12 OBJECT-ORIENTED PROGRAMMING: SMALLTALK

12.1 HISTORY AND MOTIVATION

The second paradigm represented in fifth-generation languages is *object-oriented programming*, for which Smalltalk is the best example.

Alan Kay Saw the Potential for Personal Computers

The principal person responsible for the Smalltalk language is Alan Kay.[1] When he was a graduate student at the University of Utah in the late 1960s, he became convinced that eventually it would be possible to put the power of what was then a room-sized, million-dollar computer into a package the size of a three-ring notebook. It seemed clear that given the direction in which technology was moving, it would eventually be possible for everyone to own a personal computer of considerable power. Kay remarks, "I was almost frightened by the implication; computing as we knew it couldn't survive."

At that time it was not obvious that anyone would want a personal computer. They probably would not want to use a personal computer for the scientific and commercial applications that occupy large computers. And what language would people use to program their personal computers? Existing programming languages were designed by and for specialists and were oriented to just the sort of applications for which personal computers were unlikely to be used. It seemed to Kay that the absence of an adequate programming vehicle might be the main impediment to the success of the personal computer. With a better choice of language and interface, the reaction to the new technology might be shifted from fear to fascination (Section 1.4).

[1] In "The Early History of Smalltalk" (*SIGPLAN Notices 28*, 3, March 1993, pp. 69–95) Kay discusses the many sources of his ideas and the principal contributers to Smalltalk. This article is the source of quotations from Kay in this chapter.

Kay Investigated Simulation and Graphics-Oriented Languages

While still at the University of Utah, Kay decided that a simulation and graphics-oriented programming language could make computers accessible to nonspecialists. He already had some experience with a language he had helped design called FLEX. This language took many of its most important ideas (such as *classes* and *objects*, both discussed later) from Simula, a simulation language based on Algol-60 and designed in the 1960s.

FLEX was still too oriented toward specialists, however, so Kay also incorporated ideas from LOGO, a language designed by Seymour Papert and others at MIT. Since the early 1960s, Papert had been using LOGO to teach programming concepts to children 8–12 years old. The LOGO environment taught Kay that nonspecialists require a rich interactive environment making use of graphics and audio communication. Kay was also influenced by the sophisticated Sketchpad and GRAIL graphic communication systems and by Douglas Englebart's prophetic view of interactive computing as an "augmentation of human intellect."

The design of Smalltalk was also influenced by the research of a number of cognitive scientists and psychologists, including John Dewey, Maria Montessori, Jean Piaget, Jerome Bruner, and Seymour Papert, as well as by ideas from Plato, Leibnitz, and Marshall McLuhan.

Xerox Supported Dynabook Research

Kay proposed to Xerox Corporation his idea of a personal computer; it was called the *Dynabook.* He anticipated that in the 1980s it would be possible to put into a package the size of a notebook a computer capable of executing millions of instructions per second and of holding the equivalent of thousands of pages of information. To make this information accessible, the Dynabook would have a flat-screen, liquid crystal reflective display that would be sensitive to the touch of a finger. This would allow the user to point at things and would allow a portion of the screen to double as a keyboard. It was also intended that the Dynabook could be connected to a stereo, so that it could be used to generate music, and that it could be connected to communication lines, to provide access to large, shared data banks. Kay believed that the availability of the Dynabook would be as revolutionary as the availability of inexpensive, "personal" books after the Industrial Revolution. As Don Ihde has said, "Technologies reveal worlds."

Smalltalk Is the Language for the Dynabook

In 1971 the Xerox Palo Alto Research Center began a research project to develop the Dynabook. Smalltalk-72, the language for the Dynabook, was designed and implemented by 1972, and in 1973 a desk-sized "Interim Dynabook" became available for research. Smalltalk-72 and the Interim Dynabook were used in personal computing experiments involving over 250 children (6–15 years old) and 50 adults. One of their principles was "Simple things should be simple, complex things should be possible." Experience with Smalltalk has led to several revisions of the language, including Smalltalk-74, Smalltalk-76, Smalltalk-78, and Smalltalk-80, which is the dialect described in this chapter;[2] a draft standard has been completed.

[2] The principal source we have used is Goldberg and Robson (1983).

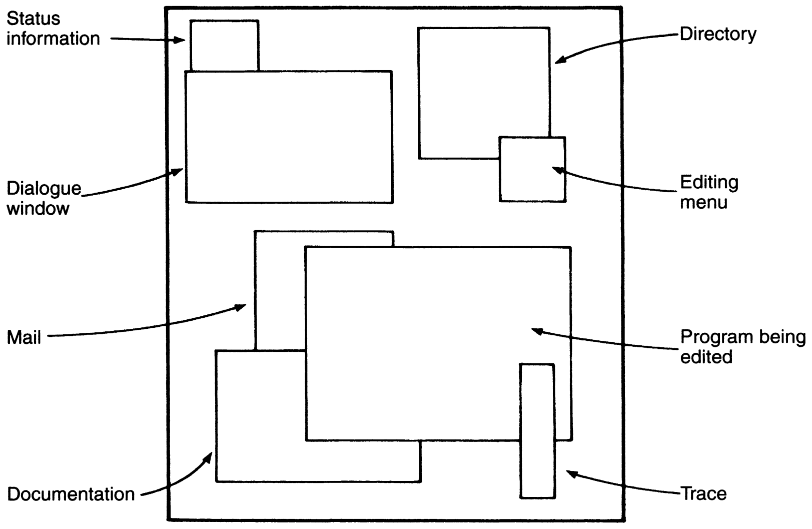


Figure 12.1 Example of a Dynabook Display

Many of the ideas developed for the Dynabook (including window-oriented display management, Figure 12.1) were later used in Apple's Lisa computer system (1983), which was one of the inspirations for the Apple Macintosh computer (introduced in 1984). It is no accident that the Dynabook window-oriented interface has become the dominant way of interacting with computers.

12.2 DESIGN: STRUCTURAL ORGANIZATION

Smalltalk Is Interactive and Interpreted

To satisfy the requirements of the Dynabook, Smalltalk has to be a highly interactive language. Although much of the user's communication with Smalltalk is accomplished by pointing, it is possible to type commands to be executed in a "dialogue" window. This style of programming is similar to that used with LISP: The user types commands to the system that either define things or call for the execution of expressions involving things already defined.

There are two primary ways to define things in Smalltalk. The first *binds* a name to an object. For example,

```
x ← 3.
y ← x + 1
```

binds 'x' to the object 3 and 'y' to the object 4. This is analogous to an application of `set` in LISP and to an assignment statement in other languages. The other way to define things is by a *class definition.* Classes are similar to Ada packages; they are discussed later.

The user requests the evaluation of an expression by typing an expression such as x*2. The dialogue looks like this:

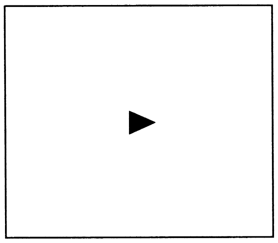
```
x*2
   6
```

Smalltalk interprets this action as sending the message '*2' to the object x. This rather unusual view of expressions is explained next.

Objects React to Messages

Since Smalltalk provides a unique programming environment, we will begin by showing a typical Smalltalk session. We have said that Smalltalk is graphics oriented and that it borrowed its interactive style from LOGO. LOGO introduced a style of graphics called "turtle graphics" that is based on objects (called "turtles") that draw as they move around the screen. Smalltalk provides a similar class of objects called *pens.*

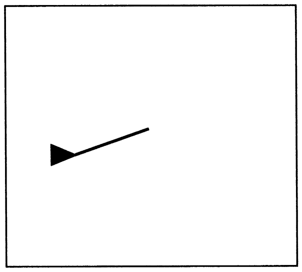
In our first example, we will investigate the behavior of a pen named Scribe and see how it can draw on the screen. The display shows us the position of Scribe.



Let's assume that Scribe is at position (500,500), the center of the screen; this is written 500@500 in Smalltalk. To draw a line from coordinates (500,500) to (200,400), we enter

```
Scribe goto: 200@400
```

This is a message to Scribe that tells it to draw a line from where it is to (200,400). The result is



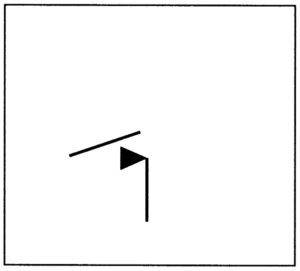
What if we want to move Scribe without drawing a line? We first send the message

```
Scribe penup
```

which tells `Scribe` to stop drawing. We then move `Scribe` to the desired point. For example, to draw a vertical line from (500,100) to (500,400), we can enter

```
Scribe penup.
Scribe goto: 500@100.
Scribe pendn.
Scribe goto: 500@400
```

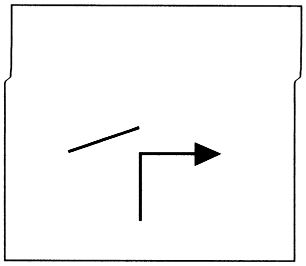
The result is



`Scribe` also has an orientation, so that if we tell it to `go`, it will go a specified distance in that direction. Suppose the direction of `Scribe` is to the right, then

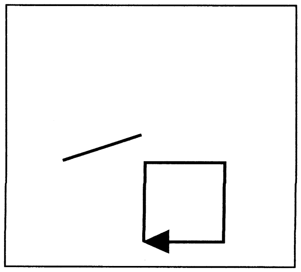
```
Scribe go: 300
```

has this effect



We can complete the square by turning and drawing the remaining lines as follows:

```
Scribe turn: 90.
Scribe go: 300.
Scribe turn: 90.
Scribe go: 300
```



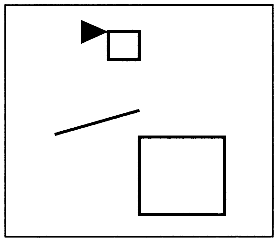
A more convenient way of drawing the square is to use a control structure. To draw a square whose upper-left-hand corner is at (400,800) with a side of length 100, we can write

```
Scribe penup; goto: 400@800; pendn; turn 180.
4 timesRepeat: [Scribe go: 100. Scribe turn: 90]
```

Notice that in the first line we used an abbreviation, a *cascaded message*, to avoid writing `Scribe` over and over again. Also notice the extent to which Smalltalk obeys the Regularity Principle: Even control structures are accomplished by sending messages to objects. In this case, the object `4` has been sent the message

```
timesRepeat: [Scribe go: 100. Scribe turn: 90]
```

The object `4` responds to this message by repeating the bracketed expressions four times. The result of these commands is



Let's summarize several important ideas in Smalltalk programming:

1. Objects have a behavior.
2. Objects can be made to do things by sending them messages.
3. Repetitive operations can be simplified by using control structures.

Objects Can Be Instantiated

Having learned to use existing objects, the next step for the Smalltalk programmer is to learn to create new objects. Continuing our previous example, suppose that we want another pen and that we want to call it `anotherScribe`. We cannot just ask for a new object; we have to say what kind of object we want. Note that a *class* is just a name for a particular kind of object; for example, the class `Integer` includes the objects `0`, `1`, `2`, `3`, ... Furthermore, every object is an *instance* of some class, so when we create an object, we have to indicate the class of which we want it to be an instance. Thus, the process of creating an object is called *instantiation.*

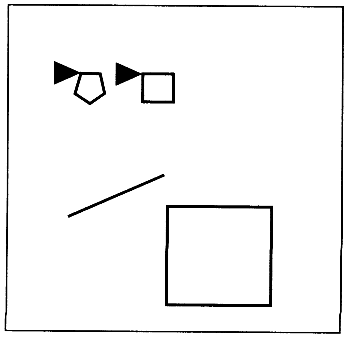
Recall that every request for action in Smalltalk must be accomplished by sending a message to some object. This includes instantiantion. The question is: Which object should be sent the message requesting the instantiation of a new pen? This message could be directed to some universal system object responsible for all instantiations, but we will be more in accord with the Information Hiding Principle if we make the class `pen` itself responsible for instantiating pens. This is the approach used in Smalltalk. To create a new pen and call it anotherScribe, we enter

```
anotherScribe ← pen newAt: 200@800
```

This sends the message newAt: 200@800 to the class pen, which creates and returns a new pen located at coordinates (200,800) and pointing to the right. The name another-Scribe is bound to this new object; hence, we can direct messages to the new object by using this name:

```
anotherScribe pendn.
5 timesRepeat: [anotherScribe go: 50; turn: 72]
```

The result will be



Exercise 12-1: What Smalltalk commands will create a new pen called Writer and cause it to draw an equilateral triangle with its apex at (800,800).

Classes Can Be Defined

In our previous example, we sent messages to a pen that caused it to draw a box on the screen. If we wanted to draw another box, we would have to move the pen (or instantiate a new one) and send it the same messages over again. This would violate the Abstraction Principle. A better solution is to define a class box that can be instantiated any number of times. Smalltalk allows us to do this, thus supporting both the Abstraction and Regularity Principles (the latter because user-defined classes are just like built-in classes). To see an example, look at Figure 12.2, which shows a definition of the box class as it would appear on the Dynabook screen.

Notice that a class definition has a two-dimensional arrangement in a window; this is quite different from the one-dimensional syntax associated with the other languages we have studied. Smalltalk obeys the Regularity Principle by adopting the window as a uniform way of interacting with users and organizing the screen.

Look at the definition of box in Figure 12.2. The first line is self-explanatory: It names the class. The second line lists four *instance variables*. These variables are local to each instance of the class (i.e., to each box) and are instantiated for each box. In other words, each

class name	box
instance variable names	loc tilt size scribe
instance messages and methods	

```
shape | |
   scribe penup; goto: loc; turnTo: tilt; pendn.
   4 timesRepeat: [scribe go: size; turn: 90]

show | |
   scribe color ink.
   self shape

erase | |
   scribe color background.
   self shape

grow: amount | |
   self erase.
   size ← size+amount.
   self show
```

Figure 12.2 Definition of Box Class

box has its own `loc`, `tilt`, `size`, and `scribe`. The purpose of these variables is as follows: `loc` contains the location of the upper-left-hand corner of the `box`; `tilt` contains the angle describing the orientation of the `box`; `size` is the length of the `box`'s side; and `scribe` is the `pen` used by the `box` to draw its shape.

The rest of the window is devoted to the *instance methods*; these are the methods that describe the behavior of the *instances* when they receive various messages (we will see below that there are also class messages that describe the behavior of *classes* when they receive messages). Suppose `B1` is a `box`; then sending it the message `shape`:

```
B1 shape
```

causes it to (1) lift its `pen`, (2) go to the specified location, (3) turn to the specified angle, and (4) draw a `box` of the specified size.

We can see that by changing the color of the `pen`'s `ink`, the `shape` method can be used either to make the `box` appear or to make it disappear. This is the purpose of the `show` and `erase` messages. Notice that a `box` responds to each of these messages by changing the `pen` color and sending itself the `shape` message. (The instance variable `self` is implicitly bound to the instance to which it is local.)

Let's look at the '`grow`' method; `B1` responds to `B1 grow: 20` by increasing its size by 20 units. It accomplishes this by erasing itself, increasing its `size` variable, and redrawing itself. Notice the formal parameter `amount` to the '`grow`' method. Moving a box would be accomplished in a similar manner.

▪ ***Exercise 12-2:*** Define the '`move`' method so that `B1 move: 100@200` moves box `B1` to location (100,200).

▪ ***Exercise 12-3:*** Define the '`turn`' method so that `B1 turn: 45` causes box `B1` to turn 45 degrees.

class name	box
instance variable names	loc tilt size scribe
class messages and methods	

```
newAt: initialLocation | newBox |
   newBox ← self new.
   newBox setLoc: initialLocation  tilt: 0  size: 100  scribe: pen new.
   newBox show.
   ⇑ newBox
```

instance messages and methods	

```
setLoc: newLoc  tilt: newTilt size: newSize  scribe: newScribe | |
   loc ← newLoc. tilt ← newTilt.
   size ← newSize. scribe ← newScribe

shape | |
   scribe penup; goto: loc; turnTo: tilt; pendn.
```

Figure 12.3 Class Method in Box Class

Classes Can Also Respond to Messages

We have left a very important part of the definition of boxes undone: the method for instantiation. As we said before, the class itself is responsible for creating new instances of itself, so this kind of method is called a *class method.* In Figure 12.3 we have scrolled up the window containing the box class to show its instantiation method. Thus, when we execute

```
B2 ←box newAt: 300@200
```

the message `newAt: 300@200` will be sent to the class box. This class method begins by sending itself (i.e., the class box) the message new. All classes automatically respond to new by creating a new, uninitialized instance of themselves. This new instance is stored in the local temporary variable newBox. The instance variables are initialized with a new instance method that has the template `setLoc: tilt: size: scribe`. The `newAt:` method then orders the instance to show itself and finally returns the instance so that it can be bound to B2.

▪ ***Exercise 12-4:*** Alter the instantiation method for boxes so that they are created with a tilt of 45° and a size of 200.

▪ ***Exercise 12-5:*** Write an additional instantiation method for boxes, called `newSize:`, that creates a box of the given size, but always at the center of the screen [i.e., (500,500)].

12.3 DESIGN: CLASSES AND SUBCLASSES

Smalltalk Objects Model Real-World Objects

In our discussion of LISP (Chapter 9, Section 9.3), we saw that atoms are often used to model objects in the real world. That is, like real-world objects, atoms in LISP can have properties and be related in various ways. This allows many programs to be viewed as a *model* or *simulation* of some aspects of the real world.

We have mentioned that many of the ideas in Smalltalk derive from Simula, an extension of Algol intended for simulation. These ideas include internally represented objects as instances of classes (internally represented objects are discussed in Chapter 7, Section 7.4). It is thus no surprise that Smalltalk objects are well suited to modeling real-world objects. In particular, the data values inside an object can represent the properties and relations in which that object participates, and the behavior of the Smalltalk object can model the behavior of the corresponding real-world object. Therefore, in Smalltalk, the dominant *paradigm* (or model) of programming is *simulation.*

Classes Group Related Objects

In the real world, all of the objects that we observe are *individuals*; every object differs from every other in a number of ways. However, if we are to be able to deal effectively with the world, we must be able to understand why objects act the way they do so that we will be able to predict their actions in the future. If every object were completely different from every other one and every action and effect were unique, then there would be no possibility of understanding the world or acting effectively. This is not the case, however; objects have many properties in common and we observe broad classes of objects acting similarly. Therefore, we focus on these common properties and behaviors and *abstract* them out. This, of course, is just an application of the Abstraction Principle. The resulting *abstraction*, or *class*, retains the similar properties and behaviors but omits the *particulars* that distinguish one individual from another.

This is exactly the situation we find in Smalltalk. The *class definition* specifies all of the properties and behaviors common to all instances of the class, while the *instance variables* in the object contain all of the particular information that distinguishes one object from another. The behavior of the members of a class, the set of all messages to which the members of that class respond, is called the *protocol of the class.* The protocol is determined by the instance methods in the class definition.

This approach is ideally suited to simulation. When we model some aspect of the real world, we are trying to find out what would happen if certain conditions held. To do this it is necessary to have laws of cause and effect that describe how certain kinds of objects act in certain situations. In other words, it is necessary to know the relevant behavior of certain *classes* of objects. It is exactly this information that is modeled by a Smalltalk *class:* the behavior common to all of the instances of the class.

Subclasses Permit Hierarchical Classification

We have seen how the Abstraction Principle can be applied to individual objects: The properties common to a group of objects are abstracted out and associated with the class of those objects. The Abstraction Principle can also be applied to classes themselves. Consider the class `pen`, which abstracts out the common properties of objects like `Scribe` and `anotherScribe`. We have also seen the class `box`, which abstracts the common properties of boxes. Similarly, the class `window` might include all of the window objects that can be displayed on the screen, such as the window that contains the definition of `box`. Notice that

class name	displayObject
instance variable names	loc
instance messages and methods	

```
goto: newLoc ||
  self erase.
  loc ← newLoc.
  self show
```

Figure 12.4 Example of Superclass DisplayObject

all of these classes have certain properties in common, for example, coordinates that determine their positions on the screen. We can presume that they also respond to certain common messages, such as to alter their position and to appear and disappear. The Abstraction Principle tells us that we should abstract out these common properties and give them a name such as `displayObject`. Then, `pen`, `box`, and `window` will be *subclasses* of the class `displayObject`. Conversely, `displayObject` would be called a *superclass* of the classes `pen`, `box`, and `window`. Figure 12.4 shows part of the definition of `display Object` and Figure 12.5 shows how `box` can be made a subclass of `displayObject`.

Notice that in the method for `shape` in the definition of `box`, we make use of the variable `loc`, which is defined in the superclass. That is, the instance variables of the superclass are *inherited* by all of its subclasses. Similarly, any methods defined in the superclass (such as `goto:`) are also inherited.

Smalltalk would violate the Zero-One-Infinity Principle of it allowed only two levels of classes, that is, superclasses and subclasses, but there is no such limitation. Classes can always be *refined* by defining subclasses within them. Conversely, classes can be grouped into more inclusive superclasses. Indeed, there is one grand superclass, called `object`, that includes all other classes as its subclasses. Therefore, all objects in the Smalltalk system are instances (perhaps indirectly) of the class `object`. The end result of all of this is a unified hierarchical classification of all objects known to Smalltalk. We can visualize this as shown in Figure 12.6. At the lowest level we have individual objects, and at all levels above them we have classes of objects. Each class is a subclass of all of the classes above it, and each class is a superclass of all of the classes below it. The actual Smalltalk class hierarchy is much richer than suggested above; Table 12.1 shows part of the Smalltalk-80 hierarchy of system (i.e., built-in) classes.

class name	box
superclass	displayObject
instance variable names	tilt size scribe
instance messages and methods	

```
shape | |
  scribe penup; goto: loc; turnTo: tilt; pendn.
  4 timesRepeat: [scribe go: size; turn: 90]
```

Figure 12.5 Example of Subclass Box

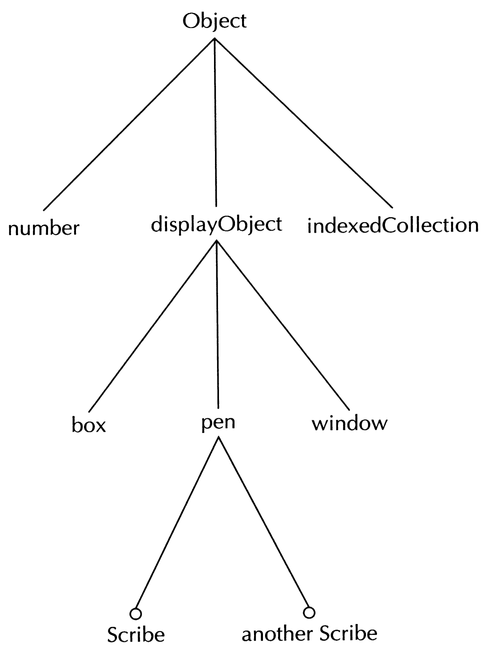


Figure 12.6 Example of Hierarchical Classification of Objects

Behavior Can Be Extended or Modified

As the previous example indicates, subclasses allow the behavior of a class to be extended. For example, boxes respond to messages in addition to those to which displayObjects respond. Thus, the behavior of boxes is an *extension* of the behavior of displayObjects. The subclass mechanism facilitates building classes on already existing classes. In this case, the goto: method, once written, can be used by any new class that is made a subclass of displayObject; it is never necessary to rewrite this method.

Subclasses can build upon the behavior of their superclasses; they can also modify it. Recall that in block-structured languages, a declaration of an identifier overrides any declarations of that same identifier in surrounding environments. The same rule applies to methods: A definition of a method in a subclass overrides any definitions of that method that may exist in its superclasses. Thus, if the standard definition of goto: in displayObject were inappropriate for windows, then this method could be redefined in window as shown:

class name	window
superclass	displayObject
instance variable names	size scribe text
instance messages and methods	
goto: loc || ... *new definition of* goto: ...	

Is the old definition of goto: inaccessible to window? No, it is not. Recall that an object can send a message to itself with an expression like self goto:loc. It is also pos-

TABLE 12.1 Smalltalk-80 System Class Hierarchy (Partial)

```
Object
  I. Magnitude
     A. Character
     B. Date
     C. Time
     D. Number
        1. Float
        2. Fraction
        3. Integer
           a. LargeNegativeInteger
           b. LargePositiveInteger
           c. SmallInteger
 II. Collection
     A. SequenceableCollection
        1. LinkedList
        2. ArrayedCollection
           a. Array
           b. Bitmap
           c. RunArray
           d. String
           e. Text
           f. ByteArray
        3. Interval
        4. OrderedCollection
     B. Bag
     C. MappedCollection
     D. Set
III. DisplayObject
     A. DisplayMedium
        1. Form
           a. Cursor
           b. DisplayScreen
     B. InfiniteForm
     C. OpaqueForm
     D. Path
        1. Arc
        2. Curve
           a. Circle
        3. Line
        4. LinearFit
        5. Spline
 IV. Behavior
  V. BitBlt
     A. Pen
```

sible for an object to send a message to itself in its capacity *as a member of its superclass.* For example, if the `goto:` method in `window` needed to make use of the `goto:` method in `displayObject`, we could write

```
goto: loc | |
   .
   .
   .
   super goto: loc
   .
   .
   .
```

The name `super` is automatically bound to the same object as `self`, but the object is considered as a member of its superclass. If we had written `self goto: loc` instead, we would have caused a recursive invocation of the `goto:` method for `windows`.

Thus, although definitions in a subclass cover up definitions in the superclass, the superclass definitions can be uncovered by an explicit request. This simplifies building new software on already existing software and makes Smalltalk a very extensible system.

Overloading Is Implicit and Inexpensive

Since objects are responsible for responding to messages, there is no reason why objects of different classes can't respond to the same messages. For example, in the expression `3 +`

5 the object 3 responds to the message +5 by returning the object 8. Similarly, in the expression "book" + "keeper" the object "book" responds to the message + "keeper" by returning the object "bookkeeper". In effect '+' has been overloaded to work on both numbers and strings. Notice, however, that there is no operator identification problem because the system always knows the class to which an object belongs, and this class determines the method that will handle the message. Thus, operators can be automatically extended to handle new data types by including methods for these operators in the class defining the data type.

Classes Allow Multiple Representations of Data Types

Classes also simplify having several concrete representations for one abstract data type. For example, we can define a class indexedStack that implements stacks as arrays. Objects that are instances of indexStack respond to messages such as pop, push: *x*, and empty. Similarly, we can define a class linkedStack that implements stacks as linked lists and that responds to the same messages. Each of these classes is a concrete representation of the same abstract type, stack. The key point is that any program that works on linked-Stacks will also work on indexedStacks, because they respond to the same messages.

Thus, in accordance with the Information Hiding Principle, we have hidden the implementation details in the class. Also notice that classes obey the Manifest Interface Principle since the interface to an object is just its *protocol*: the set of messages to which it responds. Therefore, objects with the same protocol (interface) can be used interchangeably.

Objects Can Be Self-Displaying

The problem of displaying objects demonstrates the extensibility and maintenance advantages of the Smalltalk class structure. Suppose that we define a new class complex that represents complex numbers. It is desirable that we be able to print out complex numbers, *just as we can* print out real numbers and integers. The problem is that in most languages it would be necessary to modify the print procedure to accomplish this; a new case would have to be added to handle complex numbers. This violates the Information Hiding Principle since it forces us to scatter the implementation details of complex numbers around the system.

Smalltalk has a simpler solution: We just require that every displayable object respond to the message print by returning a character string form of itself. For example, we could define complex numbers as shown in Figure 12.7. Then, if w were the object representing 1 + 2i and z were the object representing 2 + 5i, sending the message print to the object returned by w + z,

```
(w + z) print
```

would return the string.

```
"3 + 7i"
```

Similarly, we could define the print method for boxes to show their location, tilt, and size:

class name	complex
instance variable names	realPt imagPt
instance messages and methods	

```
print | |
  ⇑realPt print + "+" + imagPt print + "i".

+ y | |
```

Figure 12.7 Print Method for Complex Numbers

```
print | |
  ⇑ "box at " + loc print
    + ", size=" + size print
    + ", tilt=" + tilt print.
```

Thus, if we entered `B1 print`, we would see

```
box at 100@500, size=100, tilt=0
```

We are, of course, assuming that `points` respond to `print` with a string of the form `"`*x*`@`*y*`"`.

Methods Accept Any Object with the Proper Protocol

The fact that a method will work on any object with the appropriate protocol is one of the factors that make Smalltalk such a flexible, extensible system. It is possible at any time to define a new class and have many existing methods already applicable to it. For example, we might have a number of methods that operate on numbers by using the arithmetic operators. These methods might include expressions such as `(x+y)*x`. If we later defined a class `polynomial` that responded to messages `+y`, `*y`, and so on, then polynomial objects could be passed to these methods. This works because in the expression `x+y` the object `x` is responsible for responding to the message `+y`; if `x` is a number, it does simple addition; if `x` is a polynomial, it does polynomial addition. The method containing `x+y` does not have to know whether `x` and `y` are real numbers, polynomials, complex numbers, or any other objects that respond to the arithmetic operators.

Operators Are Handled Asymmetrically

Unfortunately, this interpretation of an operator expression (e.g., `3 + 5`) as sending a message (`+ 5`) to the operator's first argument (`3`) creates an awkward asymmetry. To see why, suppose we wanted to define string catenation to coerce integers to strings. Thus, if the variable `pageNo` referred to the integer 5, we would expect `"page" + pageNo` to return the string `"page 5"`. The + method for strings could accomplish this by sending a message such as `print` to the method's argument. This all works as we would hope. However, by the Syntactic Consistency Principle, we would also expect coercion to take place in the first argument, so that if we wrote `pageNo + " pages"` we would get the string `"5 pages"`. However, for this to take place, it is necessary for the + method *of integers* to recognize that its second argument is a string, and therefore to convert itself to a string.

This is certainly doable, but it results in a breakdown in modularity; not only must strings be aware that they may be added to integers, but also integers must be aware they can be added to strings. If we further extend + to polynomials, then not only must polynomials be aware they can be added to integers (and reals), but integers (and reals) must be aware they can be added to polynomials. This situation makes it difficult to take a class (such as `integer`) as given and then extend the system by building other classes (e.g., `string`, `polynomial`) upon it. Some newer object-oriented languages, which use an *external representation* of objects (recall p. 276), avoid this problem, although there is still a multiplicative number of type interactions (see Sections 12.4 and 12.6). (What we actually have here is a lack of orthogonality, since the definition of addition should be independent of the integer-to-string coercion.

Hierarchical Subclasses Preclude Orthogonal Classifications

We will now discuss one of the limitations of a strictly hierarchical subclass–superclass relationship. Consider an application in which Smalltalk is being used for the computer-aided design of cars. We will assume that this system assists in the design in several different ways. For example, it helps in producing engineering drawings by allowing the user to manipulate and combine diagrams of various parts, and it assists in cost and weight estimates by keeping track of the number, cost, and weight of the parts.

Presumably each part that goes into a car is an object with a number of attributes. For example, a bumper might have a weight, a cost, physical dimensions, the name of a manufacturer, a location on the screen, and an indication of its points of connection with other parts. Similarly, an engine might have weight, cost, dimensions, manufacturer, horsepower, and fuel consumption. If we presume that our display shows only the external appearance of the car, then an engine will not have a display location.

The next step is to apply the Abstraction Principle and to begin to classify the objects on the basis of their common attributes. For example, we will find that many of the classes (e.g., bumpers, roofs, grills) will have a `loc` attribute because they will be displayed on the screen. We can also presume that these objects respond to the `goto:` message so that they can be moved on the screen. This suggests that these classes should be made subclasses of `displayObject` because this class defines the methods for handling displayed objects. Thus, our (partial) class structure might look something like that in Figure 12.8. On the other hand, many of the objects that our program manipulates have `cost`, `weight`, and `manufacturer` attributes. This suggests that we should have a class called, for example, `inventoryItem` that has these attributes and that responds to messages for inventory control (e.g., `reportStock`, `reorder`). This leads to the class structure shown in Figure 12.9.

Now we can see the problem. Smalltalk organizes classes into a hierarchy; each class has exactly one immediate superclass. Notice that in our example several of the classes (e.g., `bumper` and `grill`) are subclasses of two classes: `displayObject` and `inventoryItem`. This is not possible in Smalltalk; when a class is defined, it can be specified to be an immediate subclass of exactly *one* other class.

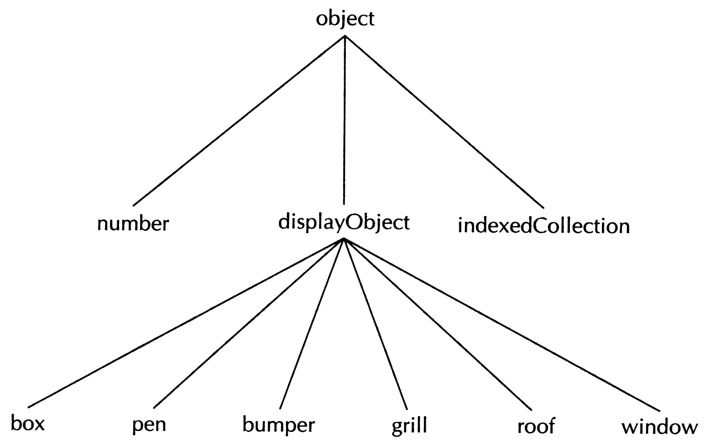


Figure 12.8 Example of `displayObject` Class Hierarchy

What are the consequences of this? We can choose either `displayObject` *or* `inventoryItem` to be the superclass of the other. Suppose we choose `displayObject`; then our class structure looks as shown in Figure 12.10. This seems to solve the problem. The display methods occur once—in `displayObject`—and the inventory control methods occur once—in `inventoryItem`. Unfortunately, this arrangement of the classes has a side effect: Some objects that are never displayed (e.g., engines and paint) now have the attributes of a displayed object, such as a display location. This means that they will respond to messages that are meaningless, which is a violation of the Security Principle. The alternative, placing `displayObject` under `inventoryItem`, is even worse since it means that objects such as `pen`s and `box`es will have attributes such as `weight`, `cost`, and `manufacturer`! Thus, we seem to be faced with a choice: either violate the Security Principle by making either `displayObject` or `inventoryItem` a subclass of the other or violate the Abstraction Principle by repeating in some of the classes the attributes of the others.

What is the source of this problem? In real life we often find that the same objects must be classified in several different ways. For example, a biologist might classify mammals as

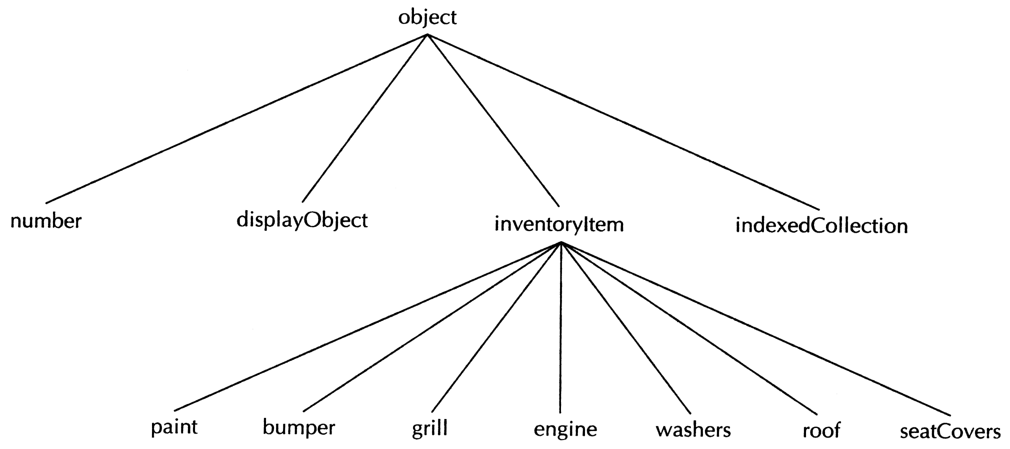


Figure 12.9 Example of `inventoryItem` Class Hierarchy

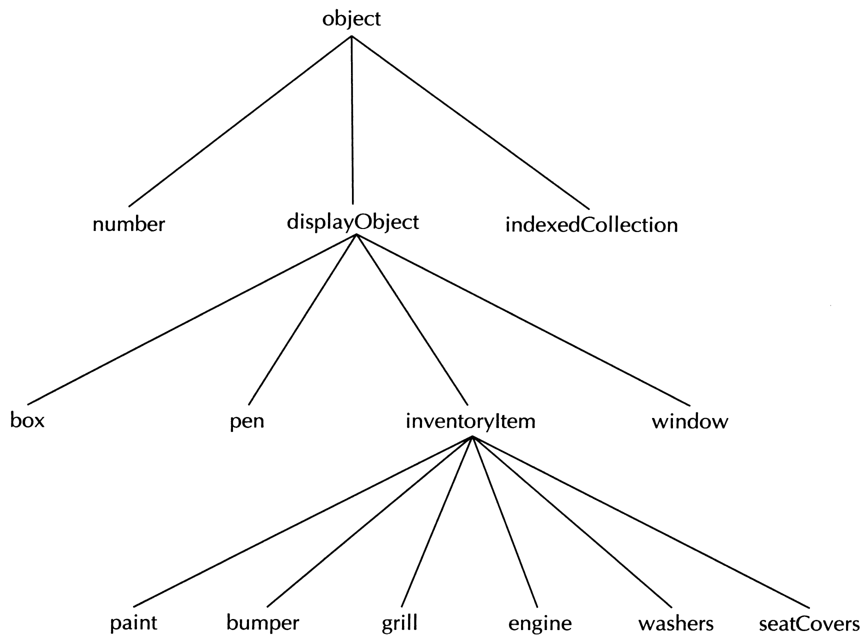


Figure 12.10 Limitations of Hierarchical Classification

primates, rodents, ruminants, and so forth. Someone interested in the uses of mammals might classify them as pets, beasts of burden, sources of food, pests, and so forth. Finally, a zoo might classify them as North American, South American, African, and so forth. These are three *orthogonal* classifications; each of the classes cuts across the others at "right angles" (see Figure 12.11). (We have shown only two of the three dimensions.)

In summary, a hierarchical classification system, such as provided by Smalltalk, precludes orthogonal classification. This in turn forces the programmer to violate either the Security Principle or the Abstraction Principle. In essence, Smalltalk ignores the fact that the appropriate classification of a group of objects depends on the context in which those objects are viewed.

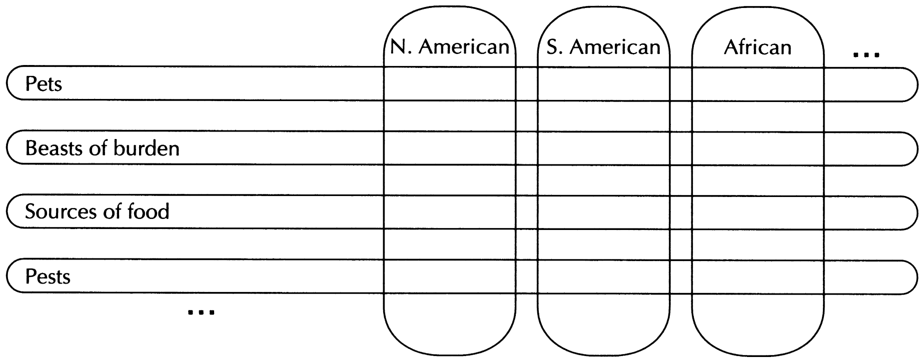


Figure 12.11 Example of Orthogonal Classification

~~Some experimental extensions of Smalltalk have attempted to remedy this problem by providing a "multiple inheritance" capability that allows an object to belong to several classes at once.~~

Multiple Inheritance Raises Difficult Problems

Some experimental extensions of Smalltalk have attempted to remedy this problem by providing a *multiple inheritance* capability, which allows a class to be an immediate subclass of more than one superclass. Multiple inheritance is also supported in the more-recent object-oriented extensions of some older languages, such as C++ and CLOS, the Common LISP Object System (see Section 12.6). However, multiple inheritance raises many issues, which are not easy to solve. For example, if an instance variable or method is inherited from two or more superclasses, do we select just one (and how do we select which), or do we get them all (and then how do we discriminate between them)? How do we handle a situation in which a class is a superclass in more than one way? For example, *B* may be declared a subclass of *A*, and *C* may be declared a subclass of both *A* and *B*. *C* may inherit instance variables and methods from *A* in two different ways, directly from *A* and indirectly via *B*. Expressed in this abstract way, it may seem a pathological situation, but there are good reasons for class relationships of this kind, and they can also occur as unintentionally in the object-oriented design of large software systems. It is not our purpose to address these issues here, but only to alert you of the design complexities facing you on both sides of the multiple inheritance question.

12.4 DESIGN: OBJECTS AND MESSAGE SENDING

The Primary Concept Is the Object

As we have seen, everything in Smalltalk is an object. This design decision satisfies both the Regularity and Simplicity Principles. In Smalltalk even classes are objects, that is, they are instances of the class named `class`. This fact reflects a design principle, which Kay has expressed, "take the hardest and most profound thing you need to do, make it great, and then build every easier thing out of it." It represents the conscious extrapolation of the ampliative aspects of a new technology. (This principle is reflected in all the fifth-generation paradigms.)

What exactly is an object? Objects have characteristics of both data and programs. For example, objects are the things that represent quantities, properties, and the entities modeled by the program. In Smalltalk numbers, strings, and so on, are all objects. Objects also have some of the characteristics of programs; like programs, they *do* things. For example, `Scribe` could be made to move, draw lines, change its `pen` status, and so on. In a conventional language, programs are *active* and data elements are *passive*; the program elements *act on* the data elements. In an object-oriented language like Smalltalk, the data elements are *active*; they respond to messages that cause them to act on themselves, perhaps modifying themselves, perhaps modifying or returning other objects. Another way of saying this is that conventional languages are *function oriented*; things are accomplished by passing objects to functions. In an object-oriented language, things are accomplished by sending messages to objects.

The set of messages to which an object can respond is called its *protocol*. The protocol is determined by the instance methods of the class of which it is an instance. When an object is sent a message, the Smalltalk system produces an error diagnostic if that message is not part of the object's protocol. This run-time checking is necessary to satisfy the Security Principle.

Names Are Not Typed

Most of the programming languages we have studied associate a definite type with each name (variable or formal parameter). Strong typing then ensures that only objects of the same type can be bound to these names. Smalltalk has a different strategy: No names are typed, and any object can be bound to any name. Type checking occurs when a message is sent to the object bound to a name. If the object responds to that message (i.e., the message is in its protocol), then the message is legal; otherwise it is not. Thus, Smalltalk performs *dynamic* type checking, like LISP, as opposed to *static* type checking, like Pascal and Ada.

Notice that dynamic typing does not imply *weak* typing. Strong typing refers to the fact that type abstractions are enforced. This enforcement may be done predominantly at compile-time (static typing) or predominantly at run-time (dynamic typing), so long as it is done. Smalltalk, like LISP, has strong, but dynamic, typing.

In previous chapters we have had a great deal to say about the security advantages of static type checking. Does this mean that Smalltalk violates the Security Principle? No, it does not. Since the Smalltalk system will allow a message to be sent to an object only if that object has a method to respond to the message, it is not possible for type violations to cause system crashes. Thus, the system and the user's program are secure.

Another reason for static type checking is that it allows earlier error detection. Specifically, it allows errors to be detected at compile-time, when the compiler can give an intelligible diagnostic, rather than at run-time, when only the machine code is available. This argument does not apply to a system like Smalltalk, however. Since all of the source code is available at run-time, the Smalltalk system can produce run-time diagnostics in terms of the source code of the program.

It might also be argued that early error detection saves both programmer and computer time since in a compiled language the entire program, or at least one module, must be recompiled each time an error is found. In Smalltalk, however, a run-time error causes the program to be suspended so that the programmer can investigate its cause. An offending class can be edited and quickly recompiled, and execution of the program can be resumed. Thus, in the Smalltalk environment, static type checking affords little savings in time.

Another argument for static typing is that it improves the efficiency of storage utilization since a character, for example, occupies less storage than a floating-point number. In Smalltalk, however, every object is accessed through an *object reference*, that is, a pointer. Therefore, all variables and parameters occupy the same amount of storage—one pointer.

A final argument for static typing is documentation. It is easier to understand a program if each variable and parameter has a type associated with it; this declares the use to which the programmer intends to put that variable or parameter. The designers of Smalltalk claim that well-named variables provide just as good documentation. For example, calling a parameter `anInteger` makes its intended value just as clear as a typed declaration like `n: integer`.

The last point in favor of Smalltalk's dynamic typing is *flexibility*. We have already seen how any object with the proper protocol can be passed to a method. This makes it much easier to build on existing software in the Smalltalk system. This can be considered a major application of the Abstraction Principle since it allows common algorithms to be factored out of a system without complicated mechanisms like Ada's generics.

▨ ***Exercise 12-6*:*** Write a short report on the pros and cons of static and dynamic typing. Evaluate each of the above arguments in favor of dynamic typing in Smalltalk. Formulate your own position on typing and defend it.

▨ ***Exercise 12-7*:*** Write a short report evaluating Ada's object-oriented typing extension.

Messages Are Essentially Procedure Invocations

Recall our discussion of the *internal* and *external* representation of objects in connection with Ada (p. 276). We can see that Smalltalk has *internal* objects since both the fields of the object, such as `size` and `loc`, and the procedures for manipulating the object, such as `goto:`, and `show`, are part of the object. In other words, there is not much difference between the Smalltalk expression

```
Scribe goto: loc
```

and the Ada statement

```
Scribe.goto(loc);
```

which invokes the `goto` procedure in the package `Scribe` with the argument `loc`.

Let's investigate this similarity in more detail. In Smalltalk, when a class is defined, we specify the variables that are to be duplicated in each instance of the class and the messages to which all instances of the class will respond. Thus, the class definition in Figure 12.2 is similar to the Ada generic package shown in Figure 12.12. Notice that each method has been translated into a public procedure and each instance variable into a private variable (of course, we have had to add type information, so the translation is not exact).

Just as in Smalltalk, in Ada the process of instantiation is separate from the process of class definition. The Smalltalk instantiations

```
B1 ← box newAt 200@500
B2 ← box newAt 800@500
```

are similar to the Ada package declarations

```
package B1 is new Box (Initial_Loc => (200,500));
package B2 is new Box (Initial_Loc => (800,500));
```

A major difference between objects in Ada and Smalltalk is that Smalltalk allows the *dynamic* instantiation of internally represented objects, whereas Ada does not. This can be

```
generic
   Initial_Loc : Point;
package Box is
   procedure Shape;
   procedure Show;
   procedure Erase;
   procedure Grow (Amount : Integer);
end Box;

package body Box is

   Loc : Point := Initial_Loc;
   Tilt : Float;
   Size : Float;
   Scribe : Pen;

   procedure Shape is
      .
      .
      .
   end Shape;
   .
   .
   .
end Box;
```

Figure 12.12 Generic Package Similar to a Class

seen in the above example; the instantiation of the packages B1 and B2 is accomplished by declarations, the number of which is limited by the written form of the program. In Smalltalk, instantiation is done at run-time, by sending an instantiation message to a class. Thus, Smalltalk is more flexible, since it is possible to decide at run-time how many instances of a class are required. In Ada this can be done only with *external* objects (Chapter 7, Section 7.4), which are more limited since they cannot respond directly to messages. We will see in Section 12.6 that Ada 95 has been extended to include more direct support for object-oriented programming.

▪ ***Exercise 12-8:*** Complete the definition of the Ada package Box.

▪ ***Exercise 12-9*:*** Discuss the trade-offs between statically and dynamically instantiated objects.

▪ ***Exercise 12-10*:*** Show that anything that can be accomplished with external objects can also be accomplished with internal objects, and vice versa.

There Are Three Forms of Message Template

You have seen that messages are essentially procedure invocations, although the formats allowed for messages are a little more flexible. In most languages parameters are surrounded by parentheses and separated by commas; in Smalltalk parameters are separated by keywords. For example, the Smalltalk message:

```
newBox setLoc: initialLocation tilt: 0 size: 100 scribe: pen new
```

is equivalent to the Ada procedure call:

```
NewBox.Set (Initial_Location, 0, 100, Pen.New);
```

although the similarity is more striking if we use position-independent parameters:

```
NewBox.Set (Loc => Initial_Location, Tilt => 0,
            Size => 100, Scribe => Pen.New);
```

Note, however, that Smalltalk is not following the Labeling Principle here since the parameters are required to be in the right order even though they are labeled.

The message format, keywords followed by colons, can be used if there are one or more parameters to the method. What if a method has no parameters? In this case, it would be confusing to both the human reader and the system if the keyword were followed by a colon. This leads to the format that we have seen for parameterless messages:

```
B1 show
```

Omitting the colon from a parameterless message is analogous to omitting the empty parentheses '( )' from a parameterless procedure call in Ada.

These message formats are adequate for all purposes since they handle any number of parameters from zero on up. Unfortunately, they would require writing arithmetic expressions in an uncommon way. For example, to compute $(x + 2) \times y$ we would have to write[3]

```
(x plus: 2) times: y
```

To avoid this unusual notation, Smalltalk has made a special exception: the arithmetic operators (and other special symbols) can be followed by exactly one parameter even though there is no colon. For example, in

```
x + 2 * y
```

the object named `x` is sent the message `+ 2`, and the object resulting from this is sent the message `* y`. Thus, this expression computes $(x + 2)y$; notice that Smalltalk does not obey the usual precedence rules (a concession to the Regularity Principle).

In summary, there are three formats for messages:

1. Keywords for parameterless messages (e.g., `B1 show`)
2. Operators for one-parameter messages (e.g., `x + y`)
3. Keywords with colons for messages with one or more parameters (e.g., `Scribe grow: 100`)

Notice that this format convention fits the Zero-One-Infinity Principle since the only special cases are for zero parameters and one parameter. However, the fact that these cases are handled differently from the general case violates the Regularity Principle. This is a conscious

[3] The parentheses are necessary, otherwise we would be sending to `x` a message with the template `plus:times:`.

trade-off that the designers of Smalltalk have made so that they can use the usual arithmetic operators. We know this because earlier versions of Smalltalk (e.g., Smalltalk-72) had a uniform method for passing parameters that did not depend on the number of parameters.

■ ***Exercise 12-11*:*** Discuss Smalltalk's message formats. Given that Smalltalk's conventions still violate the usual precedence rules for arithmetic operators, was it wise to make a special case of them? Either defend the conventions adopted by Smalltalk or propose and defend a different set of your own conventions.

Objects Hold the State of a Computation

How an object behaves when it receives a message depends on two things: the method defined for that message (which is part of the object's class and never changes) and the contents of the variables visible to the method. Since it is the instance variables that are different in different instances of a class, it is predominantly the instance variables that determine the individual behavior of objects. For example, the orientation of Scribe after it receives the message `turn: 90` depends on its orientation before it received that message, which is contained in an instance variable. There is no global table that holds the state of all of the instances of the class `pen`. Rather, each `pen` is responsible for keeping track of its own state and responding to messages appropriately.

In general, each object acts as an autonomous agent that is responsible for its own behavior but is not responsible for the behavior of any other objects. All of the information relevant to the individual behavior of an object is contained in that object; all of the information relevant to the similar behavior of a class of objects is contained in the class. Thus, the object-oriented style of programming supports the Information Hiding Principle in an essential way.

The Smalltalk Main Loop Is Written in Smalltalk

Like most interactive systems, Smalltalk is in a loop: Read a command, execute the command, print the result, and loop. This can be written in Smalltalk as follows:

```
true whileTrue: [Display put: user run]
```

This is an infinite loop that repeatedly executes the expression `Display put: user run`. The variable `user` contains an object called the *user task*; this object responds to the message `run` by reading an expression, evaluating it, and returning a string representing the result. The object `Display` represents the display screen and responds to the message `put:` by writing out the argument string.

The simplest user task is an instance of the class `userTask` shown here:

class name	userTask
instance messages and methods	

```
run ||
  Keyboard read eval print
```

`Keyboard` is the object responsible for the keyboard on which the user types; it responds to the message `read` by printing a prompt and returning a string containing the characters typed by the user. For example, if the user types `x + 2`, then the object returned by `keyboard read` is the string `″x + 2″`. Next, it is necessary that we know that strings respond to the message `eval` by calling the Smalltalk compiler and interpreter to evaluate themselves. Thus, assuming `x` is bound to `5`, the result of `″x + 2″ eval` will be the object `7`. We have already seen that objects respond to the message `print` by returning a character string representation of themselves, so the result of `7 print` is the string `″7″`. This is returned as the value of `user run` and hence printed on the user's display by the main loop of the Smalltalk system (as a result of executing `Display put: user run`).

Concurrency Is Easy to Implement

Recall that Smalltalk took many ideas from the simulation language Simula-67. Of course, in the real world many things happen simultaneously, or concurrently. Therefore, in a language intended to simulate aspects of the real world, it is important to be able to have several things going on at the same time. We have already seen an example of programming language support for concurrency: Ada's tasks. Message sending in Smalltalk is similar to the "rendezvous" in Ada. We will also see that the autonomous nature of Smalltalk's objects makes them ideal for concurrent programming.

How can we go about doing concurrent programming in Smalltalk? We saw how the main loop of the Smalltalk system ran the user task on every iteration. This suggests that we can do concurrent programming by having the main loop run each of a set of tasks. To do this we will assume that `sched` is the name of a `set` that contains all of the objects that are scheduled to be run concurrently. Therefore, we want to run each object `Task` in `sched`, that is, for each `Task` in `sched` we must evaluate `Task run`. By a task we mean anything that responds to the message `run`.

Although we could write a conventional loop for this, it is easier to use a variant of functional programming provided by Smalltalk. Specifically, a set *S* responds to *S* `map:` *B* by applying the *block B* to every element of the set. This is analogous to the LISP expression `(mapcar` *B S*`)` (see Chapter 10, Section 10.1). Finally, we need *B* to be a block that takes any object `Task` and sends it the message `run`:

```
[: Task | Task run ]
```

This is analogous to a lambda expression (Chapter 10, Section 10.1) in LISP: `(lambda (Task) (task run))`. Thus, we can define a class for scheduling concurrent tasks as follows:

class name	scheduler
instance messages and methods	

```
run | |
  sched  map: [ : Task | Task run]
```

Every time we send the `run` *message to a* `scheduler` *object, that object runs all the tasks in the* `sched` set.

class name	spinner
instance variables	theBox rate
class messages and methods	

newAt: initLoc rate: R | |
  theBox ← box newAt: initLoc.
  rate ← R.
  theBox show.
  sched add: self	

instance messages and methods	
run | | theBox tilt: rate	

Figure 12.13 Example of a Concurrent Class

To request that an object `T` be run concurrently with the other objects, it is only necessary to add it to `sched`:

```
sched add: T
```

Similarly, an object can be terminated (or temporarily suspended) by deleting it from `sched`. This is just a very simple form of first-in—first-out scheduling; more sophisticated scheduling strategies can be implemented by keeping the objects in an ordered list and reordering the list according to some priority system.

▩ ***Exercise 12-12:*** Figure 12.13 shows a simple example of concurrency—a class called `spinner`. Explain what the user would see on the screen after the following two expressions are entered:

```
spinner newAt: 500@200 rate: 1.
spinner newAt: 500@600 rate: 3
```

12.5 IMPLEMENTATION: CLASSES AND OBJECTS

Overview of the Smalltalk-80 System

In this section we will discuss several important issues in the implementation of Smalltalk. The first is portability: Most of the Smalltalk system is written in Smalltalk. This includes the compiler, decompiler, debugger, editors, and file system, which accounts for approximately 97% of the code of the Smalltalk-80 system. A major reason that Smalltalk can be programmed in Smalltalk is that most of the implementation data structures, such as activation records, are Smalltalk objects. This means that they can be manipulated by Smalltalk programs and have the properties of Smalltalk objects (e.g., they are self-displaying).

The part of Smalltalk that is not portable is called the Smalltalk-80 Virtual Machine; its size is between 6 and 12 kilobytes of assembly code. The Smalltalk designers claim that it requires about one man-year to produce a fully debugged version of the Smalltalk Virtual Machine, which makes Smalltalk an excellent example of the Portability Principle.

The Smalltalk Virtual Machine has three major components:

- Storage Manager
- Interpreter
- Primitive Subroutines

The Storage Manager is the abstract data type manager for objects. As required by the Information Hiding Principle, it encapsulates the representation of objects and the organization of memory. The only operations that other modules can perform on objects are those provided by the Storage Manager:

- Fetch the class of an object
- Fetch and store the fields of objects
- Create new objects

Of course, if the Storage Manager is to be able to create new objects, it must be able to get free space in which to put them. Thus, another responsibility of the Storage Manager is collecting and managing free space. For this purpose it uses a reference counting strategy with extensions for cyclic structures (see Chapter 11, Section 11.2).

The Interpreter is the heart of the Smalltalk system. Although it would be possible to interpret the written form of Smalltalk directly, it is more efficient if the interpreter operates on an intermediate form of the program. Recall that we did essentially this with our symbolic pseudo-code interpreter in Chapter 1: We translated a more human-oriented written form into the internal numeric codes required by the interpreter. The operations of this intermediate language are essentially the operations of the Smalltalk Virtual Machine. In other words, the interpreter is essentially the abstract data type manager for methods.

The last component of the Smalltalk Virtual Machine is the Primitive Subroutines package. This is just a collection of the methods that, for performance reasons, are implemented in machine code rather than Smalltalk. They include basic input-output functions, integer arithmetic, subscripting of indexable objects (e.g., arrays), and basic screen graphics operations.

There are three central ideas in Smalltalk: objects, classes, and message sending. We will now investigate the implementation of each of these.

Object Representation

Much of Smalltalk's implementation can be derived by application of the Abstraction and Information Hiding Principles. For example, the representation of an object must contain just that information that varies from object to object; information that is the same over a class of objects is stored in the representation of that class. What is the information that varies between the instances of a class? It is just the *instance variables*. The information stored with the class includes the class methods and instance methods.

Notice, however, that we will not be able to access the information stored with the class of an object unless we know what the class of that object is. Therefore, the representation of an object must contain some indication of the class to which the object belongs. There are many ways to do this, and several have been used by the various Smalltalk implementations. The simplest is to include in the representation of the object a pointer to the data structure representing the class.

Let's consider the example shown in Figure 12.14, which shows the representation of two boxes, `B1` and `B2`. To keep the figure clear, we have abbreviated or omitted some of the component objects. For example, the representation of the object `500@200` is shown, but the representation of `500@600` is abbreviated. Also, we have shown the names of the instance variables, although there is no reason actually to store them in the representation of objects. Finally, the representation of class objects is omitted because this topic is discussed next.

Notice that in addition to the instance variables and class description (`c.d.`), each object has a length field (`len`). This is required by the storage manager for allocating, deallocating, and moving objects in storage.

▩ ***Exercise 12-13:*** Considerable overhead is associated with the class description (`c.d.`) and length fields in the representation of objects. For example, 50% of the space required for a `point` object is used for the length and class description fields. An alternative to this is to divide memory into a number of *zones* with each zone being dedicated to holding the instances of just one class. Then, from the address of an object the storage manager can tell the zone it is in and, hence, both its class and length. This effectively encodes the class and length information as part of the address of an object. Analyze this implementation technique in detail. Describe the trade-offs between this technique and that described in the text, and discuss the advantages and disadvantages of the two techniques.

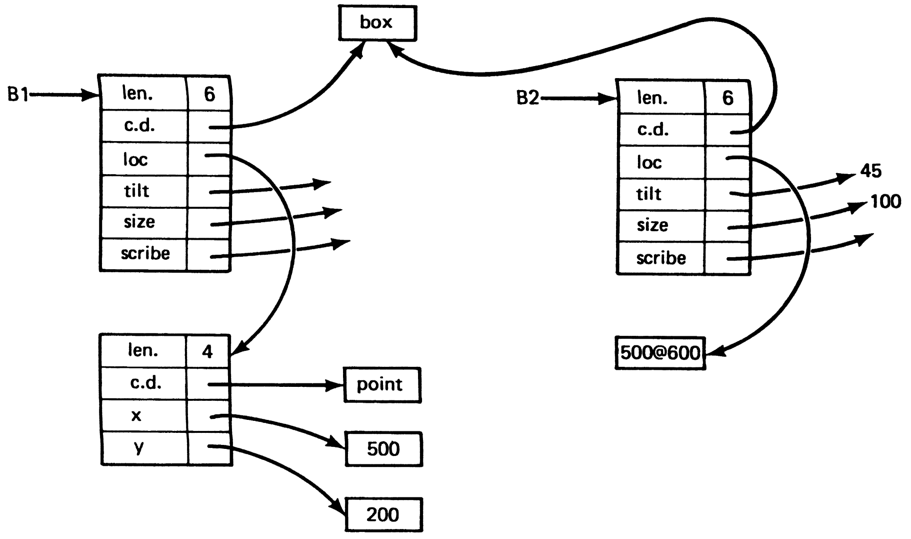


Figure 12.14 Representation of Objects

Class Representation

We have said that everything in Smalltalk is an object. This rule is true without exception (Regularity Principle). In particular it includes classes, which are just instances of the class named `class`. Therefore, classes are represented like the objects just described, with length and class description fields (the latter pointing to the class called `class`). The instance variables of a class object contain pointers to objects representing the information that is the same for all instances of the class.

What is this information? We can get a clear idea by looking at a class definition, such as the one in Figure 12.5. The information includes the following:

1. The class name
2. The superclass (which is `object`, the class of all objects, if no other is specified)
3. The instance variable names
4. The class message dictionary
5. The instance message dictionary

(Just as there are instance methods and class methods, Smalltalk allows both *instance variables* and *class variables.* We will ignore the latter, beyond mentioning that they are stored in the class object.)

Thus, observations of class definitions lead to a representation like that shown in Figure 12.15. (We have written the class of an object above the rectangle representing that object.)

Notice that we have added a field `inst. size` (instance size), which indicates the number of instance variables. The number of instance variables is needed by the storage manager when it instantiates an object since this number determines the amount of storage required. If we did not have this field, it would be necessary to count the names in the string contained in the `inst. vars.` field to determine this information, which would slow down object instantiation too much.

This leaves the message dictionaries for our consideration. In Smalltalk, a method is identified by the keywords that appear in a message invoking that method. For example, in

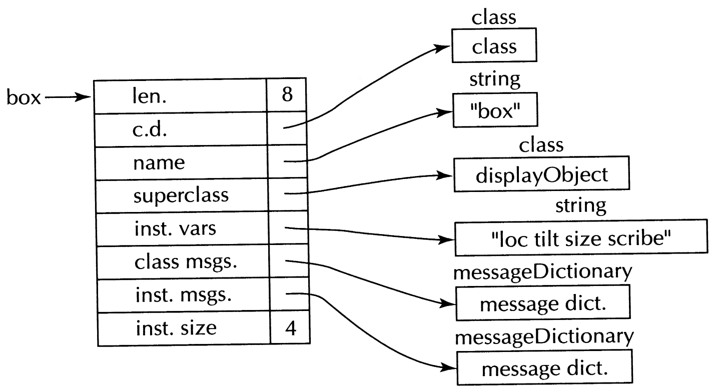


Figure 12.15 Representation of Class Object

`Scribe go: 100` the keyword `go:` identifies the method. For multiparameter messages, such as

```
spinner newAt: 500@200 rate: 1
```

the method is determined by the concatenation of the keywords: `newAt:rate:`. This is called the *message template*. Thus, for each message template acceptable to a class or its instances, one of the message dictionaries must contain an entry specifying a method for that message. The entry can be found in the message dictionary by using hashing techniques.

How should methods be represented? It would be much too slow if the interpreter had to decode the source form of a method every time the method was executed. Therefore, it is a good idea to compile the methods into some form of pseudo-code that can be rapidly interpreted. On the other hand, the source form of methods is needed for editing and displaying class definitions. Therefore, the message dictionaries contain two entries for each message template: one containing the source form of the method and one containing a compiled form. Figure 12.16 shows the representation of a message dictionary.

▩ ***Exercise 12-14:*** Why do you suppose Smalltalk provides class variables? What sort of information might it make more sense to store with a class than with the objects belonging to the class (i.e., in their class variables)?

Activation Record Representation

We will now investigate the implementation of message sending, the last of the three central ideas in Smalltalk. We have seen that there is a strong resemblance between message sending in Smalltalk and procedure calls in other languages. Thus, it should not be surpris-

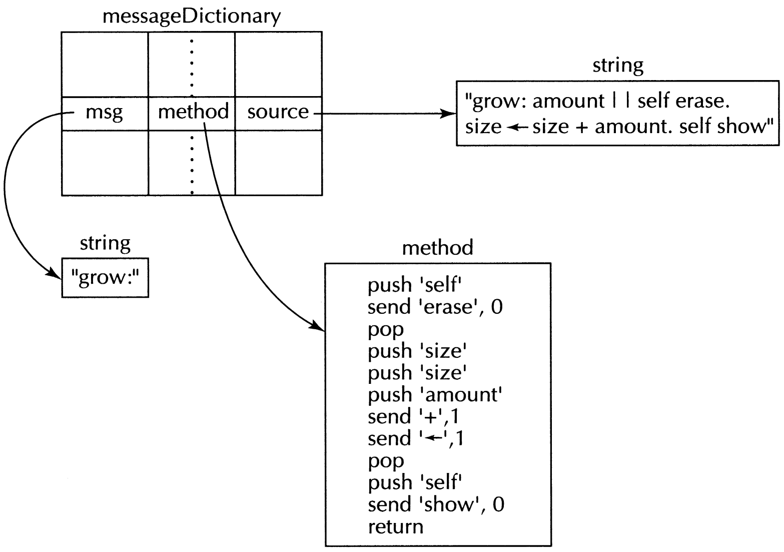


Figure 12.16 Example of a Message Dictionary and Compiled Code

ing that very similar implementation techniques are used, although there are some important differences.

In Chapters 2 and 6, we saw that *activation records* are the primary vehicle for procedure implementation. Since an activation record holds all of the information that pertains to one activation of a procedure, the processes of procedure call and return can be understood as the manipulation of activation records. The same is the case in Smalltalk: We will use activation records to hold all of the information relevant to one activation of a method.

What is the structure of an activation record? In Chapter 6 (Section 6.2), we saw that activation records for block-structured languages have three major parts. Smalltalk activation records follow the same pattern:

- *Environment part:* The context to be used for execution of the method
- *Instruction part:* The instruction to be executed when this method is resumed
- *Sender part:* The activation record of the method that sent the message invoking this method

We will consider these parts in reverse order.

The *sender part* is a *dynamic link*, that is, a pointer from the receiver's activation record back to the sender's activation record. It is just an object reference since activation records, like everything else in Smalltalk, are objects.

The *instruction part* must designate a particular instruction in a particular method. Since methods are themselves objects (instances of class `method`), a two-coordinate system is used for identifying instructions:

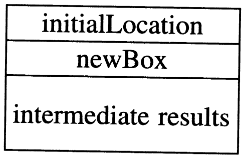
- An *object pointer* defines the `method`-object containing all of the instructions of a method.
- A *relative offset* identifies the particular instruction within the `method`-object.

This two-coordinate addressing is necessary because instruction addressing goes through the storage manager (thus adhering to the Information Hiding Principle).

The final part of the activation record is the *environment part*, which must provide access to both the local and nonlocal environments. The local environment includes space for the parameters to the method and the temporary variables. This part of the activation record also must provide space for hidden temporary variables such as the intermediate results of expressions. For example, the local environment area for the method

```
newAt: initialLocation |newBox|
  newBox ← box new.
     .
     .
     .
```

must contain space for the parameter `initialLocation`, the temporary variable `newBox`, and intermediate results:



The nonlocal environment includes all other visible variables, namely, the instance variables and the class variables. The instance variables are stored in the representation of the object that received the message, therefore, a simple pointer to this object makes them accessible. The object representation contains a pointer to the class of which the object is an instance, therefore, the class variables are also accessible via the object reference. Finally, since the class representation contains a pointer to its superclass, the superclass variables are also accessible. The parts of an activation record are shown in Figure 12.17.

Notice that this approach to accessing nonlocals is very similar to the static chain of environments that we encountered in block-structured languages. There, the static chain led from the innermost active environment to the outermost environment. Here, the static chain leads from the active method to the object that received the message, and from there up through the class hierarchy, to terminate at the class `object`. Just as in block-structured languages, variable accessing requires knowing the static distance to the variable and skipping that many links of the static chain to get to the environment that defines the variable. (If you are unclear about this, review Chapter 6, Section 6.1.)

▩ ***Exercise 12-15:*** Show the code for accessing a Smalltalk variable, given its coordinates. Assume the environment coordinate is given as a static distance.

Message Sending and Returning

We will now list and analyze the steps that must take place when a message is sent to an object:

1. Create an activation record for the receiver (callee).
2. Identify the method being invoked by extracting the template from the message and then looking it up in the message dictionary for the receiving object's class or superclasses.

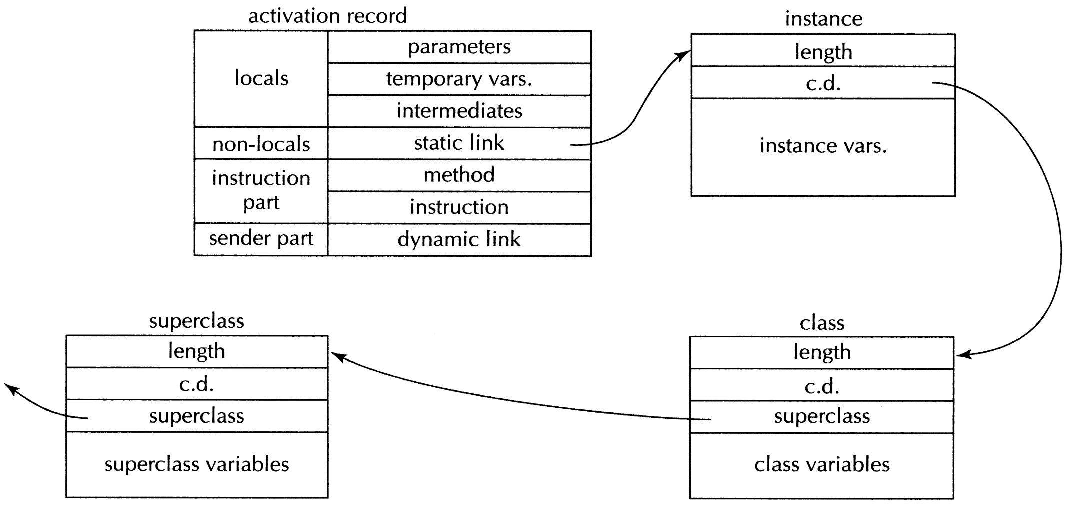


Figure 12.17 Parts of an Activation Record

That is, if it is not defined in the class, we must look in the superclass; if it is not defined in the superclass, we must look in its superclass, and so on.
3. Transmit the parameters to the receiver's activation record.
4. Suspend the sender (caller) by saving its state in its activation record.
5. Establish a path (dynamic link) from the receiver back to the sender, and establish the receiver's activation record as the active one.

As we would expect, returning from a method must reverse this process:

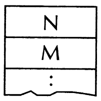
1. Transmit the returned object (if any) from the receiver back to the sender.
2. Resume execution of the sender by restoring its state from its activation record.

We have omitted deallocation of the activation record as part of returning; we explain this next.

In accordance with the Information Hiding Principle, the Storage Manager handles allocation and deallocation of *all* objects; this includes activation record objects. The effect of this is that activation records are created from free storage and reclaimed by reference counting, just like other objects. This is the reason we do not explicitly deallocate activation records. This approach is quite different from the implementation of the other languages we have studied in which activation records occupy contiguous locations on a stack. The Smalltalk approach is a little less efficient, although this is compensated for by its greater simplicity and regularity.

There is another reason that Smalltalk does not allocate its activation records on a stack. The implementation of concurrency we discussed in Section 12.4 has an important limitation: The scheduler must wait for each task to return from its `run` message before it can schedule the next task. In fact, if one of the tasks went into an infinite loop, the entire system might halt. Therefore, real Smalltalk systems provide an interrupt facility that automatically interrupts the executing task after it has run for a certain amount of time. This ensures that all of the active tasks get serviced regularly. It also means that a runaway task can be deactivated (removed from the `sched` set) by another task.

What does this have to do with activation records? Consider what must occur when a task is interrupted. Since its execution is being suspended, its state must be stored in its activation record. Now, suppose that Smalltalk's activation records were held on a stack. Further, suppose that in a task *A*, the method *M* is activated; an activation record for *M* will be placed on the top of the stack. Suppose that before *M* returns, task *A* is interrupted and task *B* is resumed. Assume that task *B* activates method *N*, which causes an activation record for *N* to be stacked. The stack now looks like this:



Next, suppose that task *B* is interrupted before *N* returns and that task *A* is resumed. Finally, suppose that task *M* returns and deletes its activation record from the stack. We are faced with two possibilities: Either (1) popping *M* will also pop off *N*'s activation record, which is incorrect since *N* has not returned; or (2) *M* is popped out of the middle of the stack leaving a hole, which means that we are not using a stack after all. The point is that a stack is the appropriate data structure only if we are dealing with something that follows a LIFO (last-in–first-out) discipline. Procedures in a sequential (i.e., nonconcurrent) language follow a LIFO discipline; procedures in a concurrent language do not.

■ ***Exercise 12-16:*** Write the instruction sequences for sending a message and returning from a method in Smalltalk. These are simple adaptations of the static chain call and return sequences discussed in Chapter 6, Section 6.2. Assume that the Storage Manager functions listed on p. 429 of the present section are available.

■ ***Exercise 12-17:*** One solution to the orthogonal classification problem (Section 12.3) is multiple inheritance, that is, allowing each class to have more than one superclass. Explain how this solution would complicate accessing variables.

12.6 DESIGN: OBJECT-ORIENTED EXTENSIONS

Many Languages Have Been Extended for Object-Oriented Programming

In the wake of Smalltalk, many languages have been extended with object-oriented features, including fifth-generation languages such as LISP, and third- and fourth-generation languages such as C and Ada 83. This is a venerable tradition, since Simula-67 is essentially an object-oriented extension to Algol-60, a second-generation language.

Tagged Types Support Programming by Extension

We have already compared Smalltalk classes and Ada packages; primarily the Ada mechanisms are more static, which is good for compile-time error checking, but bad for flexibility. Therefore Ada 95 provides new features, suggested by object-oriented languages, to support "programming by extension," the ability to extend and refine software systems without needing to recompile and retest the parts already implemented.

Ada's object-oriented features are oriented toward an external representation of objects, unlike Smalltalk's internal representation (see p. 276 on internal and external representations). For example, the Ada translation of the class in Figure 12.4 is

```
package Display_Objects is
  type Display_Object is tagged
    record
      Loc: Display_location;
    end record;
```

```
   procedure Go_to (The_Object: Display_Object;
                    New_Loc: Display_location);
   procedure Show (The_Object: Display_Object);
   procedure Erase (The_Object: Display_Object);
end Display_Objects;
```

The new keyword tagged means that the records have a hidden discriminant, which permits the type to be refined by a subtype declaration. This has the effect of a subclass declaration, as we can see from this Ada translation of the subclass in Figure 12.5:

```
with Display_Objects;
package Boxes is
   type Box is new Display_Objects.Display_Object with
      record
         Tilt, Size:   Float;
         The_Scribe:   Scribe;
      end record;
   procedure Shape (The_Box: Box);
   .
   .
   .
end Boxes;
```

Box is declared to be a subtype of Display_Object; it extends the record type Box by adding additional fields. Furthermore it inherits the operations defined on Display_Object, such as Go_to (that is, the operations defined in the same package as the type). In general, the refining package can add subprograms to those defined in the parent package, or refine their definition. For example, Box adds the Shape procedure to Go_to, Show, and Erase, which are inherited from Display_Object. If there are several subtypes of a type, which define a subprogram differently, then the discriminant can be used by the compiler to determine which subprogram to invoke.

The preceding definitions of Display_Object and Box make their components publicly visible, as they would be in Smalltalk, but we would more likely want them to be private. This can be accomplished by declaring Display_Object as a private type:

```
package Display_Objects is
   type Display_Object is tagged private;
   .
   .
   .
private
   type Display_Object is tagged
      record
         Loc: Display_location;
      end record;
end Display_Objects;
```

We can also make the `Box` type private:

```
with Display_Objects;
package Boxes is
   type Box is
      new Display_Objects.Display_Object with private;
   .
   .
   .
private
   type Box is
      new Display_Objects.Display_Object with
         record
            Tilt, Size:   Float;
            The_Scribe:   Scribe;
         end record;
end Boxes;
```

In this case `Boxes` cannot make use of the components of `Display_Object`, since they are private.

Types and Their Refinements Can be Defined in a Single Package

If we want the derived types to be able to make use of the private components of the parent type, then they must be declared together in one package. But this raises a problem: if types and derived types and all their operations are defined together in the same package, how do we tell which are the primitive operations of the type, that is, the operations inherited by the derived types? Ada specifies that the definition of the primitive operations of a type is terminated by the definition of a derivation of that type. In effect the declarations of the types and derived types (corresponding to a Smalltalk class and its subclasses) are grouped in a single package with no explicit separation between them. For example,

```
package All_Display_Objects
   type Display_Object is tagged private;
   ... operations for Display_Object ...
   type Box is new Display_Object with private;
   ... operations for Box ...
   type Window is new Display_Object with private;
   ... operations for Window ...
   ... etc. ...
end All_Display_Objects;
```

Of course, this is a violation of modular programming and impedes separate compilation and programming by extension, but that is the price we must pay if we want the extension to have access to the private parts of `Display_Objects`.

Abstract Declarations Are Place-Holders in Parent Types

The Ada facilities for object-oriented programming, as we have described them, do not permit a common practice in object-oriented programming. An example in Smalltalk is the `shape` method in the `displayObject` class: every `displayObject` is expected to have a `shape` method, but it must be provided by the subclasses of `displayObject` (e.g., Figure 12.5). Therefore, we can define methods (such as `goto:`, `show`, and `erase`) that work for any `displayObject`, since we expect each `displayObject` to be able to trace its own shape, but how it does so varies from subclass to subclass. If we do the analogous thing in Ada, defining a `Shape` procedure in each refinement of `Display_Object`, then it will result in each of these refinements having the procedure, but not the parent type. Therefore we cannot write programs (such as `Go_to`, `Show`, and `Erase`) that apply to an arbitrary `Display_Object`. To accomplish this, we must define a "place holder" `Shape` procedure for `Display_Objects`, which is replaced by a particular procedure in its refinements. Such a place holder is called an abstract subprogram in Ada. Our definitions might look like this (notice the keyword `abstract` in the specifications of `Display_Object` and `Shape`):

```
package Display_Objects
   type Display_Object is abstract tagged private;
   procedure Shape (The_Object: Display_Object) is abstract;
   procedure Go_to (The_Object: Display_Object;
                    New_Loc: Display_location);
   procedure Show (the_Object: Display_Object);
   ... other operations for Display_Object ...
end Display_Objects;

package Boxes
   type Box is new Display_Object with private;
   procedure Shape (The_Box: Box);
   ... other operations for Box ...
end Boxes;

package Windows;
   type Window is new Display_Object with private;
   procedure Shape (The_Window: Window);
   ... operations for Window ...
end Windows;
```

In the package body for `Display_Objects` we can define the `Show` and `Erase` procedures in terms of the abstract `shape` procedure, and the `Go_to` procedure in terms of `Show` and `Erase`:

```
package body Display_Objects is

   procedure Go_to (The_Object: Display_Object;
                    New_Loc: Display_location) is
```

```
      Erase (The_Object);
      Set_Loc (The_Object, New_Loc);
      Show (The_Object);
   end Go_To:

   procedure Show (The_Object: Display_Object) is
   begin
      Set_Color (The_Object, Ink);
      Shape (The_Object);
   end Show;

   . . . more definitions . . .
end Display_Objects;
```

(These definitions assume that other necessary procedures, such as `Set_Loc` and `Set_Color`, are defined on `Display_Objects`.)

Class Types Allow Limited Dynamic Typing

There is one more significant new feature provided by Ada 95 to support object-oriented programming. It is important to realize that `Box` and `Window` are *refinements* of `Display_Object`; although `Display_Object` is the parent type of `Box` and `Window`, it does not *include* `Box` and `Window` in the same sense that a Smalltalk superclass includes its subclasses. However, for a tagged type such as `Display_Object` Ada automatically defines the associated type `Display_Object'Class`, which does include `Display_Object` and all its refinements. Objects of a `Class` type have a tag at run-time, which allows determination of their subtype and allows the appropriate procedures (e.g., `Shape` for `Boxes` or `Shape` for `Windows`) to be applied. A `Class` is an unconstrained type so, although it can be passed as a parameter, it cannot be used as the type of a variable (since the compiler does not know how much storage to allocate for it). However, an `access` (i.e., pointer) to a class type does occupy a fixed amount of storage. Thus a variable capable of holding any `Display_Object` could be declared:

```
The_Object: access Display_Object'Class;
```

With the addition of `class` types Ada has in effect two type systems: the ordinary static typing inherited from Ada 83, and Smalltalk-like dynamic typing, with run-time tags, to support object-oriented programming. Programmers must decide which is appropriate for their applications, just as they must determine whether an object can be accessed directly or indirectly through a pointer. We may contrast Ada's design decisions with Common LISP, in which all types are dynamic, unless a variable has been declared to be of a specific type, and Smalltalk, in which all types are dynamic.

Ada's Support for Object-Oriented Programming Is Complex

Although we have discussed only the most important features, you can see that the support for object-oriented programming in Ada adds considerable complexity to the language. Part of the complexity comes from grafting the object-oriented features onto a type system that was essentially complete and designed for different trade-offs (e.g., efficiency on small embedded computers). In effect, the Ada type system has been stretched to accommodate two different philosophies of programming, one, in which the programmer assumes close control of storage allocation and access by deciding in advance how this storage will be used, and another, in which these decisions are delayed and entrusted to the language system. Such growth is typical of the "featuritis" that often plagues the evolution of programming languages (recall Section 8.3).

C++ Extends C for Object-Oriented Programming

C is another language that has been extended to support object-oriented programming. C++ is largely the work of Bjarne Stroustrup, who learned the advantages of the Simula class construct in the late 1970s and began investigating how it could be incorporated into C. Over the period 1979–1988 his efforts resulted in the C++ language, which has continued to evolve, but remains largely upward compatible with C.

C++ supports internally represented objects through a class declaration, which allows multiple inheritance of definitions from superclasses. The components of a class can be `public` (visible everywhere), `protected` (visible in the class and its subclasses), or `private` (visible only in the class). Procedures can be redefined in subclasses if they were defined as `virtual` procedures in the superclass (the `virtual` terminology comes from Simula and corresponds to `abstract` in Ada). Through "templates" C++ supports limited parameterization of classes (analogous to Ada `generic` specification of packages).

C++ continues C's third-generation emphasis on efficiency, and its type checking is completely static. Although it is more strongly typed than C, it is also committed to supporting the low level machine access expected by many C programmers.

Java Is A More Advanced Object-Oriented Language Derived from C

More recently C++ became the starting point for the Java programming language, which was designed for networked and distributed programming environments. Therefore there has been a greater emphasis than in C or C++ on security, robustness, machine independence, portability, and late binding. To this end, Java omits some C++ features, such as operator overloading, pointers, excessive coercions, and multiple inheritance (retained in a limited form), but adds others, such as automatic garbage collection and more secure type checking. In this regard it is a move toward the higher level storage management and dynamic typing of languages such as Smalltalk and LISP. Overall, Java is a much more advanced language than C++, which may make it unacceptable to many C programmers; time will tell.

CLOS Extends LISP for Object-Oriented Programming

LISP has been extended—several times—to support object-oriented programming, first in the late 1970s, inspired by Smalltalk, and most recently in CLOS, the Common LISP Object System, which is part of the draft Common Lisp standard. In many ways LISP is a convenient base on which to build an object-oriented programming system; we saw in Section 9.3 how LISP uses property lists to represent information about atomic objects. In particular, object-oriented programming fits well into LISP's dynamic typing, and LISP's built-in types are integrated with its class system. Classes are first-class objects and belong to a class called `standard-class`. CLOS provides externally represented objects since methods are implemented as generic functions. One consequence is that the method to be applied is determined by all the method's arguments, not just its first argument, as in Smalltalk (p. 417). CLOS also supports multiple inheritance, but, although the basic idea is simple, the precise rules for multiple inheritance must be expressed *as an algorithm* for determining the "class precedence list" of a class!

■ ***Exercise 12-18*:*** Compare and contrast the object-oriented programming faculties of Ada 95 and C++.

■ ***Exercise 12-19*:*** Compare and contrast the object-oriented programming facilities of C++ and Java.

■ ***Exercise 12-20*:*** Compare and contrast object-oriented programming in Smalltalk, Java, and CLOS.

■ ***Exercise 12-21*:*** It is claimed that the *interface* feature of Java eliminates the need for multiple inheritance. Do you agree? Defend your answer.

■ ***Exercise 12-22*:*** Compare and contrast the multiple inheritance facilities of C++, Java, and CLOS.

■ ***Exercise 12-23*:*** Defend or attack Java's provision of garbage collection in place of the explicit (user controlled) storage management of C++.

12.7 EVALUATION AND EPILOG

Smalltalk is Small, Flexible, and Extensible

Smalltalk demonstrates what can be accomplished with a small, simple, regular language. The small number of independent concepts in Smalltalk makes it an easy language to learn. Its ideas are orthogonal—they can be understood independently—and are individually quite simple. Even concurrency is quite manageable since the object orientation of Smalltalk means that individual objects can be treated as autonomous agents. As Kay has said, "point of view is worth 80 IQ points."

Smalltalk has demonstrated its flexibility by being used in a number of applications, including simulations, games, office automation, graphics, computer-assisted instruction, and artificial intelligence. It has even been used in systems programming; recall that most of the Smalltalk system is written in Smalltalk.

The design of Smalltalk has been dictated by several powerful principles: Simplicity, Regularity, Abstraction, Security, and Information Hiding. It is an excellent example of the Elegance Principle.

Smalltalk Is an Example of a Programming Environment

Smalltalk has introduced several important ideas other than the language itself. One of these is the idea of windows as a uniform interactive interface. The importance of windows is that they allow a large amount of information to be organized on a display screen and that they allow a programmer to keep track of several different activities at a time. They can be manipulated like sheets of paper and thus build upon our everyday skills with physical objects. Window-like systems have been adopted by many newer programming systems, so that now they are virtually universal. It is more than a metaphor to call windows a *transparent* interface.

Smalltalk is also an excellent example of an advanced programming environment (or integrated programming system). Although it was not the first programming environment, its integration of graphics, windows, and an advanced language demonstrates what can now be accomplished. Like LISP programming environments (e.g., Interlisp), the Smalltalk environment is a natural extension of the language, so they form a seamless whole. The experience acquired in Smalltalk's design and use will be valuable to future programming environments, including those for conventional languages such as Ada.

Smalltalk Introduces a New Programming Paradigm

Smalltalk's most important contribution may be its distinctive view of programming. Although most of its ideas have appeared in other languages, Smalltalk's integration of these ideas is novel.

The essence of the Smalltalk view is that programming is simulation. Smalltalk incorporates into the language the idea that programs model, sometimes in a very abstract form, some aspects of the real world. Since the real world is populated by objects, Smalltalk provides an object-oriented method of programming. These programming objects have properties just like real-world objects.

Science develops models of the real world by classifying objects according to similarities in their behavior. Smalltalk does the same thing with its class structure. Classification is an essential part of organizing knowledge, although the strictly hierarchical classification provided by Smalltalk is inferior to a system that permits orthogonal classification.

We understand the interactions of objects by causal laws, in particular, by laws that determine how objects act when acted upon by other objects. Smalltalk permits causal interactions among objects by message sending: One object may cause another to do something by sending it a message. Just as in the real world, an object responds to a situation according to its nature, that is, according to its particular properties (its instance variables) and its general properties (its class).

Object-oriented programming—the systematic treatment of programming as simulation—provides a fundamentally different model of programming from those we've seen before.[4] It is our second example of a *fifth-generation* programming paradigm. Whereas func-

[4] Unfortunately the term "object-oriented" is often used loosely. In particular, because of the importance of classes in object-oriented programming, the term often refers to no more than the use of abstract-data types in programming (as in fourth-generation languages), which Kay calls "simply old style programming with fancier constructs."

tional programming concentrates on atemporal mathematical relationships, object-oriented programming addresses directly the behavior of objects in time. Which is better? The answer probably depends on the nature of the problem, on whether time is an essential aspect of the problem to be solved. Each paradigm has its own stylistic inclinations. It may very well be that function-oriented programming and object-oriented programming are *complementary*, rather than *competing*, fifth-generation programming language technologies.[5]

EXERCISES

1*. For what purposes would you (or do you) use a very inexpensive, powerful personal computer. Would Smalltalk be a good language for these purposes? Would another language be better?

2. Define a class `cell` that implements LISP-like list structures.

3*. Read about the Simula *class* mechanism (e.g., in Dahl, Dijkstra, and Hoare, *Structured Programming*, Academic Press, 1972) and compare it with Smalltalk's classes.

4.** Modify Smalltalk to permit nonhierarchical and orthogonal classification. Briefly discuss the implementation issues.

5*. Smalltalk has been used to implement several experimental software systems. Critique Alan Borning's article, "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory" (*ACM Transactions on Programming Languages and Systems 3*, 4, October 1981).

6*. Design a better message sending syntax for Smalltalk.

7*. Discuss Smalltalk's two-dimensional syntax.

8*. Show how to do object-oriented programming in LISP. That is, if `B1` is an object of class box, then `(B1 grow: 10)` will send the message `grow: 10` to `B1`.

9*. Discuss the problems of compiling Smalltalk into conventional object code. Pay particular attention to the matter of type checking.

10.** (Difficult) Write a Smalltalk Virtual Machine in some conventional language such as Pascal or Ada.

11*. The August 1981 issue of *Byte Magazine* (*6*, 8) is devoted to Smalltalk. Critique one of the articles in this issue.

12*. Discuss the Smalltalk programming environment. Is this a good environment in which to develop software? What improvements can you suggest?

13*. Write an essay on the topic "Computer Programming Is Just Simulation."

14*. Read Alan Kay's "Early History of Smalltalk" (*SIGPLAN Notices 28*, 3, March 1993) and write a report on innovation in programming language design.

[5] See MacLennan (1983) for a discussion of the fundamental distinction between *value-oriented* (function-based) and *object-oriented* (simulation-based) programming and of the roles for each.