                          (- (vector-length a) 1))
                              class)
                         (> (vector-length a) level)
                         (eq? (vector-ref a level)
                              widget-class)))))))))
```

Exercise 14.28

Write out an English description of what each of the consistency checks is.

Exercise 14.29

Recall that the infinite recursion is due to the predicate using getters that themselves use predicates to do precautionary checking. The last time we had a problem caused by precautionary checking in the normal getter or setter procedures was in `make-widget` in the Instantiators subsection. Solve the infinite recursion problem using the same approach as we used there.

Exercise 14.30

Using your `widget?` predicate as a starting point, write the `class/predicate` method, which acts as a procedure factory for generating such predicate procedures.

Making Classes

Classes are themselves instances of the class class, made using `make-class`. To see what needs to be done to make a class, let's review the instance variables that we have come to assume each class has: `name`, `subclasses`, `num-ivars`, `method-vector`, `method-set?-vector`, and `ancestry`. In addition, we've assumed the availability of the `ivar-position` and `method-position` methods.

To support `ivar-position` and `method-position`, we'll equip each class with a table showing which vector position each instance variable is stored in and another table showing which vector position each method is stored in. We'll store each table as simply a list of lists; for example, the list `((class 0) (size 1))` would be used to show that `class` is in position 0 and `size` in position 1. Because these tables *associate* a position with each name, they are conventionally called *association lists* or simply *alists*. We'll add two more instance variables to our above list of instance variables the class class needs: `ivar-alist` and `method-alist`. These will hold the two association lists. For convenience we'll also add a `num-methods` instance variable, paralleling `num-ivars`.

At this point, we can show a full definition for `class-class`:

```
(define class-class
  (make-class 'class           ; name
              object-class     ; superclass
              '(name           ; instance variables
                subclasses
                num-ivars
                ivar-alist
                num-methods
                method-alist
                method-vector
                method-set?-vector
                ancestry)
              '(instantiator ; methods
                predicate
                getter
                setter
                method
                non-overridable-method
                set-method!
                ivar-position
                method-position)))
```

Of course, this again raises the bootstrapping issue of circularity: We just used `make-class`, which makes an instance of the class class, to specify what the class

class is. We'll resolve this problem in the next subsection, which is devoted to bootstrapping. For now, we at least have a human-readable description of the class class.

The `make-class` procedure is simply the instantiator for the class class, obtained using `class/instantiator`. As such, all it does is create an appropriately sized vector, install `class-class` into element 0 of that vector, and then invoke the `init` method to do all the real work. We can already sketch out most of what the `init` method will need to do, simply by initializing each of the instance variables:

```
(class/set-method!
 class-class 'init
 (lambda (this class-name superclass instvar-names method-names)
   (object^init this)
   ;; some code should go here to check that none of the new
   ;; instvar-names or method-names are already in use in the
   ;; superclass -- and if any are, to signal an error
   (class/set-name! this class-name)
   (class/set-subclasses! this '())
   (class/set-subclasses! superclass
                          (cons this
                                (class/get-subclasses superclass)))
   (class/set-num-ivars! this (+ (class/get-num-ivars superclass)
                                 (length instvar-names)))
   (class/set-ivar-alist! this
                          ;; some code needs to go here to
                          ;; assign the positions for the instance
                          ;; variables
                          )
   (class/set-method-alist! this
                            ;; some code needs to go here to
                            ;; assign the positions for the methods
                            )
   (let ((num-methods (+ (class/get-num-methods superclass)
                         (length method-names))))
     (class/set-num-methods! this num-methods)
     (let ((method-vector (make-vector num-methods)))
       (class/set-method-vector! this method-vector)
       (vector-copy! (class/get-method-vector superclass)
                     method-vector)
       ;;(continued)
```

```
        (for-each (lambda (method-name)
                    (vector-set! method-vector
                                 (class/method-position this
                                                        method-name)
                                 (lambda (object . args)
                                   (error "Unimplemented method"
                                          method-name))))
                  method-names))
      (let ((method-set?-vector (make-vector num-methods)))
        (class/set-method-set?-vector! this method-set?-vector)
        (vector-fill! method-set?-vector #f)))
    (let ((ancestry (make-vector (+ (vector-length
                                     (class/get-ancestry superclass))
                                    1))))
      (class/set-ancestry! this ancestry)
      (vector-copy! (class/get-ancestry superclass) ancestry)
      (vector-set! ancestry
                   (- (vector-length ancestry) 1)
                   this))))
```

The above outline of `class-class`'s `init` method has two “holes” that need to be filled in with code that computes alists, assigning positions for the instance variables and methods. Recall that all the superclass's instance variable or method names need to be assigned the same positions as in the superclass, with the new names assigned larger position numbers. Thus, a simple approach is to cons new name/position associations onto the front of the superclass's alists, starting with the superclass's number of instance variables or methods as the next available position number. The following procedure handles this process of building onto an existing alist:

```
(define alist-from-onto
  (lambda (names num alist)
    (if (null? names)
        alist
        (alist-from-onto (cdr names)
                         (+ num 1)
                         (cons (list (car names)
                                     num)
                               alist)))))
```

Using this procedure, the two pieces of code we need to fill into the holes are as follows:

```
(alist-from-onto instvar-names
                 (class/get-num-ivars superclass)
                 (class/get-ivar-alist superclass))
```

and

```
(alist-from-onto method-names
                 (class/get-num-methods superclass)
                 (class/get-method-alist superclass))
```

With these filled in, the `init` method will now work; however, it won't provide the user with any error message if they pick a name for an instance variable or method that is already in use from one of the ancestor classes. To provide this error-checking, we need to check to see if any element of `instvar-names` or `method-names` appears in the corresponding alist of the superclass. To check to see if a particular name appears in the alist, we can use the built-in procedure `assq`, which searches the alist for a particular name and returns the first pair that has that name as its car or returns `#f` if the name isn't found. For example,

```
(assq 'size '((class 0) (size 1)))
(size 1)

(assq 'color '((class 0) (size 1)))
#f
```

This procedure makes checking for erroneously reused names fairly straightforward.

Exercise 14.31

Write the error-checking (and reporting) code.

The only remaining features of the class class we need to implement are the `ivar-position` and `method-position` methods. These methods can also be straightforwardly written using `assq`:

```
(class/set-method!
 class-class 'ivar-position
 (lambda (this ivar-name)
   (let ((lookup (assq ivar-name (class/get-ivar-alist this))))
     (if lookup
         (cadr lookup)
         (error "instance variable name not present in class"
                ivar-name (class/get-name this))))))
```

```
(class/set-method!
 class-class 'method-position
 (lambda (this method-name)
   (let ((lookup (assq method-name (class/get-method-alist this))))
     (if lookup
         (cadr lookup)
         (error "method name not present in class"
                method-name (class/get-name this))))))
```

The class class's `init` method in turn invokes the object class's `init` method, using `object^init`. This is just another example of our general practice of having each class's `init` give the superclass's `init` a chance to do any initializing it needs to. As it happens, there is no initializing to do in the object class, so we simply write a place holder:

```
(class/set-method!
 object-class 'init
 (lambda (this) 'done))
```

The next two subsections address the bootstrapping problem and the question of how the `define-class` abbreviation is expanded. Other than these issues, all we have omitted from our description of the object-oriented programming system are the features used for debugging and exploration: the object class's `describe` method, the class class's overriding `describe` method, and the `show-class-hierarchy` procedure. The material above, showing how classes and objects are structured, provides the information you would need to write these debugging tools.

Exercise 14.32

Either write the debugging tools, or write English explanations of our definitions of them, which you can find in the complete version of the object-oriented programming system that is on the web site for this book.

Bootstrapping

Having thus far ignored the circularities inherent in using the object-oriented programming system to implement itself, we now are faced with the following problems:

1. The `class-class` can't possibly be made using `make-class` because what `make-class` does is to instantiate `class-class`.

2. The `object-class` can't be made using `make-class` either. Not only does the same circularity issue apply (because `make-class` would be trying to create `object-class` by instantiating one of its subclasses, `class-class`), but in addition `object-class` has no superclass.
3. Many of the getters, setters, and methods of `object-class` and `class-class` can't be obtained in the usual way, using `class/getter`, `class/setter`, and `class/method`, because of circularity problems. For example, `class/method` clearly can't be used to get the `method` method of the `class-class`, because that would mean it was being used to get itself.

The simplest approach to the problem of creating `class-class` is simply to build it "by hand" as a 10-element vector (because there are 10 instance variables) with the appropriate contents:

```
(define class-class
  (vector 'class-class-goes-here
          'class ; name
          '()    ; subclasses
          10     ; num-ivars
          '((class 0)      ; ivar-alist (These position numbers must
            (name 1)       ; be matched by the actual positioning
            (subclasses 2) ; of the items in this class-class vector
            (num-ivars 3)  ; as well as the ones in the object-class
            (ivar-alist 4) ; vector below.)
            (num-methods  5)
            (method-alist 6)
            (method-vector 7)
            (method-set?-vector 8)
            (ancestry 9))
          11     ; num-methods
          '((init 0) ; method-alist
            (describe 1)
            (instantiator 2)
            (predicate 3)
            (getter 4)
            (setter 5)
            (method 6)
            (non-overridable-method 7)
            (set-method! 8)
            (ivar-position 9)
            (method-position 10))
          ;;(continued)
```

```
         (make-vector 11)    ; method-vector
         (make-vector 11)    ; method-set?-vector
         (make-vector 2)))   ; ancestry
```

Of the 10 positions in this vector, one of them needed to be temporarily filled in with a place holder rather than the real value: Position 0 is supposed to be the class of which the object is an instance, but in the case of `class-class`, it is an instance of itself. Thus, we put the placeholder `'class-class-goes-here` in the preceding and now can fix it:

```
(unchecked-object/set-class! class-class class-class)
```

At this point, the `class-class` has been constructed, but the vectors it contains—its `method-vector`, `method-set?-vector`, and `ancestry`—still need to be filled in. We'll take care of that later.

The `object-class` can be constructed analogously:

```
(define object-class
  (vector class-class         ; class
          'object             ; name
          (list class-class)  ; subclasses
          1                   ; num-ivars
          '((class 0))        ; ivar-alist
          2                   ; num-methods
          '((init 0)          ; method-alist
            (describe 1))
          (make-vector 2)     ; method-vector
          (make-vector 2)     ; method-set?-vector
          (make-vector 1)))   ; ancestry
```

Now we can move on to the getters and setters. For these, we'll write quick-and-dirty versions of those actually needed during the bootstrapping process, and then once the bootstrapping is over, we can (re)define them all the normal way, using `class/getter` and `class/setter`. That will fill in all the ones we skipped over because they weren't needed for bootstrapping, and moreover it will replace all our quick-and-dirty versions (i.e., ones without error-checking) with the normal versions with error-checking. For example, to fill in the above two classes' ancestry vectors, we need `class/get-ancestry`, so we'll define it as follows:

```
(define class/get-ancestry (lambda (obj) (vector-ref obj 9)))
```

Note that the number 9 needs to match the position given in the instance variable alist that is part of `class-class`. The other getters and setters we need can be defined similarly.

Once these are in place, we can fill in some missing details in the classes to show that no methods have been set yet and to show the correct ancestries:

```
(vector-fill! (class/get-method-set?-vector object-class) #f)

(vector-fill! (class/get-method-set?-vector class-class) #f)

(vector-set! (class/get-ancestry object-class)
             0
             object-class)

(let ((a (class/get-ancestry class-class)))
  (vector-set! a 0 object-class)
  (vector-set! a 1 class-class))
```

Once the two fundamental classes are constructed and the necessary getters and setters are jury-rigged, we can set to work defining the various methods. However, we again run into some circularity problems. For example, we can't very well use `class/set-method!` to install itself, can we? Well, we actually can, if we do it carefully, as follows:

```
(define class/method-position ; temporary real, later replaced
  (lambda (this method-name)  ; with virtual
    (let ((lookup (assq method-name
                        (class/get-method-alist this))))
      (if lookup
          (cadr lookup)
          (error "method name not present in class"
                 method-name (class/get-name this))))))

(define class/set-method! ; temporary real, later replaced
  (lambda (this method-name method)         ; with virtual
    (let ((index (class/method-position this method-name)))
      (vector-set! (class/get-method-vector this)
                   index
                   method)
      (vector-set! (class/get-method-set?-vector this)
                   index
                   #t)
      ;;(continued)
      (apply-below this
                   (lambda (class)
                     (vector-set! (class/get-method-vector class)
                                  index
                                  method))
                   (lambda (class)
                     (not (vector-ref
                           (class/get-method-set?-vector class)
                           index))))
    method-name))

(class/set-method! class-class 'method-position
                   class/method-position)

(class/set-method! class-class 'set-method!
                   class/set-method!)

; similarly for other methods, including class/method

(define class/method-position
  (class/method class-class 'method-position))

(define class/set-method!
  (class/method class-class 'set-method!))

; and so forth
```

You'll notice that the name `class/set-method!` is temporarily defined to be the specific real method rather than the virtual method. If we had any subclasses of `class-class` during the bootstrapping process, this definition would be a problem, but we don't. Once the bootstrapping is complete, we redefine `class/set-method!` to be the virtual method, which is obtained in the usual way.

The three preceding techniques are all it takes to get the object-oriented programming system off the ground: hand construction of the `class-class` and `object-class` vectors, hand construction of temporary versions of some getters and setters, and temporary definition of method names such as `class/set-method!` and `class/method` to be the real methods rather than the virtual ones. If you want to see all the details, such as exactly which getters and setters need to be provided by hand, you can look at the full implementation of the object-oriented programming system, which is on the web site for this book. (Or, you could work it out for yourself.)

Define-Class

At this point we have a working object-oriented programming system but not one that is very pleasant to use because for each class we need to first make the class using `make-class`, then get its instantiator using `class/instantiator`, then its predicate using `class/predicate`, and then all its getters and setters and methods.

The solution, which we employed in our earlier object-oriented programming, is to use `define-class` as an abbreviation for this tedious list of definitions. Although this is just a superficial abbreviation of many definitions by one definition, it is worth understanding, and so in this subsection we'll look at the mechanism that is used.

We can implement `define-class` using a two-step process: First the definitions for the class are computed, and then those definitions are evaluated "globally," that is, as though they had been typed in:

```
(define define-class
  (lambda (class-name superclass instvar-names method-names)
    (eval-globally
     (class-definitions
      class-name superclass instvar-names method-names))))
```

This code still leaves the problems of writing `class-definitions`, which produces the long list of definitions for a class, and `eval-globally`, which makes those definitions take effect.

To write `class-definitions`, we'll need some way of "gluing together" symbols so that we can take a class name, such as `widget` and glue `make-` onto the front to make the instantiator name `make-widget` or glue a `?` onto the end to make the predicate name `widget?` or glue `/` and the method name `activate` onto the end to make the name `widget/activate`, etc. We can write a procedure to do this gluing together, which we'll call `symbol-append`. Here are some examples of `symbol-append` in use:

```
(symbol-append 'make- 'widget)
make-widget
```

```
(symbol-append 'widget '/ 'activate)
widget/activate
```

We can write `symbol-append` in terms of some built-in procedures: `symbol->string`, for converting the symbols to strings; `string-append`, for gluing the strings together; and `string->symbol` for converting the result back into a symbol. Here is the definition:

```
(define symbol-append
  (lambda symbols
    (string->symbol
      (apply string-append
             (map symbol->string symbols)))))
```

Now that we have this tool in hand, writing `class-definitions` is simply a large amount of relatively boring list-construction code. As an example of one little piece of it, consider the following definition:

```
(define class-predicate-definition
  (lambda (class-name)
    (list 'define (symbol-append class-name '?)
          (list 'class/predicate
                (symbol-append class-name '-class)))))
```

If we apply that procedure to the class name `widget`, we get the definition that would need to be evaluated in order to define `widget?`:

```
(class-predicate-definition 'widget)
(define widget? (class/predicate widget-class))
```

All the definitions that the `define-class` is abbreviating can be produced similarly. The various definitions can then be packaged together by wrapping them in a list starting with `begin`, as in

```
(begin
  (define widget-class ...)
  (define make-widget ...)
  (define widget? ...)
  ...)
```

If you want to see the full, tedious code for generating the necessary definitions as lists, you can find it in the full version of the object-oriented programming system that is on the web site for this book. However, the previous sample should suffice to indicate the general mechanism by which `class-definitions` works.

Now that the definitions have been produced as a list structure, the remaining problem is how to make the Scheme system process those definitions; this is the job of `eval-globally`. Most Scheme systems have an `eval-globally` procedure already built in, but typically it is under a different name. For example, it might just be called `eval`, in which case all you need to do is `(define eval-globally eval)`. However, the R^{4}RS standard for Scheme doesn't specify any version of this procedure at all. Therefore, the system-independent version of our software takes a somewhat more roundabout approach, but one that works in nearly any

Scheme system. Namely, `eval-globally` writes the definitions out to a file called `evaltemp.scm` and then does `(load "evaltemp.scm")` to load the file in. Because the R^{4}RS standard specifies both the mechanisms necessary for writing to a file and for loading a file in, the only problem that may arise is with the choice of the filename.

As before, if you are interested in the details, we invite you to look at the definition of `eval-globally` in the full version of the object-oriented programming system that we distribute. (The web site also has versions that have been tailored to specific Scheme systems.)

14.5 An Application: Adventures in the Imaginary Land of Gack

Although object-oriented programming is now used for developing every imaginable kind of software, its historical roots are in the development of *simulation* systems, in which the software's objects are used to model the interactions between the real-world objects being simulated. Simulations are used for many practical purposes, such as studying the effectiveness of emergency preparedness plans without needing a real emergency. However, this category of software has also begotten a derivative category with no more practical purpose than having fun: *adventure games*. In an adventure game, the simulation is of a fantasy world and the characters inhabiting it. Typically the player of the game controls one character, and the others are automated. As fun as adventure games can be to play, they are nowhere near as fun to play as they are to construct because in their construction you can exercise unbounded creativity. In this section we'll give you the opportunity to exercise your own creativity in constructing a simulated world for an adventure game, using object-oriented programming as the underlying technology.

We obtained the idea of using an adventure game to illustrate object-oriented programming from Harold Abelson and Gerald Jay Sussman, who together with their colleagues at MIT developed an adventure game in Scheme for use in their course, Structure and Interpretation of Computer Programs. Their game was designed to have places and characters that would be familiar to their students at MIT, and our game started its life simply as an attempt to relocate to places and characters that would be more familiar to our own students. Since then it has evolved, both in terms of the underlying technology and, to a lesser extent, the game, to become increasingly dissimilar from the MIT game. We nonetheless owe a great debt to Abelson, Sussman, and the rest of the MIT team because the game is still recognizably theirs at heart. We use it here with their permission. We would also like to encourage others to engage in the same sort of "localization" we did; rather than just adding on to the base set of locations and characters we provide, how about completely replacing them with ones more familiar to you or more to your own imaginative taste?

Our game, which we call Adventures in the Imaginary Land of Gack, has three major components. The most important of these is a hierarchy of classes for representing people, places, and things. Subclassing is used to achieve many of the special effects of the game, for example, special kinds of automated people who behave differently than normal automated people do. The second component of the game is the particular world, the Land of Gack, which is produced simply by creating specific instances of the various classes, such as a specific automated person named Max, and establishing the relationships between them, such as positioning Max in the offices. The third and final component of the game is a user interface, which allows the player to interact with the game; this component is based on the pattern/action idea that we first introduced in the movie query system.

In contrast to most of the prior chapters' application sections, in which we've chosen to jointly develop an application program with our readers by alternating between portions we supply and exercises for the readers, in this section we present a complete version of the adventure game and then call upon you to extend it in whatever directions suit your fancy (after studying our version, which includes playing it, of course). We provide some suggestions, but they are just starting points.

The class hierarchy portion of our game can be seen in overview most easily by loading it in and then evaluating `(show-class-hierarchy)`. If you do this, you'll see the following output:

```
object
   registry
   named-object
      thing
         scroll
      place
      person
         auto-person
            wizard
            witch
   class
```

We can also draw a UML class diagram of this hierarchy, as in Figure 14.1. As you can see, there are two "top-level" classes in the game: registry and named-object. The named-object class clearly plays an important role in that its subclasses are used for representing all the things, places, and persons in the simulated Land of Gack. What these have in common is that they have names. You can also see that we've provided a specialized kind of person, the auto-person, for the characters that are controlled by the program rather than by the player. These characters can all act, although some of them act differently than others. In particular, there are wizards and witches with distinctive behavior patterns. Similarly there could be things or places that behave in supernormal ways. We've left most of those possibilities unexplored (leaving them for you) but provide a simple illustration with a special kind of thing, scrolls that can be read.

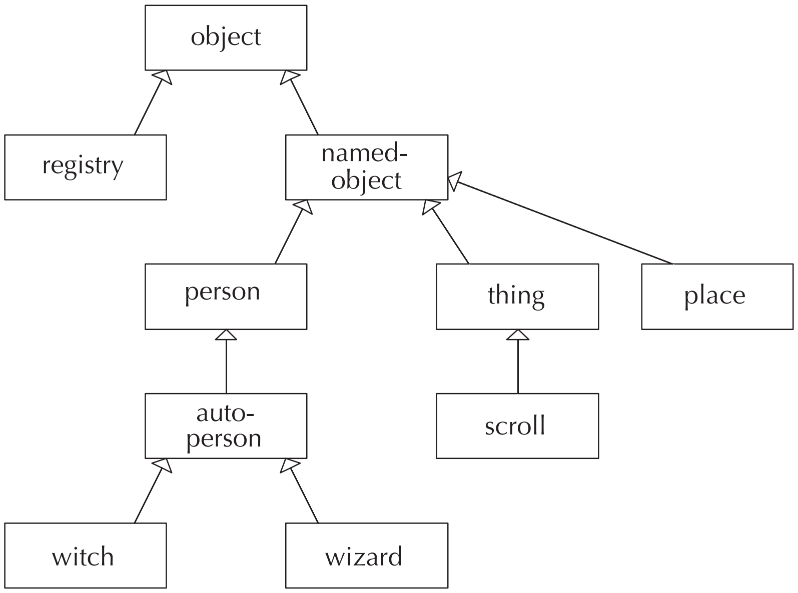


Figure 14.1 The class hierarchy for the Land of Gack

We'll return to the named-object class and its various descendants in more detail shortly, but first we should explain the one other class, the registry class. This class is used to create a single object that serves as a central registry of all the auto-persons who are roaming the Land. To preserve this property, whenever an auto-person is created, the `auto-person/init` method registers it with the registry. When an auto-person becomes no longer free to act (because a witch turned the auto-person into a frog—more on that later), the auto-person is removed from the registry. The registry is used to give all the automated characters a chance to act after each action taken by the player. This makes important use of the generic operations inherent in object-oriented programming. The registry goes down its list of auto-persons applying `auto-person/maybe-act` to each of them in a uniform way, without needing to know or care that some might be witches or wizards. Yet the appropriate kind of action is triggered in each case.

Before we continue with our description of the registry class, we really should take a moment to update our UML class diagram with the new association we have identified between the registry class and the auto-person class. While we're at it, we'll give you a sneak preview (or overview) of the remaining associations by putting them in Figure 14.2 as well. You should be able to look over the diagram and make some sense of it. For example, you can see that there is a bidirectional association between the person and thing classes, whereby each person knows the things that are its possessions, and conversely each thing knows its owner, if any. Each person also knows its place, and the place knows its contents. One new notation is used

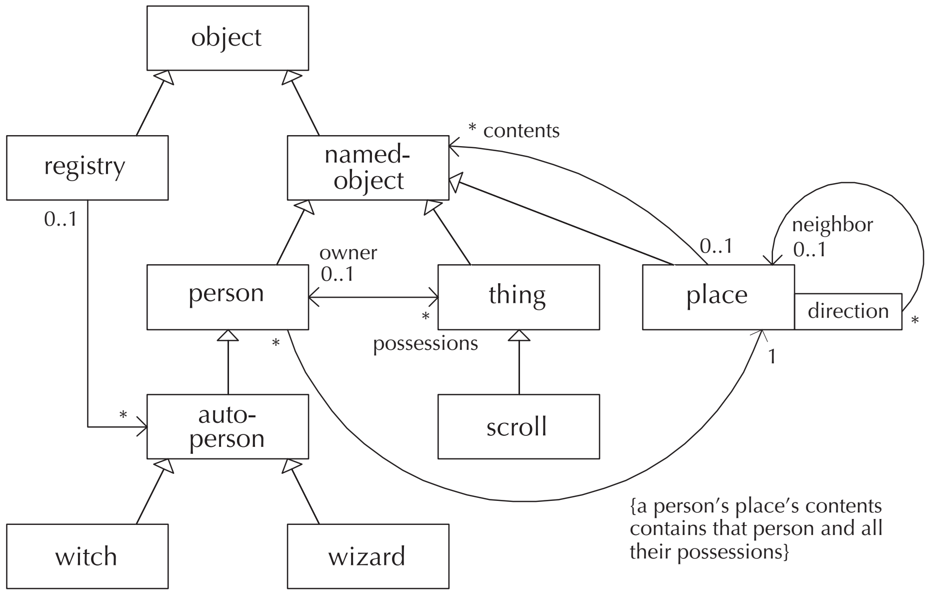


Figure 14.2 The Land of Gack class diagram, including associations

for the neighbor relationship between places. A box labeled "direction" is on the side of the place class, with an arrow back to the place class. This notation means that starting from a particular place object, and given a particular direction, we can then get to the neighboring place in that direction—except that there might not be any neighbor in that direction, a contingency we are warned of by the multiplicity of 0..1.

Having seen the overview diagram, we can look at the individual classes in more detail. To start with, let's return to the registry class. We'll give the registry methods for adding and removing an auto-person, a method for triggering all the auto-persons, and finally a method for repeatedly triggering all the auto-persons, with the repetition count specified by an argument. This last method, `trigger-times`, provides a convenient way for the game to provide variable difficulty levels because it allows for a faster-paced game in which each auto-person gets multiple opportunities to act after each action of the player. The code is as follows:

```
(define-class
  'registry
  object-class
  '(list)
  '(add
    remove
    trigger
    trigger-times))
```

```
(class/set-method!
 registry-class 'init
 (lambda (this)
   (object^init this)
   (registry/set-list! this '())))

(class/set-method!
 registry-class 'add
 (lambda (this person)
   (registry/set-list! this
                       (cons person
                             (registry/get-list this)))))

(class/set-method!
 registry-class 'remove
 (lambda (this person)
   (registry/set-list! this
                       (delq person
                             (registry/get-list this)))))

(class/set-method!
 registry-class 'trigger
 (lambda (this)
   (for-each auto-person/maybe-act
             (registry/get-list this))))

(class/set-method!
 registry-class 'trigger-times
 (lambda (this n)
   (if (> n 0)
       (begin (registry/trigger this)
              (registry/trigger-times this (- n 1)))
       'done)))
```

The `remove` method relies on a procedure called `delq`. This procedure takes an object and a list and returns a new list that looks like the old one with all occurrences of the object removed from it. We can define `delq` as follows:

```
(define delq
  (lambda (item list)
    (filter (lambda (x)
              (not (eq? x item)))
            list)))
```

Now we can move on to the named-object class and its descendants, which are used to model persons, places, and things from the Land of Gack. These objects use instance variables to keep track of their relationships; for example, a person has a list of the things it possesses. Because each thing also has an instance variable recording the person who owns it, this portion of the game also provides a good reinforcement of the importance of representation invariants as always-preserved *consistency conditions*. Clearly one invariant property must be that the person objects' possession lists and the things' owner instance variables remain in agreement; whenever the owner of a thing is changed, the thing must be removed from one person's list of possessions and added to the other's. Similar considerations apply elsewhere in the design as well. For example, each person object knows what place the person is in, and each place knows what persons (and things) are in it.

We maintain these consistency constraints by identifying one of the partners in each relationship as being "in charge." In particular, we will designate persons as being in charge of the things they own and places they are in. Therefore when a person takes or loses a thing, it will tell the thing to change its owner, and when a person moves from place to place, it will tell the places to update their lists of contents. The reverse arrangement would have also been possible, in which (for example) when a thing changed its owner, it told the persons involved to update their lists of possessions. However, having decided which object is in charge, we simply need to initiate all changes through that object, leaving it to update the subordinate partner object.

The named-object class itself is quite simple because it captures only the notion that the object has a name, which can potentially be changed:

```
(define-class
  'named-object
  object-class
  '(name)
  '(name
    change-name))

(class/set-method!
 named-object-class 'init
 (lambda (this name)
   (object^init this)
   (named-object/set-name! this name)))

(class/set-method!
 named-object-class 'name
 (lambda (this)
   (named-object/get-name this)))
```

```
(class/set-method!
 named-object-class 'change-name
 (lambda (this new-name)
   (named-object/set-name! this new-name)))
```

Of the subclasses of `named-object-class`, the simplest is the class for things because all that distinguishes a plain thing (other than its name) is who owns it. The owner is normally a person object but can also be the symbol `no-one`, in which case the `owned?` predicate returns false. Initially a thing is owned by no one, but it can become owned by a person or become unowned again:

```
(define-class
  'thing
  named-object-class
  '(owner)
  '(owned?
    owner
    become-unowned
    become-owned-by))

(class/set-method!
 thing-class 'init
 (lambda (this name)
   (named-object^init this name)
   (thing/set-owner! this 'no-one)))

(class/set-method!
 thing-class 'owned?
 (lambda (this)
   (not (equal? (thing/owner this)
                'no-one))))

(class/set-method!
 thing-class 'owner
 (lambda (this)
   (thing/get-owner this)))

(class/set-method!
 thing-class 'become-unowned
 (lambda (this)
   (thing/set-owner! this 'no-one)))
```

```
(class/set-method!
 thing-class 'become-owned-by
 (lambda (this person)
   (thing/set-owner! this person)))
```

A scroll is simply a thing that can be read. (Its name is the title of the scroll.) A normal scroll is pretty boring to read, but we are including scrolls in the game primarily so that you can add subclasses that behave more interestingly when read. Here is the scroll class:

```
(define-class
  'scroll
  thing-class
  '()
  '(be-read))

(class/set-method!
 scroll-class 'init
 (lambda (this title)
   (thing^init this title)))

(class/set-method!
 scroll-class 'be-read
 (lambda (this)
   (let ((owner (scroll/owner this))
         (title (scroll/name this)))
     (if (scroll/owned? this)
         (person/say owner
                     (list "I have read" title))
         (display-message (list "No one has" title))))))
```

As you can see, although normal scrolls may not do anything very exciting when read, even the little that they do accomplish requires some moderately interesting machinery. In particular, we have a good example of an object's method invoking a method on a related object—the scroll asks its owner to say that he or she has read the scroll. That way, if some people have unusual ways of speaking, when they read a scroll, they will report having done so in their own peculiar way. Because only a scroll's owner is allowed to read it, we report any attempt to read an unowned scroll. This reporting is done using a `display-message` procedure that takes a list of items (such as strings or symbols) to display; it is defined as follows:

```
(define display-message
  (lambda (list-of-stuff)
    (newline)
    (for-each (lambda (s) (display s) (display " "))
              list-of-stuff)))
```

The place class represents the places in the Land of Gack, or "rooms" as they are frequently called by adventure gamers, whether they are actually rooms or not. Each place has a list of contents, which are the things and persons that are at the place, and has the ability to gain or lose such an item (when they arrive or leave). Each place also has a `neighbor-map`, which is a list of pairs used as an association list, associating each exit direction with the neighboring place in that direction. For example, if a pair in the list has the symbol `up` as its car, the cdr is the place object that one would arrive at by going up. This information regarding the spatial connections between places can be accessed via a number of different methods. The `exits` method simply returns the list of valid direction symbols (such as `up`). The `neighbors` method, in contrast, supplies a list of the actual neighboring places. Finally, one can determine which neighbor is in a particular direction using `neighbor-towards` or add a new neighbor in a particular direction using `add-new-neighbor`. The class and its methods are as follows:

```
(define-class
  'place
  named-object-class
  '(neighbor-map ; pairs: car = direction, cdr = neighbor
    contents)    ; people and things
  '(exits
    neighbors
    neighbor-towards
    add-new-neighbor
    gain
    lose
    contents))

(class/set-method!
 place-class 'init
 (lambda (this name)
   (named-object^init this name)
   (place/set-neighbor-map! this '())
   (place/set-contents! this '())))
```

```
(class/set-method!
 place-class 'exits
 (lambda (this)
   (map car (place/get-neighbor-map this))))

(class/set-method!
 place-class 'neighbors
 (lambda (this)
   (map cdr (place/get-neighbor-map this))))

(class/set-method!
 place-class 'neighbor-towards
 (lambda (this direction)
   (let ((p (assq direction
                  (place/get-neighbor-map this))))
      (if (not p)
          #f
          (cdr p)))))

(class/set-method!
 place-class 'add-new-neighbor
 (lambda (this direction new-neighbor)
   (let ((neighbor-map (place/get-neighbor-map this)))
     (if (assq direction neighbor-map)
         (display-message
          (list "there is already a neighbor"
                direction
                "from"
                (place/name this)))
         (place/set-neighbor-map! this
                                  (cons (cons direction
                                              new-neighbor)
                                        neighbor-map))))))

(class/set-method!
 place-class 'gain
 (lambda (this new-item)
   (let ((contents (place/contents this)))
     (if (memq new-item contents)
         (display-message
          (list (named-object/name new-item)
                ;;(continued)
                "is already at"
                (place/name this)))
         (place/set-contents! this
                              (cons new-item contents))))))

(class/set-method!
 place-class 'lose
 (lambda (this item)
   (let ((contents (place/contents this)))
     (if (not (memq item contents))
         (display-message
          (list (named-object/name item)
                "is not at"
                (place/name this)))
         (place/set-contents! this
                              (delq item contents))))))

(class/set-method!
 place-class 'contents
 (lambda (this)
   (place/get-contents this)))
```

The most interesting of the various kinds of named objects are the persons. They have a variety of methods intended for internal use within the game but also methods that, in effect, are directly used by the human player of the game. Strictly speaking, these are used by the user interface component of the game. For example, when the player types in to the user interface the command `(go north)`, the effect is the same as if the user had evaluated the expression `(person/go player 'north)`, although actually the user interface evaluated it on the user's behalf.

The person class and its methods follow. The methods make use of a utility procedure that we define afterward, `verbalize-list`, which turns a list of items into a spoken form by inserting "and" between consecutive words and using a specified replacement if there are no items. In reading the methods below, remember that the person class is "in charge" of its reciprocal relationships with things and places and hence needs to keep those up to date. The code is as follows:

```
(define-class
  'person
  named-object-class
  '(place
    possessions)
  ;;(continued)
  '(say
    look-around
    list-possessions
    read
    have-fit
    move-to
    go
    take
    lose
    place
    possessions
    greet
    other-people-at-same-place))

(class/set-method!
 person-class 'init
 (lambda (this name place)
   (named-object^init this name)
   (person/set-place! this place)
   (person/set-possessions! this '())
   (place/gain place this)))

(class/set-method!
 person-class 'say
 (lambda (this list-of-stuff)
   (let ((name (person/name this))
         (place (person/place this)))
     (let ((place-name (place/name place)))
       (display-message
        (append (list "At" place-name ":" name "says --")
                list-of-stuff))))))

(class/set-method!
 person-class 'look-around
 (lambda (this)
   (let ((place (person/place this)))
     (let ((other-items
            (map named-object/name
                 (delq this
                       (place/contents place))))
           (exits (place/exits place)))
       ;;(continued)
       (person/say this
                   (append '("I see")
                           (verbalize-list other-items "nothing")
                           '("and can go")
                           (verbalize-list exits "nowhere")))))))

(class/set-method!
 person-class 'list-possessions
 (lambda (this)
   (let ((stuff (map thing/name
                     (person/possessions this))))
     (person/say
      this
      (append '("I have")
              (verbalize-list stuff "nothing"))))))

(class/set-method!
 person-class 'read
 (lambda (this scroll)
   (if (eq? this (scroll/owner scroll))
       (scroll/be-read scroll)
       (display-message
        (list (person/name this)
              "does not have"
              (scroll/name scroll))))))

(class/set-method!
 person-class 'have-fit
 (lambda (this)
   (person/say this '("Yaaaah! I am upset!"))))

(class/set-method!
 person-class 'move-to
 (lambda (this new-place)
   (let ((name (person/name this))
         (old-place (person/place this))
         (possessions (person/possessions this)))
     (display-message
      (list name "moves from" (place/name old-place)
            "to" (place/name new-place)))
     (place/lose old-place this)
     ;;(continued)
     (place/gain new-place this)
     (for-each (lambda (p)
                 (place/lose old-place p)
                 (place/gain new-place p))
               possessions)
     (person/set-place! this new-place)
     (person/greet this
                   (person/other-people-at-same-place this)))))

(class/set-method!
 person-class 'go
 (lambda (this direction)
   (let ((old-place (person/place this)))
     (let ((new-place (place/neighbor-towards
                       old-place
                       direction)))
       (if new-place
           (person/move-to this new-place)
           (display-message
            (list "you cannot go" direction "from"
                  (place/name old-place))))))))

(class/set-method!
 person-class 'take
 (lambda (this thing)
   (if (eq? this (thing/owner thing))
       (display-message
        (list (person/name this) "already has" (thing/name thing)))
       (begin
         (if (thing/owned? thing)
             (let ((owner (thing/owner thing)))
               (person/lose owner thing)
               (person/have-fit owner))
             'unowned)
         (thing/become-owned-by thing this)
         (person/set-possessions!
          this
          (cons thing (person/possessions this)))
         (person/say
          this
          (list "I take" (thing/name thing)))))))
```

```
(class/set-method!
 person-class 'lose
 (lambda (this thing)
   (if (not (eq? this (thing/owner thing)))
       (display-message (list (person/name this) "doesn't have"
                              (thing/name thing)))
       (begin
         (thing/become-unowned thing)
         (person/set-possessions!
          this
          (delq thing (person/possessions this)))
         (person/say
          this
          (list "I lose" (thing/name thing)))))))

(class/set-method!
 person-class 'place
 (lambda (this)
   (person/get-place this)))

(class/set-method!
 person-class 'possessions
 (lambda (this)
   (person/get-possessions this)))

(class/set-method!
 person-class 'greet
 (lambda (this people)
   (if (not (null? people))
       (person/say this
                   (cons "Hi"
                         (verbalize-list
                          (map person/name people)
                          "no one")))
       'no-one-to-greet)))

(class/set-method!
 person-class 'other-people-at-same-place
 (lambda (this)
   (delq this
         (filter person?
                 (place/contents (person/place this))))))
```

The `verbalize-list` utility procedure used by the `person` class is as follows:

```
(define verbalize-list
  (lambda (items none-word)
    (define loop
      (lambda (items)
        (if (null? (cdr items))
            items
            (cons (car items)
                  (cons "and"
                        (loop (cdr items)))))))
    (if (null? items)
        (list none-word)
        (loop items))))
```

The player of the game controls a normal person, but all the remaining characters are automatically triggered by the registry and so are instances—at least indirectly—of the auto-person class. What distinguishes the auto-person class is that it has a method `maybe-act` for use by the registry. Each auto-person has some frequency with which it acts that is controlled using a threshold to which its "restlessness" has to rise before it acts. Each time the `maybe-act` method is invoked, it checks to see if the threshold has been reached, and if not, increases the restlessness by 1. When the restlessness reaches the threshold, the `maybe-act` method in turn invokes the `act` method, to trigger the automated person's behavior, and then resets the restlessness to zero. Normal auto-persons `act` by moving to a randomly chosen adjoining room, but there are subclasses that have more interesting `act` methods. The auto-person class and its methods are given below; note that the auto-person is in charge of its relationship with the registry.

```
(define-class
  'auto-person
  person-class
  '(threshold
    restlessness)
  '(maybe-act
    act))

(class/set-method!
 auto-person-class 'init
 (lambda (this name place threshold)
   (person^init this name place)
   ;;(continued)
```

```
    (auto-person/set-threshold! this threshold)
    (auto-person/set-restlessness! this 0)
    (registry/add registry this)))

(class/set-method!
 auto-person-class 'maybe-act
 (lambda (this)
   (let ((threshold (auto-person/get-threshold this))
         (restlessness (auto-person/get-restlessness this)))
      (if (< restlessness threshold)
          (auto-person/set-restlessness! this
                                         (+ 1 restlessness))
          (begin (auto-person/act this)
                 (auto-person/set-restlessness! this 0))))))

(class/set-method!
 auto-person-class 'act
 (lambda (this)
   (let ((new-place (random-element
                      (place/neighbors
                       (auto-person/place this)))))
     (if new-place
         (auto-person/move-to this new-place)))))

(define random-element
  (lambda (list)
    (if (null? list)
        #f
        (list-ref list (random (length list))))))
```

We've provided two subclasses of the `auto-person-class` with more interesting `act` methods. The witch class has the special behavior that when there is someone else in the room with the witch when she acts, she curses the person. (Otherwise she acts just like any other auto-person.) The wizard, on the other hand, is basically a scroll collector. The two classes, with their methods, follow:

```
(define-class
  'witch
  auto-person-class
  '()
  '(curse))
```

```
(class/set-method!
 witch-class 'act
 (lambda (this)
   (let ((victim (random-element
                  (witch/other-people-at-same-place this))))
     (if victim
         (witch/curse this victim)
         (auto-person^act this)))))

(class/set-method!
 witch-class 'curse
 (lambda (this person)
   (let ((person-name (person/name person)))
     (person/say this
                 (list
                  "Hah hah hah, I'm going to turn you into a frog"
                  person-name))
     (turn-into-frog person)
     (person/say this
                 (list "Hee hee" person-name
                       "looks better in green!")))))

(define turn-into-frog
  (lambda (person)
    (for-each (lambda (item) (person/lose person item))
              (person/possessions person))
    (person/say person '("Ribbitt!"))
    (person/move-to person pond)
    (registry/remove registry person)))

(define-class
  'wizard
  auto-person-class
  '()
  '())

(class/set-method!
 wizard-class 'act
 (lambda (this)
   (let ((place (wizard/place this)))
     (let ((scrolls (filter scroll?
                            (place/contents place))))
```

```
          (if (and (not (null? scrolls))
                   (not (eq? place chamber-of-wizards)))
              (begin
                (wizard/take this (car scrolls))
                (wizard/move-to this chamber-of-wizards)
                (wizard/lose this (car scrolls)))
              (auto-person^act this))))))
```

At this point we have the entire class hierarchy for the Land of Gack—or rather, the entire class hierarchy, as far as we've developed it for you. After you've played (and become bored with) the existing game, you'll be expected to invent new kinds of characters, new and magical scrolls and other kinds of things, and maybe even enchanted places that behave in unusual ways.

The next major component of the game, you will recall, is simply the "setting of the stage": in other words, the establishment of the specific individuals, places, and things that constitute the Land of Gack. For example, this is where we decide that the Land of Gack has (initially) only a single witch, named Barbara:

```
;; The "registry" is an object that keeps track of all
;; the auto-person objects that need to be given an
;; opportunity to act.

(define registry (make-registry))

;; Here we define the places in the imaginary world of Gack

(define food-service (make-place 'food-service))
(define PO (make-place 'PO))
(define alumni-hall (make-place 'alumni-hall))
(define chamber-of-wizards (make-place 'chamber-of-wizards))
(define library (make-place 'library))
(define good-ship-olin (make-place 'good-ship-olin))
(define lounge (make-place 'lounge))
(define computer-lab (make-place 'computer-lab))
(define offices (make-place 'offices))
(define dormitory (make-place 'dormitory))
(define pond (make-place 'pond))

;; One-way paths connect individual places in the world.

(place/add-new-neighbor food-service 'down PO)
(place/add-new-neighbor PO 'south alumni-hall)
```

```
(place/add-new-neighbor alumni-hall 'north food-service)
(place/add-new-neighbor alumni-hall 'east chamber-of-wizards)
(place/add-new-neighbor alumni-hall 'west library)
(place/add-new-neighbor chamber-of-wizards 'west alumni-hall)
(place/add-new-neighbor chamber-of-wizards 'south dormitory)
(place/add-new-neighbor dormitory 'north chamber-of-wizards)
(place/add-new-neighbor dormitory 'west good-ship-olin)
(place/add-new-neighbor library 'east alumni-hall)
(place/add-new-neighbor library 'south good-ship-olin)
(place/add-new-neighbor good-ship-olin 'north library)
(place/add-new-neighbor good-ship-olin 'east dormitory)
(place/add-new-neighbor good-ship-olin 'up lounge)
(place/add-new-neighbor lounge 'west computer-lab)
(place/add-new-neighbor lounge 'south offices)
(place/add-new-neighbor computer-lab 'east lounge)
(place/add-new-neighbor offices 'north lounge)

;; We define persons as follows:

;; We've chosen to define max-the-person rather than
;; redefining max, which is predefined in Scheme to
;; be a procedure for finding the largest of its numeric
;; arguments.
(define max-the-person
  (make-auto-person 'max offices 2))

(define karl
  (make-auto-person 'karl computer-lab 4))

(define barbara
  (make-witch 'barbara offices 3))

(define elvee
  (make-wizard 'elvee chamber-of-wizards 1))

(define player
  (make-person 'player dormitory))

;; and now we'll strew some scrolls around:

(define scroll-of-enlightenment
  (make-scroll 'scroll-of-enlightenment))
(place/gain library scroll-of-enlightenment)
```

```
(for-each (lambda (title)
            (place/gain library
                        (make-scroll title)))
          '(crime-and-punishment war-and-peace
                                 iliad
                                 collected-works-of-rilke))

(define unix-programmers-manual
  (make-scroll 'unix-programmers-manual))
(place/gain computer-lab unix-programmers-manual)

(define next-users-reference
  (make-scroll 'next-users-reference))
(place/gain computer-lab next-users-reference)
```

Finally, we need a user interface for the game so that the player can say things like `(go north)` instead of `(person/go player 'north)`, as you would have to do to play from the normal read-eval-print loop. The user interface also makes the game more fun by triggering the registry after each move to give the automated persons a chance to do their things. The user interface is entered via the `play` procedure, by simply evaluating `(play)`. This procedure is based on the pattern/action-list idea we've seen in earlier chapters. Each command you can execute has a particular pattern. For example, the pattern for commands like `(go north)` and `(go up)` is `(go _)`.

One additional kind of pattern has been added, beyond those described in Chapter 7. A pattern can contain not only symbols and wild cards but also predicate procedures. For example, the pattern for commands for taking things is made using the expression `(list 'take thing?)`, which makes a pattern rather like `(take _)`, except that the word after `take` is restricted to be the name of a thing that is in the place where the player is. The requirement that the object satisfies the predicate (as well as has the right name and is nearby) prevents you from committing kidnaping by saying `(take max)`, because Max is not a thing. A similar pattern, made using `(list 'read scroll?)`, ensures that you can only read scrolls, not other kinds of things or people. These predicate wild cards also behave specially once it has been determined that the pattern matches, and the action procedure is being applied. For example, suppose you are in the library with the `scroll-of-enlightenment` and you say `(take scroll-of-enlightenment)`. The pattern matches, because the scroll is a thing, has the right name, and is in the same place as the player. Now the action procedure is applied. However, rather than being applied to the name `scroll-of-enlightenment`, the way it would if the pattern were `(take _)`, the action procedure is instead applied to the scroll object itself.

The code for the user interface is as follows:

```
(define difficulty 1)

(define play
  (lambda ()
    (define loop
      (lambda ()
        (newline)
        (let ((user-input (read)))
          (if (equal? user-input '(quit))
              'done
              (begin
                (respond-to-using user-input gack-p/a-list)
                (loop))))))
    (newline)
    (display "Enter your name, using one word only, please.")
    (newline)
    (person/change-name player (read))
    (display-message
     (list "OK," (person/name player)
           "enter your commands one by one"
           "as scheme lists; to get help enter (help)."))
    (loop)))

(define gack-p/a-list
  (list (make-pattern/action '(help)
                             (lambda ()
                               (newline)
                               (display "Possibilities:")
                               (newline)
                               (for-each (lambda (command)
                                           (display " ")
                                           (display command)
                                           (newline))
                                         '((help)
                                           (quit)
                                           (drop thing)
                                           (lose thing)
                                           (take thing)
                                           ;;(continued)
```

```
                                   (go direction)
                                   (read scroll)
                                   (inventory)
                                   (list possessions)
                                   (look)
                                   (look around)
                                   (say ...)))
                       (newline)))
(make-pattern/action (list '(drop lose) thing?)
                     (lambda (verb thing)
                       (person/lose player thing)
                       (registry/trigger-times
                        registry difficulty)))
(make-pattern/action (list 'take thing?)
                     (lambda (thing)
                       (person/take player thing)
                       (registry/trigger-times
                        registry difficulty)))
(make-pattern/action '(go _)
                     (lambda (direction)
                       (person/go player direction)
                       (registry/trigger-times
                        registry difficulty)))
(make-pattern/action (list 'read scroll?)
                     (lambda (scroll)
                       (person/read player scroll)
                       (registry/trigger-times
                        registry difficulty)))
(make-pattern/action '(inventory)
                     (lambda ()
                       (person/list-possessions player)
                       (registry/trigger-times
                        registry difficulty)))
(make-pattern/action '(list possessions)
                     (lambda ()
                       (person/list-possessions player)
                       (registry/trigger-times
                        registry difficulty)))
(make-pattern/action '(look)
                     (lambda ()
                       (person/look-around player)))
;;(continued)
```

```
        (make-pattern/action '(look around)
                             (lambda ()
                               (person/look-around player)))
        (make-pattern/action '(say ...)
                             (lambda (stuff)
                               (person/say player stuff)
                               (registry/trigger-times
                                registry difficulty)))))

(define respond-to-using
  (lambda (command p/a-list)
    (cond ((null? p/a-list)
           (display-message '("I don't understand.")))
          ((matches? (pattern (car p/a-list)) command)
           (apply (action (car p/a-list))
                  (substitutions-in-to-match
                   (pattern (car p/a-list))
                   command)))
          (else (respond-to-using command (cdr p/a-list))))))

;; The versions of matches? and substitutions-in-to-match
;; given below not only are after doing various chapter 7
;; exercises, but moreover have an additional feature that a
;; predicate can be used as one of the components of a pattern,
;; in which case it means that at that position in the command,
;; a symbol is needed that is the name of an item in the player's
;; place that satisfies the predicate.

(define matches?
  (lambda (pattern question)
    (cond ((null? pattern)  (null? question))
          ((not (pair? question)) #f)
          ((equal? (car pattern) '_)
           (matches? (cdr pattern) (cdr question)))
          ((list? (car pattern))
           (if (member (car question) (car pattern))
               (matches? (cdr pattern)
                         (cdr question))
               #f))
          ((equal? (car pattern) '...) #t)
          ;;(continued)
          ((equal? (car pattern) (car question))
           (matches? (cdr pattern)
                     (cdr question)))
          ((procedure? (car pattern))
           (let ((object (object-with-name (car question))))
             (if (and object
                      ((car pattern) object))
                 (matches? (cdr pattern)
                           (cdr question))
                 #f)))
          (else #f))))

(define substitutions-in-to-match
  (lambda (pattern question)
    (cond ((null? pattern)
           (if (null? question)
               '()
               (error
                "substitutions-in-to-match without a match")))
          ((not (pair? question))
           (error "substitutions-in-to-match without a match"))
          ((equal? (car pattern) '_)
           (cons (car question)
                 (substitutions-in-to-match (cdr pattern)
                                            (cdr question))))
          ((list? (car pattern))
           (if (member (car question) (car pattern))
               (cons (car question)
                     (substitutions-in-to-match (cdr pattern)
                                                (cdr question)))
               (error
                "substitutions-in-to-match without a match")))
          ((equal? (car pattern) '...) (list question))
          ((equal? (car pattern) (car question))
           (substitutions-in-to-match (cdr pattern)
                                      (cdr question)))
          ;;(continued)
          ((procedure? (car pattern))
           (let ((object (object-with-name (car question))))
             (if (and object
                      ((car pattern) object))
                 (cons object
                       (substitutions-in-to-match
                        (cdr pattern) (cdr question)))
                 (error
                  "substitutions-in-to-match without a match"))))
          (else (error
                 "substitutions-in-to-match without a match")))))

(define object-with-name
  (lambda (name)
    (let ((objects (filter (lambda (obj)
                             (equal? (named-object/name obj)
                                     name))
                           (place/contents
                            (person/place player)))))
      (if (or (null? objects)
              (not (null? (cdr objects))))
          #f
          (car objects)))))
```

Now it is time for you to take charge of the game.

Exercise 14.33

First you'll need to familiarize yourself with the game. Load it and try playing the following simple game, using the `(play)` procedure. Try to get from the dormitory to the computer lab without getting turned into a frog along the way. To make it more challenging, try to get to the lab possessing the scroll of enlightenment. If you want, you can make the other characters move several times after each move of yours (rather than just once) by redefining `difficulty` to be some number larger than 1. Be sure to get familiar with the game's other features too.

Exercise 14.34

Modify the Land of Gack so that there is a new scroll, called `late-lab-report`, in the `dormitory`. Now you can play the game with a new goal: Pick up the `late-lab-report`, go catch up with `max`, wherever he may have wandered to, and try to give `max` the report even though it's late. You'll need to add some additional mechanisms to the person class and the interface's pattern/action list to be able to use a command like `(give max late-lab-report)` or `(give late-lab-report to max)`. Verify that at the end the lab report thinks `max` is its owner, `max` thinks the lab report is one of his possessions, and `player` no longer thinks the lab report is one of its possessions.

Exercise 14.35

Of course, you can make the game much more interesting by adding additional twists. Implement some that interest you. We give a list of suggestions below but feel free to come up with some of your own. Be sure to describe in English whatever new ideas you introduce.

- Add chocolate as a kind of thing. It should be able to `be-eaten`. When a chocolate is eaten, the owner loses it and it is gone from the place where the owner is located. You can optionally add cute sound effects or make a wrapper or some crumbs appear at the place. Make some chocolates appear at `food-service`.
- Modify the person class so people have an `eat` method that causes a chocolate they own to be eaten, much like the `read` method.
- Now that we have chocolates, we can modify the witch class to reflect a little-known property of witches: They can be bought off with chocolates. Change the `act` method so that if Barbara's victim has any chocolates, Barbara will take and eat one rather than turning the victim into a frog. Only victims who possess no chocolates will become frogs. This makes the game more interesting in that you have to decide whether to take the extra time to make a detour to food service for chocolates or go for speed but risk being caught unprepared by Barbara.
- If you want to make the game harder yet, you can make it difficult to hoard protective chocolates by creating a troll who wanders around eating chocolates. Perhaps you should change Max from being a normal person to being the troll—it was probably just the fact that he wrote this section that kept him from being cast as a troll in the first place.
- Introduce magic scrolls into the game as an abstract subclass of `scroll-class` that has a limited number of "charges" (let's say n) and when read the first n times invokes its `do-magic` method. Define one or more specific kinds of magic scrolls as subclasses of the magic-scroll class with interesting `do-magic` methods; for example, you might make a `scroll-of-teleportation`.
- It makes the game rather easy if the title of a magic scroll tells what it does. You could add a feature so that initially the `name` method of any magic scroll just returns `a-mysterious-scroll`, but when you read the scroll the first time the name changes to the actual title. This feature is particularly effective if you

include scrolls with undesirable effects as well as desirable ones. Of course, you could counter by including a scroll-of-identification that can be used to decode the titles.

- Currently Elvee, the wizard, always takes the first scroll at a place. Because some scrolls are more valuable than others, this action can make the game boring. Change the definition of the wizard class so wizards choose a scroll to take at random.
- Feel free to add you own interesting kinds of people, places and things.

Review Problems

Exercise 14.36

If you apply `object/describe` to most objects, you'll see the instance variables that constitute the object. On the other hand, if you apply `object/describe` to a class object (such as `item-class`), you'll instead see a description of the class that the object represents, with detailed information such as the ancestor class from which it inherits each inherited instance variable, method name, or method implementation. How can you get a description of a class object that shows the actual instance variables of the class object? (For example, the description should make the `method-set?-vector` instance variable visible.)

Exercise 14.37

a. Write a procedure that returns a list of all the classes currently in existence. (It should return the actual class objects, not the names of the classes.) *Hint:* See the definition of `show-class-hierarchy` and also the material on tree traversal in Chapter 8.

b. Write a predicate procedure that tests whether a given class has a given method name.

c. Write a procedure, using the previous two, that returns a list of the class names for all those classes in existence that have a given method name.

Exercise 14.38

In Section 13.5, we built a layered collection of data types: `ranked-binary-trees`, `binary-search-trees`, and `red-black-trees`. This layering is suggestive of object-oriented programming's class hierarchy; we can imagine having `ranked-binary-tree` as a base class, with `binary-search-tree` derived from it as a subclass, and `red-black tree` derived further as a subclass of binary search tree.

On closer examination, no reason exists to have *ranked* binary trees as the base. After all, the notion of rank only becomes relevant when we build the red-black tree abstraction. In Section 13.5, we had to include the ranks from the beginning because only the procedures for operating on the trees changed from layer to layer; the data representation stayed the same. Now with object-oriented programming, we can add a new instance variable in a subclass rather than just adding new procedures (methods).

Reimplement red-black trees in the object-oriented programming system. Start with a base class that is just plain binary trees, without ranks. Then build the binary search tree class as a subclass. Finally, implement red-black trees as a subclass of binary search trees, which is where you'll add the rank instance variable.

▷ Exercise 14.39

When a method is overridden in a subclass, the new implementation should comply with whatever the external specification of the method is so that users are not surprised. For example, it would not be appropriate for a kind of item (in the `compu-duds` example) to implement the `display` method by switching to a new randomly selected color rather than by actually displaying the item's description and price. On the other hand, the specification will no doubt have some flexibility in it; for example, we haven't pinned down exactly what sort of "description" should be displayed because if we were too precise about that, there would be no room to accommodate different kinds of subclasses.

One interesting point is that depending which of several potential specifications the designer chooses to articulate for a particular class and its methods, the exact same subclass of that class might be either "in compliance" or "out of compliance" with the specification, even though the specifications might all be reasonable for the original class.

Give a concrete illustration of this phenomenon by defining a class and its subclass and giving two reasonable specifications for the superclass, one of which leads to the subclass being compliant and the other of which doesn't. Also, write up some thoughts on the subject of how a software designer should anticipate and cope with this phenomenon.

▷ Exercise 14.40

Suppose you wanted to change the Land of Gack so that it was possible to ask people to introduce themselves; that is, you could do the following:

```
(person/introduce-self barbara)
```

or

```
(person/introduce-self max-the-person)
```

or

```
(person/introduce-self elvee)
```

Whoever you ask to introduce themselves would then say

```
Hello, I'm barbara.
I'm known to be fond of chocolate and turning people into frogs.
Pleased to meet you.
```

or

```
Hello, I'm max.
Pleased to meet you.
```

or

```
Hello, I'm elvee.
I've got this problem with scrolls.
Pleased to meet you.
```

Note that the first and last lines are similar for all people, but what comes in between (if anything) is particular to the given subclass of `person-class`. How would you arrange for this dialogue in a way that takes full advantage of the class hierarchy? Illustrate with sample code.

Exercise 14.41

In Exercise 14.34, you added to the Land of Gack the ability to give a scroll to someone else. There the intent was to give the late lab report to Max. However, you probably made the mechanism general enough that you could also give a scroll to the wizard, Elvee. If you do this, you may later see Elvee (when he acts) trying to take a scroll that he already has, which results in a message reporting this oddity. Explain how this occurs and what can be done to eliminate this behavior.

Exercise 14.42

When a person says something in the Land of Gack, it is "heard" only by the player, in that whatever is said gets displayed. It would be more interesting if other objects within the Land of Gack itself could also hear—and potentially respond to—what was said. That way you could have characters who behaved differently if you said "please," magic portals that opened when you said "open sesame," etc.

a. Add a method called `named-object/hear` that takes two arguments: the person speaking and what was said. Put an implementation in the `named-object-class` that does nothing because ordinary objects ignore what they hear.

b. Change the `person/say` method so that it invokes the `named-object/hear` method of each object in the place where the person is speaking.

c. Now have fun introducing special classes of object that respond to what they hear.

Exercise 14.43

Write a procedure `instance-of?` that can be used to test whether a particular object is an instance of a particular class. Examples of its use follow:

```
(instance-of? barbara witch-class)
#t

(instance-of? barbara person-class)
#t

(instance-of? barbara place-class)
#f

(instance-of? lounge place-class)
#t
```

Chapter Inventory

Vocabulary

object-oriented programming
class
class hierarchy
object-oriented design
subclass
superclass
object
instance
derived class
base class
ancestry
Unified Modeling Language (UML)
class diagram
association
multiplicity
class object
instance variable
inherit
method
method name
method implementation
override
getter
`this` or `self`
setter
instantiator
public
private

garbage collection	constraint (on a UML association)
augment (the superclass's method)	association list
abstract class	alist
pure abstract class	simulation
bootstrapping	adventure game
role (in a UML association)	consistency condition
virtual method	

Classes

`item-class`	`named-object-class`
`item-list-class`	`thing-class`
`oxford-shirt-class`	`place-class`
`chinos-class`	`person-class`
`object-class`	`auto-person-class`
`class-class`	`wizard-class`
`special-item-class`	`witch-class`
`pants-class`	`scroll-class`
`thrifty-item-list-class`	`chocolate-class`
`item-list-as-vector-class`	`troll-class`
`item-list-as-list-class`	`magic-scroll-class`
`registry-class`	

New Predefined Scheme Names

```
assq
string-append
load
```

Methods

`item-list/add`	`item-list/grow`
`item-list/display`	`item/init`
`item-list/total-price`	`special-item/input-specifics`
`item-list/delete`	`oxford-shirt/init`
`item-list/choose`	`oxford-shirt/display`
`item-list/empty?`	`oxford-shirt/input-specifics`
`item/input-specifics`	`chinos/init`
`item/display`	`chinos/display`
`item/revise-specifics`	`chinos/input-specifics`
`item/price`	`class/predicate`
`class/set-method!`	`class/instantiator`
`object/init`	`class/getter`
`item-list/init`	`class/setter`
`object/describe`	`class/method`

```
class/non-overridable-method
class/ivar-position
class/method-position
auto-person/init
auto-person/maybe-act
registry/init
registry/add
registry/remove
registry/trigger
registry/trigger-times
named-object/init
named-object/name
named-object/change-name
thing/owned?
thing/init
thing/owner
thing/become-unowned
thing/become-owned-by
scroll/init
scroll/be-read
place/exits
place/neighbors
place/neighbor-towards
place/add-new-neighbor
place/init
place/gain
place/lose
place/contents
person/init
person/say
person/look-around
person/list-possessions
person/read
person/have-fit
person/move-to
person/go
person/take
person/lose
person/place
person/possessions
person/greet
person/other-people-at-same-place
auto-person/act
witch/act
witch/curse
wizard/act
chocolate/be-eaten
person/eat
magic-scroll/do-magic
magic-scroll/name
```

Objects From the Land of Gack

```
registry
food-service
PO
alumni-hall
chamber-of-wizards
library
good-ship-olin
lounge
computer-lab
offices
dormitory
pond
max-the-person
karl
barbara
elvee
player
scroll-of-enlightenment
unix-programmers-manual
next-users-reference
late-lab-report
```

Other Scheme Names Defined in This Chapter

```
define-class                      eval-globally
vector-copy!                      symbol-append
display-price                     class-predicate-definition
input-integer-in-range            delq
input-selection                   display-message
compu-duds                        verbalize-list
input-item                        turn-into-frog
show-class-hierarchy              play
unchecked-object/set-class!       difficulty
apply-below                       gack-p/a-list
alist-from-onto                   respond-to-using
class-definitions                 object-with-name
```

Notes

One version of the MIT adventure game we based ours on was published in [1].

We touched only briefly on the design of object-oriented software, emphasizing the search for nouns in the problem statement as potential classes of objects and verbs and implicit operations as potential methods. Books such as those of Rumbaugh [44] and Booch [7] provide a much more extensive treatment of these issues of analyzing a problem, modeling it in terms of objects, and designing the software. Another important approach, complementing the finding of "natural objects" in the problem specification, is the recognition of common patterns of object creation and behavior; an excellent source-book for this approach is *Design Patterns*, by Gamma, Helm, Johnson, and Vlissides [21].

In addition to these general topics of design, you may want to read up on the concrete realization of object-oriented software in currently popular object-oriented programming languages. At the time we are writing this book, the greatest enthusiasm centers around the Java programming language. The next chapter contains a quick introduction to Java, and the notes at the end of that chapter provide some suggestions for further reading.

At the time of our writing, the only definitive source for information on UML is the web site, `http://www.rational.com/uml/`; documentation published on paper is still forthcoming. There is, however, an introductory overview book by Fowler [18].

The techniques we used to implement the object-oriented programming system are typical of those used for implementing such systems. Most of these techniques have been used for decades. Surprisingly, however, one of them was published as recently as 1991. That was the year when the technique we use for the class predicates, based on ancestry vectors, was described by Norman Cohen [13].