Chapter 1

Thinking Object-Oriented

Although the fundamental features of what we now call object-oriented programming were invented in the 1960's, object oriented languages really came to the attention of the computing public-at-large in the 1980's. Two seminal events were the publication of a widely-read issue of *Byte* (August 1981) that described the programming language Smalltalk, and the first international conference on object-oriented programming languages and applications, held in Portland, Oregon in 1986.

Now, almost twenty years later, it is still the case that, as I noted in the first edition of this book (in 1991):

> Object-oriented programming (OOP) has become exceedingly popular in the past few years. Software producers rush to release object-oriented versions of their products. Countless books and special issues of academic and trade journals have appeared on the subject. Students strive to list "experience in object-oriented programming" on their résumés. To judge from this frantic activity, object-oriented programming is being greeted with even more enthusiasm than we saw heralding earlier revolutionary ideas, such as "structured programming" or "expert systems."

My intent in these first two chapters is to investigate and explain the basic principles of object-oriented programming, and in doing so to illustrate the following two propositions:

- OOP is a revolutionary idea, totally unlike anything that has come before in programming.
- OOP is an evolutionary step, following naturally on the heels of earlier programming abstractions.

1.1 Why Is OOP Popular?

There are a number of important reasons why in the past two decades object-oriented programming has become the dominant programming paradigm. Object-oriented programming scales very well, from the most trivial of problems to the most complex tasks. It provides a form of abstraction that resonates with techniques people use to solve problems in their everyday life. And for most of the dominant object-oriented languages there are an increasingly large number of libraries that assist in the development of applications for many domains.

Object-oriented programming is just the latest in a long series of solutions that have been proposed to help solve the "software crisis." At heart, the software crisis simply means that our imaginations, and the tasks we would like to solve with the help of computers, almost always outstrip our abilities.

But while object-oriented techniques *do* facilitate the creation of complex software systems, it is important to remember that OOP is not a panacea. Programming a computer is still one of the most difficult tasks ever undertaken by humans; becoming proficient in programming requires talent, creativity, intelligence, logic, the ability to build and use abstractions, and experience–even when the best of tools are available.

I suspect another reason for the particular popularity of languages such as C++ and Delphi (as opposed to languages such as Smalltalk and Beta) is that managers and programmers alike hope that a C or Pascal programmer can be changed into a C++ or Delphi programmer with no more effort than the addition of a few characters to their job title. Unfortunately, this hope is a long way from being realized. Object-oriented programming is a new way of thinking about what it means to compute, about how we can structure information and communicate our intentions both to each other and to the machine. To become proficient in object-oriented techniques requires a complete reevaluation of traditional software development.

1.2 Language and Thought

> "Human beings do not live in the objective world alone, nor alone in the world of social activity as ordinarily understood, but are very much at the mercy of the particular language which has become the medium of expression for their society. It is quite an illusion to imagine that one adjusts to reality essentially without the use of language and that language is merely an incidental means of solving specific problems of communication or reflection. The fact of the matter is that the 'real world' is to a large extent unconsciously built up on the language habits of the group.... We see and hear and otherwise experience very largely as we do because the language habits of our community predispose certain choices of interpretation."
>
> Edward Sapir (quoted in [Whorf 1956])

This quote emphasizes the fact that the languages we speak directly influence the way in which we view the world. This is true not only for natural languages, such as the kind studied by the early twentieth century American linguists Edward Sapir and Benjamin Lee Whorf, but also for artificial languages such as those we use in programming computers.

1.2.1 Eskimos and Snow

An almost universally cited example of the phenomenon of language influencing thought, although also perhaps an erroneous one (see the references cited at the end of the chapter) is the "fact" that Eskimo (or Inuit) languages have many words to describe various types of snow–wet, fluffy, heavy, icy, and so on. This is not surprising. Any community with common interests will naturally develop a specialized vocabulary for concepts they wish to discuss. (Meteorologists, despite working in English, face similar problems of communication and have also developed their own extensive vocabulary).

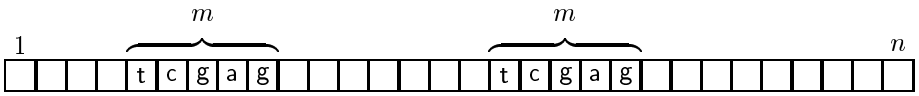
What is important is to not overgeneralize the conclusion we can draw from this simple observation. It is not that the Eskimo eye is in any significant respect different from my own, or that Eskimos can see things I cannot perceive. With time and training I could do just as well at differentiating types of snow. But the language I speak (namely, English) does not *force* me into doing so, and so it is not natural to me. Thus, a different language (such as Inuktitut) can *lead* one (but does not *require* one) to view the world in a different fashion.

To make effective use of object-oriented principles requires one to view the world in a new way. But simply using an object-oriented language (such as Java or C++) does not, by itself, force one to become an object-oriented programmer. While the use of an object-oriented language will simplify the development of object-oriented solutions, it is true, as it has been quipped, that "FORTRAN programs can be written in any language."

1.2.2 An Example from Computer Languages

The relationship we noted between language and thought for natural languages is even more pronounced in artificial computer languages. That is, the language in which a programmer thinks a problem will be solved will color and alter, fundamentally, the way in which an algorithm is developed.

An example will illustrate this relationship between computer language and problem solution. Several years ago a student working in genetic research was faced with a task in the analysis of DNA sequences. The problem could be reduced to relatively simple form. The DNA is represented as a vector of N integer values, where N is very large (on the order of tens of thousands). The problem was to discover whether any pattern of length M, where M was a fixed and small constant (say five or ten) is ever repeated in the array of values.



The programmer dutifully sat down and wrote a simple and straightforward FORTRAN program something like the following:

```
    DO 10 I = 1, N-M
    DO 10 J = 1, N-M
    FOUND = .TRUE.
    DO 20 K = 1, M
20  IF X[I+K-1] .NE. X[J+K-1] THEN FOUND = .FALSE.
    IF FOUND THEN ...
10  CONTINUE
```

He was somewhat disappointed when trial runs indicated his program would need many hours to complete. He discussed his problem with a second student, who happened to be proficient in the programming language APL, who said that she would like to try writing a program for this problem. The first student was dubious; after all, FORTRAN was known to be one of the most "efficient" programming languages–it was compiled, and APL was only interpreted. So it was with a certain amount of incredulity that he discovered that the APL programmer was able to write an algorithm that worked in a matter of minutes, not hours.

What the APL programmer had done was to rearrange the problem. Rather than working with a vector of N elements, she reorganized the data into a matrix with roughly N rows and M columns:

x_1	x_2	...	x_m
x_2	x_3	...	x_{m+1}
...			...
x_{n-m}	...	...	x_{n-1}
$x_{n-(m-1)}$	...	x_{n-1}	x_n

She then ordered this matrix by rows (that is, treated each row as a unit, moving entire rows during the process of sorting). If any pattern was repeated, then two adjacent rows in the ordered matrix would have identical values.

```
      .   .   .
T  G  G  A  C  C
T  G  G  A  C  C
      .   .   .
```

It was a trivial matter to check for this condition. The reason the APL program was faster had nothing to do with the speed of APL versus FORTRAN; it was simply that the FORTRAN program employed an algorithm that was

$O(M \times N^2)$, whereas the sorting solution used by the APL programmer required approximately $O(M \times N \log N)$ operations.

The point of this story is not that APL is in any way a "better" programming language than FORTRAN, but that the APL programmer was naturally led to discover an entirely different form of solution. The reason, in this case, is that loops are very difficult to write in APL whereas sorting is trivial–it is a built-in operator defined as part of the language. Thus, because the sorting operation is so easy to perform, good APL programmers tend to look for novel applications for it. It is in this manner that the programming language in which the solution is to be written directs the programmer's mind to view the problem in a certain way.

1.2.3 Church's Conjecture and the Whorf Hypothesis *

The assertion that the language in which an idea is expressed can influence or direct a line of thought is relatively easy to believe. However, a stronger conjecture, known in linguistics as the Sapir-Whorf hypothesis, goes much further and remains controversial.

The Sapir-Whorf hypothesis asserts that it may be possible for an individual working in one language to imagine thoughts or to utter ideas that cannot in any way be translated, cannot even be understood, by individuals operating in a different linguistic framework. According to advocates of the hypothesis, this can occur when the language of the second individual has no equivalent words and lacks even concepts or categories for the ideas involved in the thought. It is interesting to compare this possibility with an almost directly opposite concept from computer science–namely, Church's conjecture.

Starting in the 1930s and continuing through the 1940s and 1950s there was a great deal of interest within the mathematical and nascent computing community in a variety of formalisms that could be used for the calculation of functions. Examples are the notations proposed by Church [Church 1936], Post [Post 1936], Markov [Markov 1951], Turing [Turing 1936], Kleene [Kleene 1936] and others. Over time a number of arguments were put forth to demonstrate that many of these systems could be used in the simulation of other systems. Often, such arguments for a pair of systems could be made in both directions, effectively showing that the systems were identical in computation power. The sheer number of such arguments led the logician Alonzo Church to pronounce a conjecture that is now associated with his name:

> **Church's Conjecture:** Any computation for which there exists an effective procedure can be realized by a Turing machine.

By nature this conjecture must remain unproven and unprovable, since we have no rigorous definition of the term "effective procedure." Nevertheless, no

[0] Section headings followed by an asterisk indicate optional material.

counterexample has yet been found, and the weight of evidence seems to favor affirmation of this claim.

Acceptance of Church's conjecture has an important and profound implication for the study of programming languages. Turing machines are wonderfully simple mechanisms, and it does not require many features in a language to simulate such a device. In the 1960s, for example, it was demonstrated that a Turing machine could be emulated in any language that possessed at least a conditional statement and a looping construct [Böhm 1966]. (This greatly misunderstood result was the major ammunition used to "prove" that the infamous `goto` statement was unnecessary.)

If we accept Church's conjecture, any language in which it is possible to simulate a Turing machine is sufficiently powerful to perform *any* realizable algorithm. (To solve a problem, find the Turing machine that produces the desired result, which by Church's conjecture must exist; then simulate the execution of the Turing machine in your favorite language.) Thus, arguments about the relative "power" of programming languages–if by power we mean "ability to solve problems"–are generally vacuous. The late Alan Perlis had a term for such arguments, calling them a "Turing Tarpit" because they are often so difficult to extricate oneself from, and so fundamentally pointless.

Note that Church's conjecture is, in a certain sense, almost the exact opposite of the Sapir-Whorf hypothesis. Church's conjecture states that in a fundamental way all programming languages are identical. Any idea that can be expressed in one language can, in theory, be expressed in any language. The Sapir-Whorf hypothesis claims that it is possible to have ideas that can be expressed in one language that can not be expressed in another.

Many linguists reject the Sapir-Whorf hypothesis and instead adopt a sort of "Turing-equivalence" for natural languages. By this we mean that, with a sufficient amount of work, any idea can be expressed in any language. For example, while the language spoken by a native of a warm climate may not make it instinctive to examine a field of snow and categorize it by type or use, with time and training it certainly can be learned. Similarly, object-oriented techniques do not provide any new computational power that permits problems to be solved that cannot, *in theory*, be solved by other means. But object-oriented techniques *do* make it *easier* and more natural to address problems in a fashion that tends to favor the management of large software projects.

Thus, for both computer and natural languages the language will *direct* thoughts but cannot *proscribe* thoughts.

1.3 A New Paradigm

Object-oriented programming is frequently referred to as a new programming *paradigm*. Other programming paradigms include the imperative-programming paradigm (languages such as Pascal or C), the logic programming paradigm (Prolog), and the functional-programming paradigm (ML or Haskell).

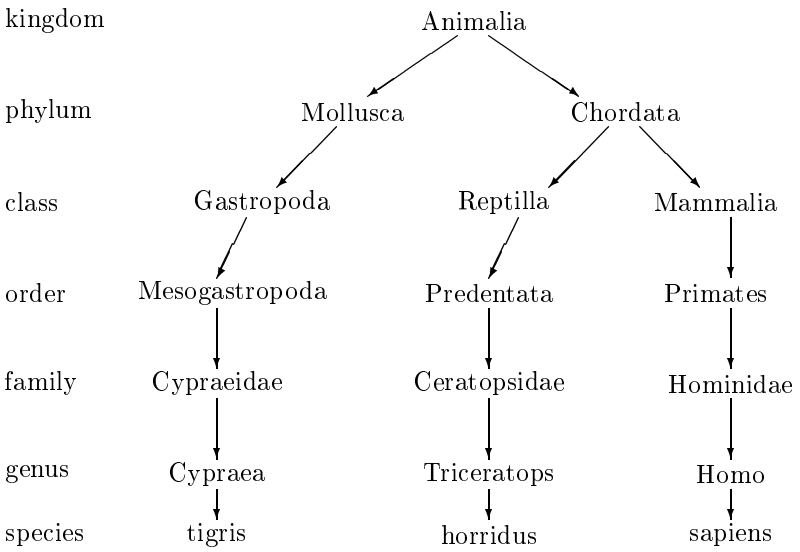


Figure 1.1: The Linnæn Inheritance Hierarchy

It is interesting to examine the definition of the word "paradigm." The following is from the *American Heritage Dictionary of the English Language*:

> **par a digm** *n.* **1**. A list of all the inflectional forms of a word taken as an illustrative example of the conjugation or declension to which it belongs. **2.** Any example or model. [Late Latin *paradīgma*, from Greek *paradeigma*, model, from *paradeiknunai*, to compare, exhibit.]

At first blush, the conjugation or declension of Latin words would seem to have little to do with computer programming languages. To understand the connection, we must note that the word was brought into the modern vocabulary through an influential book, *The Structure of Scientific Revolutions*, by the historian of science Thomas Kuhn [Kuhn 1970]. Kuhn used the term in the second form, to describe a set of theories, standards, and methods that together represent a way of organizing knowledge–that is, a way of viewing the world. Kuhn's thesis was that revolutions in science occur when an older paradigm is reexamined, rejected, and replaced by another.

It is in this sense, as a model or example and as an organizational approach, that Robert Floyd used the term in his 1979 ACM Turing Award lecture [Floyd 1979], "The Paradigms of Programming." A programming paradigm is a way of conceptualizing what it means to perform computation and how tasks to be carried out on a computer should be structured and organized.

Although new to computation, the organizing technique that lies at the heart

of object-oriented programming can be traced back at least as far as Carolus Linnæus (1707–1778) (Figure 1.1). It was Linnæus, you will recall, who categorized biological organizisms using the idea of phylum, genus, species, and so on.

Paradoxically, the style of problem solving embodied in the object-oriented technique is frequently the method used to address problems in everyday life. Thus, computer novices are often able to grasp the basic ideas of object-oriented programming easily, whereas people who are more computer literate are often blocked by their own preconceptions. Alan Kay, for example, found that it was often easier to teach Smalltalk to children than to computer professionals [Kay 1977].

In trying to understand exactly what is meant by the term *object-oriented programming*, it is useful to examine the idea from several perspectives. The next few sections outline two aspects of object-oriented programming; each illustrates a particular reason that this technique should be considered an important new tool.

1.4 A Way of Viewing the World

To illustrate some of the major ideas in object-oriented programming, let us consider first how we might go about handling a real-world situation and then ask how we could make the computer more closely model the techniques employed.

Suppose an individual named Chris wishes to send flowers to a friend named Robin, who lives in another city. Because of the distance, Chris cannot simply pick the flowers and take them to Robin in person. Nevertheless, it is a task that is easily solved. Chris simply walks to a nearby flower shop, run by a florist named Fred. Chris will tell Fred the kinds of flowers to send to Robin, and the address to which they should be delivered. Chris can then be assured that the flowers will be delivered expediently and automatically.

1.4.1 Agents and Communities

At the risk of belaboring a point, let us emphasize that the mechanism that was used to solve this problem was to find an appropriate *agent* (namely, Fred) and to pass to this agent a *message* containing a request. It is the *responsibility* of Fred to satisfy the request. There is some *method*–some algorithm or set of operations–used by Fred to do this. Chris does not need to know the particular method that Fred will use to satisfy the request; indeed, often the person making a request does not want to know the details. This information is usually *hidden* from inspection.

An investigation, however, might uncover the fact that Fred delivers a slightly different message to another florist in the city where Robin lives. That florist, in turn, perhaps has a subordinate who makes the flower arrangement. The florist then passes the flowers, along with yet another message, to a delivery person, and so on. Earlier, the florist in Robin's city had obtained her flowers from a

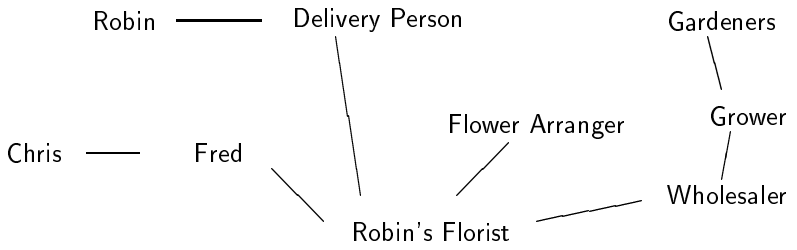


Figure 1.2: The community of agents helping delivery flowers

flower wholesaler who, in turn, had interactions with the flower growers, each of whom had to manage a team of gardeners.

So, our first observation of object-oriented problem solving is that the solution to this problem required the help of many other individuals (Figure 1.2). Without their help, the problem could not be easily solved. We phrase this in a general fashion as the following:

> An object oriented program is structured as a *community* of interacting agents, called *objects*. Each object has a role to play. Each object provides a service, or performs an action, that is used by other members of the community.

1.4.2 Messages and Methods

The chain reaction that ultimately resulted in the solution to Chris's problem began with a request given to the florist Fred. This request lead to other requests, which lead to still more requests, until the flowers ultimately reached Chris's friend Robin. We see, therefore, that members of this community interact with each other by making requests. So, our next principle of object-oriented problem solving is the vehicle used to indicate an action to be performed:

> Action is initiated in object-oriented programming by the transmission of a *message* to an agent (an *object*) responsible for the action. The message encodes the request for an action and is accompanied by any additional information (arguments) needed to carry out the request. The *receiver* is the object to whom the message is sent. If the receiver accepts the message, it accepts the responsibility to carry out the indicated action. In response to a message, the receiver will perform some *method* to satisfy the request.

We have noted the important principle of *information hiding* in regard to message passing–that is, the client sending the request need not know the actual

means by which the request will be honored. There is another principle, all too human, that we see is implicit in message passing. If there is a task to perform, the first thought of the client is to find somebody else he or she can ask to do the work. This second reaction often becomes atrophied in many programmers with extensive experience in conventional techniques. Frequently, a difficult hurdle to overcome is the idea in the programmer's mind that he or she must write everything and not use the services of others. An important part of object-oriented programming is the development of reusable components, and an important first step in the use of reusable components is a willingness to trust software written by others.

Messages versus Procedure Calls

Information hiding is also an important aspect of programming in conventional languages. In what sense is a message different from, say, a procedure call? In both cases, there is a set of well-defined steps that will be initiated following the request. But, there are two important distinctions.

The first is that in a message there is a designated *receiver* for that message; the receiver is some object to which the message is sent. In a procedure call, there is no designated receiver.

The second is that the *interpretation* of the message (that is, the method used to respond to the message) is determined by the receiver and can vary with different receivers. Chris could give a message to a friend named Elizabeth, for example, and she will understand it and a satisfactory outcome will be produced (that is, flowers will be delivered to their mutual friend Robin). However, the method Elizabeth uses to satisfy the request (in all likelihood, simply passing the request on to Fred) will be different from that used by Fred in response to the same request.

If Chris were to ask Kenneth, a dentist, to send flowers to Robin, Kenneth may not have a method for solving that problem. If he understands the request at all, he will probably issue an appropriate error diagnostic.

Let us move our discussion back to the level of computers and programs. There, the distinction between message passing and procedure calling is that, in message passing, there is a designated receiver, and the interpretation–the selection of a method to execute in response to the message–may vary with different receivers. Usually, the specific receiver for any given message will not be known until run time, so the determination of which method to invoke cannot be made until then. Thus, we say there is late *binding* between the message (function or procedure name) and the code fragment (method) used to respond to the message. This situation is in contrast to the very early (compile-time or link-time) binding of name to code fragment in conventional procedure calls.

1.4.3 Responsibilities

A fundamental concept in object-oriented programming is to describe behavior in terms of *responsibilities*. Chris's request for action indicates only the desired outcome (flowers sent to Robin). Fred is free to pursue any technique that achieves the desired objective, and in doing so will not be hampered by interference from Chris.

By discussing a problem in terms of responsibilities we increase the level of abstraction. This permits greater *independence* between objects, a critical factor in solving complex problems. The entire collection of responsibilities associated with an object is often described by the term *protocol*.

A traditional program often operates by acting *on* data structures, for example changing fields in an array or record. In contrast, an object oriented program *requests* data structures (that is, objects) to perform a service. This difference between viewing software in traditional, structured terms and viewing it from an object-oriented perspective can be summarized by a twist on a well-known quote:

> Ask not what you can do *to* your data structures,
> but rather ask what your data structures can do *for* you.

1.4.4 Classes and Instances

Although Chris has only dealt with Fred a few times, Chris has a rough idea of the transaction that will occur inside Fred's flower shop. Chris is able to make certain assumptions based on previous experience with other florists, and hence Chris can expect that Fred, being an instance of this category, will fit the general pattern. We can use the term Florist to represent the category (or *class*) of all florists. Let us incorporate these notions into our next principle of object-oriented programming:

> All objects are *instances* of a *class*. The method invoked by an object in response to a message is determined by the class of the receiver. All objects of a given class use the same method in response to similar messages.

1.4.5 Class Hierarchies–Inheritance

Chris has more information about Fred–not necessarily because Fred is a florist but because he is a shopkeeper. Chris knows, for example, that a transfer of money will be part of the transaction, and that in return for payment Fred will offer a receipt. These actions are true of grocers, stationers, and other shopkeepers. Since the category Florist is a more specialized form of the category Shopkeeper, any knowledge Chris has of Shopkeepers is also true of Florists and hence of Fred.

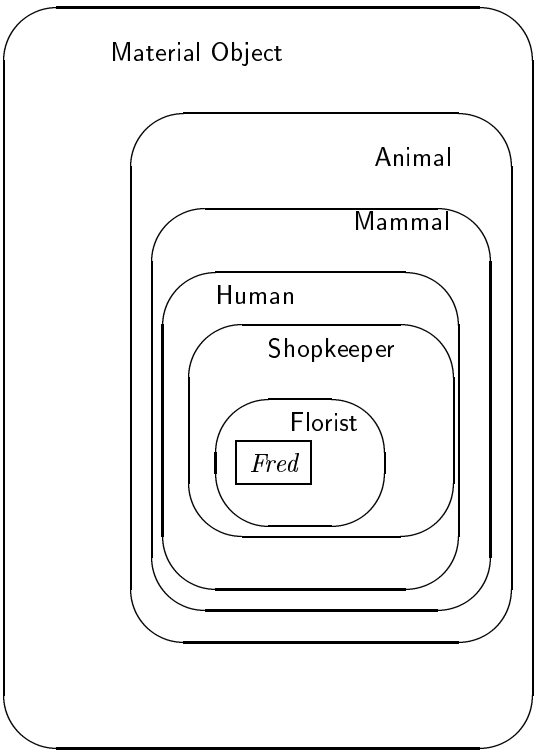


Figure 1.3: – The categories surrounding Fred.

One way to think about how Chris has organized knowledge of Fred is in terms of a hierarchy of categories (see Figure 1.3). Fred is a Florist, but Florist is a specialized form of Shopkeeper. Furthermore, a Shopkeeper is also a Human; so Chris knows, for example, that Fred is probably bipedal. A Human is a Mammal (therefore they nurse their young and have hair), and a Mammal is an Animal (therefore it breathes oxygen), and an Animal is a Material Object (therefore it has mass and weight). Thus, quite a lot of knowledge that Chris has that is applicable to Fred is not directly associated with him, or even with the category Florist.

The principle that knowledge of a more general category is also applicable to a more specific category is called *inheritance*. We say that the class Florist will inherit attributes of the class (or category) Shopkeeper.

There is an alternative graphical technique often used to illustrate this relationship, particularly when there are many individuals with differing lineage's. This technique shows classes listed in a hierarchical tree-like structure, with more abstract classes (such as Material Object or Animal) listed near the top of

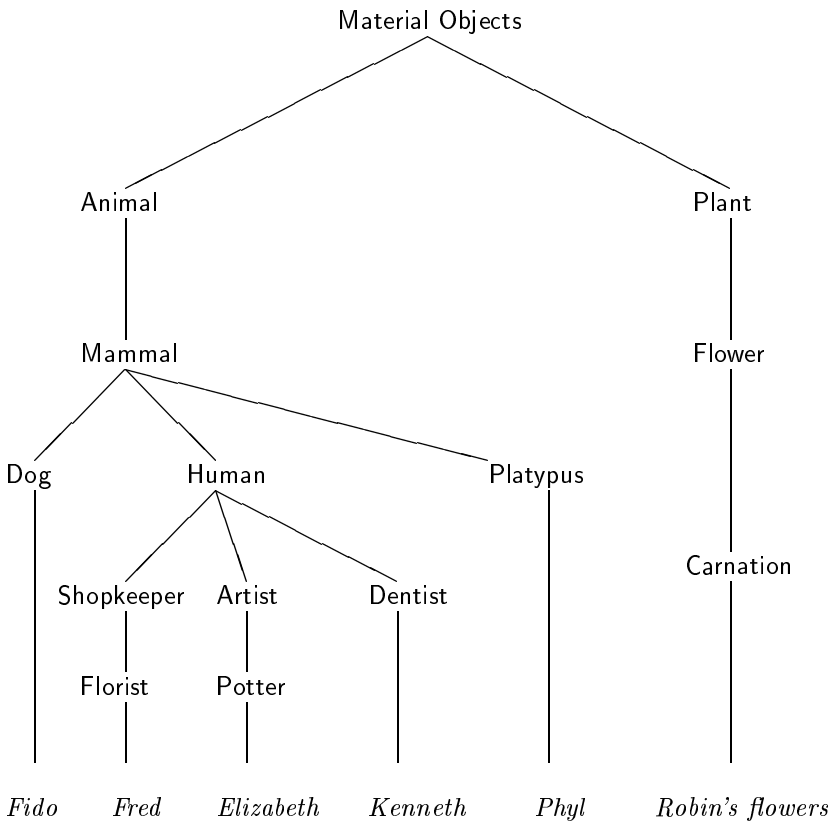


Figure 1.4: – A class hierarchy for various material objects.

the tree, and more specific classes, and finally individuals, are listed near the bottom. Figure 1.4 shows this class hierarchy for Fred. This same hierarchy also includes Elizabeth, Chris's dog Fido, Phyl the platypus who lives at the zoo, and the flowers the Chris is sending to Robin. Notice that the structure and interpretation of this type of diagram is similar to the biological hierarchy presented earlier in Figure 1.1.

Information that Chris possess about Fred because Fred is an instance of class Human is also applicable to Elizabeth, for example. Information that Chris knows about Fred because he is a Mammal is applicable to Fido as well. Information about all members of Material Object is equally applicable to Fred and to his flowers. We capture this in the idea of inheritance:

> Classes can be organized into a hierarchical *inheritance* structure. A *child class* (or *subclass*) will inherit attributes from a *parent class* higher in the tree. An *abstract parent class* is a class (such as Mam-

mal) for which there are no direct instances; it is used only to create subclasses.

1.4.6 Method Binding and Overriding

Phyl the platypus presents a problem for our simple organizing structure. Chris knows that mammals give birth to live children, and Phyl is certainly a Mammal, yet Phyl (or rather his mate Phyllis) lays eggs. To accommodate this, we need to find a technique to encode *exceptions* to a general rule.

We do this by decreeing that information contained in a subclass can *override* information inherited from a parent class. Most often, implementations of this approach takes the form of a method in a subclass having the same name as a method in the parent class, combined with a rule for how the search for a method to match a specific message is conducted:

> The search for a method to invoke in response to a given message begins with the *class* of the receiver. If no appropriate method is found, the search is conducted in the *parent class* of this class. The search continues up the parent class chain until either a method is found or the parent class chain is exhausted. In the former case the method is executed; in the latter case, an error message is issued. If methods with the same name can be found higher in the class hierarchy, the method executed is said to *override* the inherited behavior.

Even if a compiler cannot determine which method will be invoked at run time, in many object-oriented languages, such as Java, it can determine whether there will be an appropriate method and issue an error message as a compile-time error diagnostic rather than as a run-time message.

The fact that both Elizabeth and Fred will react to Chris's messages, but use different methods to respond, is one form of *polymorphism*. As explained, that Chris does not, and need not, know exactly what method Fred will use to honor the request is an example of *information hiding*.

1.4.7 Summary of Object-Oriented Concepts

Alan Kay, considered by some to be the father of object-oriented programming, identified the following characteristics as fundamental to OOP [Kay 1993]:

1. Everything is an *object*.

2. Computation is performed by objects communicating with each other, requesting that other objects perform actions. Objects communicate by sending and receiving *messages*. A message is a request for action bundled with whatever arguments may be necessary to complete the task.

3. Each object has its own *memory*, which consists of other objects.

4. Every object is an *instance* of a *class*. A class simply represents a grouping of similar objects, such as integers or lists.

5. The class is the repository for *behavior* associated with an object. That is, all objects that are instances of the same class can perform the same actions.

6. Classes are organized into a singly rooted tree structure, called the *inheritance hierarchy*. Memory and behavior associated with instances of a class are automatically available to any class associated with a descendant in this tree structure.

1.5 Computation as Simulation *

The view of programming represented by the example of sending flowers to a friend is very different from the conventional conception of a computer. The traditional model describing the behavior of a computer executing a program is a *process-state* or *pigeon-hole* model. In this view, the computer is a data manager, following some pattern of instructions, wandering through memory, pulling values out of various slots (memory addresses), transforming them in some manner, and pushing the results back into other slots (see Figure 1.5). By examining the values in the slots, one can determine the state of the machine or the results produced by a computation. Although this model may be a more or less accurate picture of what takes place inside a computer, it does little to help us understand how to solve problems using the computer, and it is certainly not the way most people (pigeon handlers and postal workers excepted) go about solving problems.

In contrast, in the object-oriented framework we never mention memory addresses, variables, assignments, or any of the conventional programming terms. Instead, we speak of objects, messages, and responsibility for some action. In Dan Ingalls's memorable phrase:

> Instead of a bit-grinding processor...plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires [Ingalls 1981].

Another author has described object-oriented programming as "animistic": a process of creating a host of helpers that form a community and assist the programmer in the solution of a problem [Actor 1987].

This view of programming as creating a "universe" is in many ways similar to a style of computer simulation called "discrete event-driven simulation." In brief, in a discrete event-driven simulation the user creates computer models of the various elements of the simulation, describes how they will interact with one

[0] Section headings followed by an asterisk indicate optional material.

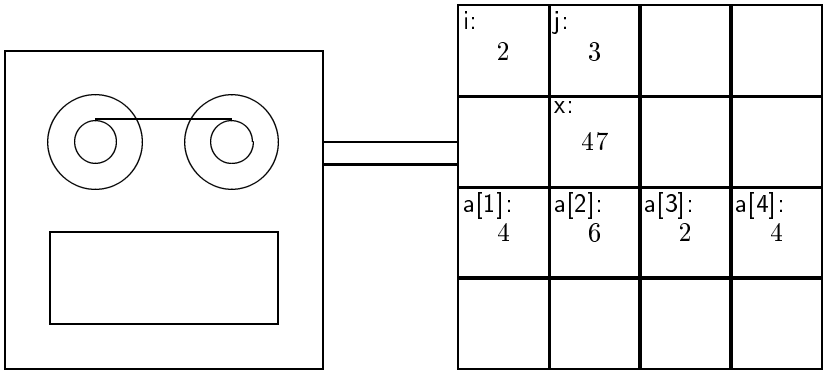


Figure 1.5: – Visualization of imperative programming.

another, and sets them moving. This is almost identical to the average object-oriented program, in which the user describes what the various entities in the universe for the program are, and how they will interact with one another, and finally sets them in motion. Thus, in object-oriented programming, we have the view that *computation is simulation* [Kay 1977].

1.5.1 The Power of Metaphor

An easily overlooked benefit to the use of object-oriented techniques is the power of *metaphor*. When programmers think about problems in terms of behaviors and responsibilities of objects, they bring with them a wealth of intuition, ideas, and understanding from their everyday experience. When envisioned as pigeon holes, mailboxes, or slots containing values, there is little in the programmer's background to provide insight into how problems should be structured.

Although anthropomorphic descriptions such as the quote by Ingalls may strike some people as odd, in fact they are a reflection of the great expositive power of metaphor. Journalists make use of metaphor every day, as in the following description of object-oriented programming from *Newsweek:*

> Unlike the usual programming method–writing software one line at a time–NeXT's "object-oriented" system offers larger building blocks that developers can quickly assemble the way a kid builds faces on Mr. Potato Head.

Possibly this feature, more than any other, is responsible for the frequent observation that it is sometimes easier to teach object-oriented programming concepts to computer novices than to computer professionals. Novice users quickly

Figure 1.6: Mr. Potato Head, an Object-Oriented Toy

adapt the metaphors with which they are already comfortable from their everyday life, whereas seasoned computer professionals can be blinded by an adherence to more traditional ways of viewing computation.

1.5.2 Avoiding Infinite Regression

Of course, objects cannot always respond to a message by politely asking another object to perform some action. The result would be an infinite circle of requests, like two gentlemen each politely waiting for the other to go first before entering a doorway, or like a bureaucracy of paper pushers, each passing on all papers to some other member of the organization. At some point, at least a few objects need to perform some work besides passing on requests to other agents. This work is accomplished differently in various object-oriented languages.

In blended object-oriented/imperative languages, such as C++, Object Pascal, and Objective-C, it is accomplished by methods written in the base (non-object-oriented) language. In more purely object-oriented languages, such as Smalltalk or Java, it is accomplished by "primitive" or "native" operations that are provided by the underlying system.

1.6 A Brief History *

It is commonly thought that object-oriented programming is a relatively recent phenomenon in computer science. To the contrary, in fact, almost all the major concepts we now associate with object-oriented programs, such as objects, classes, and inheritance hierarchies, were developed in the 1960's as part of a language called Simula, designed by researchers at the Norwegian Computing Center. Simula, as the name suggests, was a language inspired by problems involving the simulation of real life systems. However the importance of these constructs, even to the developers of Simula, was only slowly recognized [Nygaard 81].

In the 1970's Alan Kay organized a research group at Xerox PARC (the Palo Alto Research Center). With great prescience, Kay predicated the coming revolution in personal computing that was to develop nearly a decade later (see, for example, his 1977 article in *Scientific American* [Kay 1977]). Kay was concerned with discovering a programming language that would be understandable to non computer professionals, to ordinary people with no prior training in computer use.[1] He found in the notion of classes and computing as simulation a metaphor that could easily be understood by novice users, as he then demonstrated by a series of experiments conducted at PARC using children as programmers. The

[0] Section headings followed by an asterisk indicate optional material.

[1] I have always found it ironic that Kay missed an important point. He thought that to *use* a computer one would be required to *program* a computer. Although he correctly predicated in 1977 the coming trend in hardware, few could have predicated at that time the rapid development of general purpose computer applications that was to accompany, perhaps even drive, the introduction of personal computers. Nowadays the vast majority of people who use personal computers have no idea how to program.

programming language developed by his group was named Smalltalk. This language evolved through several revisions during the decade. A widely read 1981 issue of *Byte* magazine, in which the quote by Ingalls presented earlier appears, did much to popularize the concepts developed by Kay and his team at Xerox.

Roughly contemporaneous with Kays work was another project being conducted on the other side of the country. Bjarne Stroustrup, a researcher at Bell Laboratories who had learned Simula while completing his doctorate at Cambridge University in England, was developing an extension to the C language that would facilitate the creation of objects and classes [Stroustrup 82]. This was to eventually evolve into the language C++ [Stroustrup 1994].

With the dissemination of information on these and other similar projects, an explosion of research in object-oriented programming techniques began. By the time of the first major conference on object-oriented programming, in 1986, there were literally dozens of new programming languages vying for acceptance. These included Eiffel [Meyer 1988a], Objective-C [Cox 1986], Actor [Actor 1987], Object Pascal, and various Lisp dialects.

In the two decades since the 1986 OOPSLA conference, object-oriented programming has moved from being revolutionary to being mainstream, and in the process has transformed a major portion of the field of computer science as a whole.

Chapter Summary

- Object-oriented programming is not simply a few new features added to programming languages. Rather, it is a new way of *thinking* about the process of decomposing problems and developing programming solutions.
- Object-oriented programming views a program as a collection of loosely connected agents, termed *objects*. Each object is responsible for specific tasks. It is by the interaction of objects that computation proceeds. In a certain sense, therefore, programming is nothing more or less than the simulation of a model universe.
- An object is an encapsulation of *state* (data values) and *behavior* (operations). Thus, an object is in many ways similar to special purpose computer.
- The behavior of objects is dictated by the object *class*. Every object is an instance of some class. All instances of the same class will behave in a similar fashion (that is, invoke the same method) in response to a similar request.
- An object will exhibit its behavior by invoking a method (similar to executing a procedure) in response to a message. The interpretation of the message (that is, the specific method used) is decided by the object and may differ from one class of objects to another.

- Classes can be linked to each other by means of the notion of *inheritance*. Using inheritance, classes are organized into a hierarchical inheritance tree. Data and behavior associated with classes higher in the tree can also be accessed and used by classes lower in the tree. Such classes are said to inherit their behavior from the parent classes.
- Designing an object oriented program is like organizing a community of individuals. Each member of the community is given certain responsibilities. The achievement of the goals for the community as a whole come about through the work of each member, and the interactions of members with each other.
- By reducing the interdependency among software components, object-oriented programming permits the development of reusable software systems. Such components can be created and tested as independent units, in isolation from other portions of a software application.
- Reusable software components permit the programmer to deal with problems on a higher level of abstraction. We can define and manipulate objects simply in terms of the messages they understand and a description of the tasks they perform, ignoring implementation details.

Further Reading

I noted earlier that many consider Alan Kay to be the father of object-oriented programming. Like most simple assertions, this one is only somewhat supportable. Kay himself [Kay 1993] traces much of the influence on his development of Smalltalk to the earlier computer programming language Simula, developed in Scandinavia in the early 1960s [Dahl 1966, Kirkerud 1989]. A more accurate history would be that most of the principles of object-oriented programming were fully worked out by the developers of Simula, but that these would have been largely ignored by the profession had they not been rediscovered by Kay in the creation of the Smalltalk programming language. A widely read 1981 issue of *Byte* magazine did much to popularize the concepts developed by Kay and his team at Xerox PARC.

The term "software crisis" seems to have been coined by Doug McIlroy at a 1968 NATO conference on software engineering. It is curious that we have been in a state of crisis now for more than half the life of computer science as a discipline. Despite the end of the Cold War, the end of the software crisis seems to be no closer now than it was in 1968. See, for example, Gibb's article "Software's Chronic Crisis" in the September 1994 issue of *Scientific American* [Gibbs 1994].

To some extent, the software crisis may be largely illusory. For example, tasks considered exceedingly difficult five years ago seldom seem so daunting today. It is only the tasks that we wish to solve *today* that seem, in comparison, to be nearly impossible, which seems to indicate that the field of software development has, indeed, advanced steadily year by year.

The quote from the American linguist Edward Sapir is taken from "The Relation of Habitual Thought and Behavior to Language," reprinted in *Language, Thought and Reality* [Whorf 1956]. This book contains several interesting papers on the relationships between language and our habitual thinking processes. I urge any serious student of computer languages to read these essays; some of them have surprising relevance to artificial languages. (An undergraduate once exclaimed to me "I didn't know the Klingon was a linguist!").

Another interesting book along similar lines is *The Alphabet Effect* by Robert Logan [Logan 1986], which explains in terms of language why logic and science developed in the West while for centuries China had superior technology. In a more contemporary investigation of the effect of natural language on computer science, J. Marshall Unger [Unger 1987] describes the influence of the Japanese language on the much-heralded Fifth Generation project.

The commonly held observation that Eskimo languages have many words for snow was debunked by Geoffrey Pullum in his book of essays on linguistics [Pullum 1991]. In an article in the *Atlantic Monthly* ("In Praise of Snow" January 1995), Cullen Murphy pointed out that the vocabulary used to discuss snow among English speakers for whom a distinction between types of snow is important–namely, those who perform research on the topic–is every bit as large or larger than that of the Eskimo.

Those who would argue in favor of the Sapir-Whorf hypothesis have a difficult problem to overcome; namely, the simple question "Can you give me an example?" Either they can, which (since it must be presented in the language of the speaker), serves to undercut their argument. Or they cannot, which also weakens their argument. In any case, the point is irrelevant to our discussion. It is certainly true that groups of individuals with common interests tend to develop their own specialized vocabulary, and once developed, the vocabulary itself tends to direct their thoughts along paths that may not be natural to those outside the group. Such is the case with OOP. While object-oriented ideas can, with discipline, be used without an object-oriented language, the use of object-oriented terms helps direct the programmer's thought along lines that may not have been obvious without the OOP terminology.

My history is slightly imprecise with regard to Church's conjecture and Turing machines. Church actually conjectured about partial functions [Church 1936]; which were later shown to be equivalent to computations performed with Turing machines [Turing 1936]. Kleene described the conjecture in the form we have here, also giving it the name by which it has become known. Rogers gives a good summary of the arguments for the equivalence of various computational models [Rogers 1967].

Information on the history of Smalltalk can be found in Kays article from the History of Programming Languages conference [Kay 1993]. Bjarne Stroustrup has provided a history of C++ [Stroustrup 1994]. A more general history of OOP is presented in The Handbook of Programming Languages [Salus 1998].

Like most terms that have found their way into the popular jargon, *object-oriented* is used more often than it is defined. Thus, the question What is object-

oriented programming? is surprisingly difficult to answer. Bjarne Stroustrup has quipped that many arguments appear to boil down to the following syllogism:

- X is good.
- Object-oriented is good.
- *Ergo,* X is object-oriented [Stroustrup 1988].

Roger King argued [Kim 1989], that his cat is object-oriented. After all, a cat exhibits characteristic behavior, responds to messages, is heir to a long tradition of inherited responses, and manages its own quite independent internal state.

Many authors have tried to provide a precise description of the properties a programming language must possess to be called *object-oriented.* See, for example, the analysis by Josephine Micallef [Micallef 1988], or Peter Wegner [Wegner 1986]. Wegner, for example, distinguishes *object-based* languages, which support only abstraction (such as Ada), from *object-oriented* languages, which must also support inheritance.

Other authors–notably Brad Cox [Cox 1990]–define the term much more broadly. To Cox, object-oriented programming represents the *objective* of programming by assembling solutions from collections of off-the-shelf subcomponents, rather than any particular *technology* we may use to achieve this objective. Rather than drawing lines that are divisive, we should embrace any and all means that show promise in leading to a new software Industrial Revolution. Cox's book on OOP [Cox 1986], although written early in the development of object-oriented programming and now somewhat dated in details, is nevertheless one of the most readable manifestos of the object-oriented movement.

Self Study Questions

1. What is the original meaning of the word paradigm?
2. How do objects interact with each other?
3. How are messages different from procedure calls?
4. What is the name applied to describe an algorithm an object uses to respond to a request?
5. Why does the object-oriented approach naturally imply a high degree of information hiding?
6. What is a class? How are classes linked to behavior?
7. What is a class inheritance hierarchy? How is it linked to classes and behavior?
8. What does it mean for one method to override another method from a parent class?

9. What are the basic elements of the process-state model of computation?

10. How does the object-oriented model of computation differ from the process-state model?

11. In what way is a object oriented program like a simulation?

Exercises

1. In an object-oriented inheritance hierarchy, each level is a more specialized form of the preceding level. Give an example of a hierarchy found in everyday life that has this property. Some types of hierarchy found in everyday life are not inheritance hierarchies. Give an example of a noninheritance hierarchy.

2. Look up the definition of *paradigm* in at least three dictionaries. Relate these definitions to computer programming languages.

3. Take a real-world problem, such as the task of sending flowers described earlier, and describe its solution in terms of agents (objects) and responsibilities.

4. If you are familiar with two or more distinct computer programming languages, give an example of a problem showing how one language would direct the programmer to one type of solution, and a different language would encourage an alternative solution.

5. If you are familiar with two or more distinct natural languages, describe a situation that illustrates how one language directs the speaker in a certain direction, and the other language encourages a different line of thought.

6. Argue either for or against the position that computing is basically simulation. (You may want to read the *Scientific American* [Kay 1977] article by Kay cited earlier.)

Chapter 2

Abstraction

If you open an atlas you will often first see a map of the world. This map will show only the most significant features. For example, it may show the various mountain ranges, the ocean currents, and other extremely large structures. But small features will almost certainly be omitted.

A subsequent map will cover a smaller geographical region, and will typically possess more detail. For example, a map of a single continent (such as South America) may now include political boundaries, and perhaps the major cities. A map over an even smaller region, such as a country, might include towns as well as cities, and smaller geographical features, such as the names of individual mountains. A map of an individual large city might include the most important roads leading into and out of the city. Maps of smaller regions might even represent individual buildings.

Notice how, at each level, certain information has been included, and certain information has been purposely omitted. There is simply no way to represent all the details when an artifact is viewed at a higher level of abstraction. And even if all the detail could be described (using tiny writing, for example) there is no way that people could assimilate or process such a large amount of information. Hence details are simply left out.

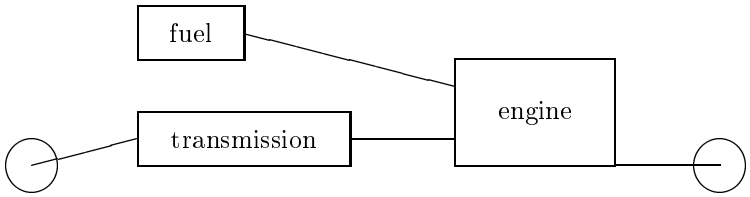
Fundamentally, people use only a few simple tools to create, understand, or manage complex systems. One of the most important techniques is termed *abstraction*.

Abstraction

Abstraction is the purposeful suppression, or hiding, of some details of a process or artifact, in order to bring out more clearly other aspects, details, or structure.

Consider the average persons understanding of an automobile. A laymans view of an automobile engine, for example, is a device that takes fuel as input and produces a rotation of the drive shaft as output. This rotation is too fast to

connect to the wheels of the car directly, so a transmission is a mechanism used to reduce a rotation of several thousand revolutions per minute to a rotation of several revolutions per minute. This slower rotation can then be used to propel the car. This is not exactly correct, but it is sufficiently close for everyday purposes. We sometimes say that by means of abstraction we have constructed a *model* of the actual system.



In forming an abstraction, or model, we purposely avoid the need to understand many details, concentrating instead of a few key features. We often describe this process with another term, *information hiding*.

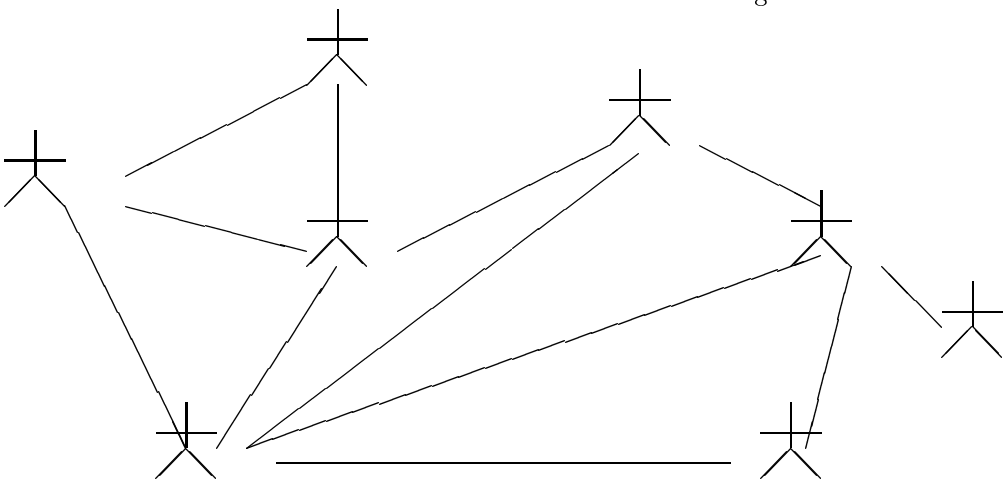
Information Hiding

Information hiding is the purposeful omission of details in the development of an abstract representation.

2.1 Layers of Abstraction

In a typical program written in the object-oriented style there are many important levels of abstraction. The higher level abstractions are part of what makes an object-oriented program object-oriented.

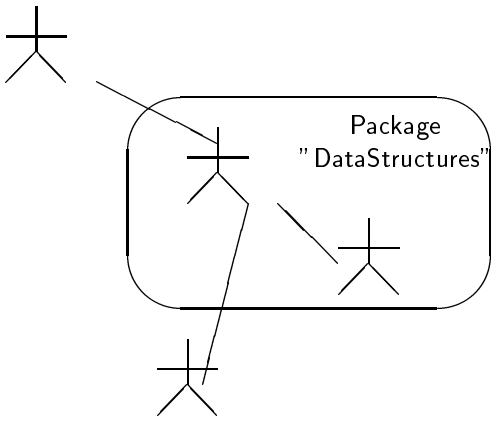
At the highest level a program is viewed as a "community" of objects that must interact with each other in order to achieve their common goal:



This notion of community finds expression in object-oriented development in two distinct forms. First there is the community of programmers, who must interact with each other in the real world in order to produce their application. And second there is the community of objects that they create, which must interact with each other in a virtual universe in order to further their common goals. Key ideas such as information hiding and abstraction are applicable to both levels.

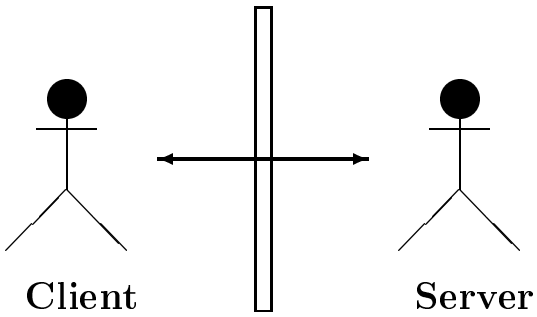
Each object in this community provides a service that is used by other members of the organization. At this highest level of abstraction the important features to emphasize are the lines of communication and cooperation, and the way in which the members must interact with each other.

The next level of abstraction is not found in all object-oriented programs, nor is it supported in all object-oriented languages. However, many languages permit a group of objects working together to be combined into a unit. Examples of this idea include *packages* in Java, *name spaces* in C++, or *units* in Delphi. The unit allows certain names to be exposed to the world outside the unit, while other features remain hidden inside the unit.



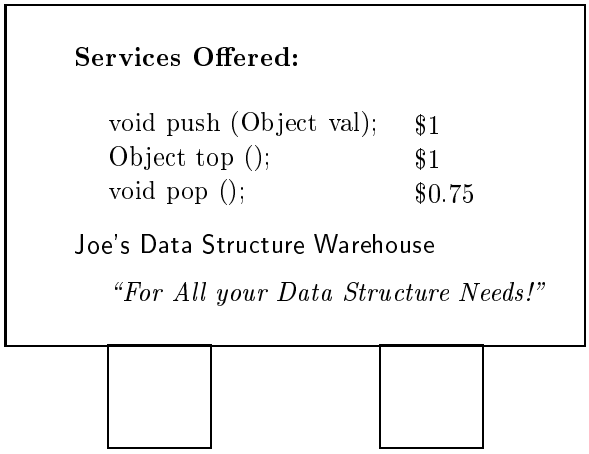
For readers familiar with concepts found in earlier languages, this notion of a unit is the heir to the idea of a *module* in languages such as C or Modula. Later in this chapter we will present a short history of programming language abstractions, and note the debt that ideas of object-oriented programming owe to the earlier work on modules.

The next two levels of abstraction deal with the interactions between two individual objects. Often we speak of objects as proving a *service* to other objects. We build on this intuition by describing communication as an interaction between a *client* and a *server*.



We are not using the term *server* in the technical sense of, say, a web server. Rather, here the term server simply means an object that is providing a service. The two layers of abstraction refer to the two views of this relationship; the view from the client side and the view from the server side.

In a good object-oriented design we can describe and discuss the services that the server provides without reference to any actions that the client may perform in using those services. One can think of this as being like a billboard advertisement:



The billboard describes, for example, the services provided by a data structure, such as a Stack. Often this level of abstraction is represented by an interface, a class-like mechanism that defines behavior without describing an implementation:

```
interface Stack {
    public void push (Object val);
    public Object top () throws EmptyStackException;
    public void pop () throws EmptyStackException;
}
```

Finding the Right Level of Abstraction

In early stages of software development a critical problem is finding the right level of abstraction. A common error is to dwell on the lowest levels, worrying about the implementation details of various key components, rather than striving to ensure that the high level organizational structure promotes a clean separation of concerns.

The programmer (or, in larger projects, the design team) must walk a fine line in trying to identify the right level of abstraction at any one point of time. One does not want to ignore or throw away too much detail about a problem, but also one must not keep so much detail that important issues become obscured.

The next level of abstraction looks at the same boundary but from the server side. This level considers a concrete implementation of the abstract behavior. For example, there are any number of data structures that can be used to satisfy the requirements of a Stack. Concerns at this level deal with the way in which the services are being realized.

```
public class LinkedList implements Stack ... {
    public void pop () throws  EmptyStackException { ... }
    ...
}
```

Finally, the last level of abstraction considers a single task in isolation; that is, a single method. Concerns at this level of abstraction deal with the precise sequence of operations used to perform just this one activity. For example, we might investigate the technique used to perform the removal of the most recent element placed into a stack.

```
public class LinkedList implements Stack ... {
    ...
    public void pop () throws  EmptyStackException {
        if (isEmpty())
            throw new EmptyStackException();
        removeFirst(); // delete first element of list
    }
    ...
}
```

Each level of abstraction is important at some point during software development. In fact, programmers are often called upon to quickly move back and forth between different levels of abstraction. We will see analysis of object-oriented

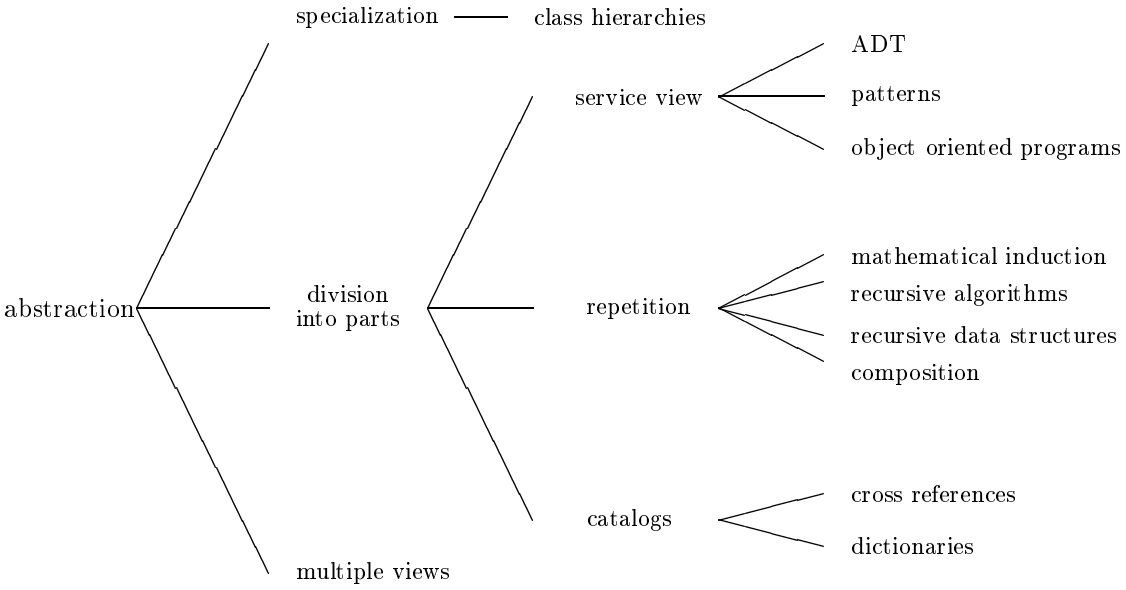


Figure 2.1: Some Techniques for Handling Complexity, with Examples

programs performed at each of these levels of abstraction as we proceed through the book.

2.2 Other Forms of Abstraction

Abstraction is used to help understand a complex system. In a certain sense, abstraction is the imposition of structure on a system. The structure we impose may reflect some real aspects of the system (a car really does have both an engine and a transmission) or it may simply be a mental abstraction we employ to aid in our understanding.

This idea of abstraction can be further subdivided into a variety of different forms (Figure 2.1). A common technique is to divide a layer into constituent parts. This is the approach we used when we described an automobile as being composed of the engine, the transmission, the body and the wheels. The next level of understanding is then achieved by examining each of these parts in turn. This is nothing more than the application of the old maxim *divide and conquer*.

Other times we use different types of abstraction. Another form is the idea of layers of specialization (Figure 2.2). An understanding of an automobile is based, in part, on knowledge that it is a *wheeled vehicle*, which is in turn a *means of transportation*. There is other information we know about wheeled vehicles, and that knowledge is applicable to both an automobile and a bicycle. There is other knowledge we have about various different means of transportation, and that information is also applicable to pack horses as well as bicycles. Object

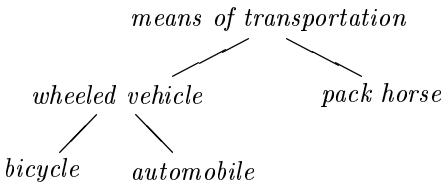


Figure 2.2: Layers of Specialization

Is-a and Has-a Abstraction

The ideas of division into parts and division into specializations represent the two most important forms of abstraction used in object-oriented programming. These are commonly known as *is-a* and *has-a* abstraction.

The idea of division into parts is has-a abstraction. The meaning of this term is easy to understand; a car "has-a" engine, and it "has-a" transmission, and so on.

The concept of specialization is referred to as "is-a" abstraction. Again, the term comes from the English sentences that can be used to illustrate the relationships. A bicycle "is-a" wheeled vehicle, which in turn "is-a" means of transportation.

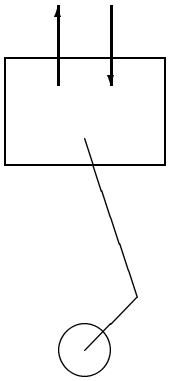
Both is-a and has-a abstractions will reappear in later chapters and be tied to specific programming language features.

oriented languages make extensive use of this form of abstraction.

Yet another form of abstraction is to provide multiple views of the same artifact. Each of the views can emphasize certain detail and suppress others, and thus bring out different features of the same object. A laymans view of a car, for example, is very different from the view required by a mechanic.

2.2.1 Division into Parts

The most common technique people use to help understand complex systems is to combine abstraction with a division into component parts. Our description of an automobile is an example of this. The next level of understanding is then achieved by taking each of the parts, and performing the same sort of analysis at a finer level of detail. A slightly more precise description of an engine, for example, views it as a collection of cylinders, each of which converts an explosion of fuel into a vertical motion, and a crankshaft, which converts the up and down motion of the cylinder into a rotation.



Another example might be organizing information about motion in a human body. At one level we are simply concerned with mechanics, and we consider the body as composed of bone (for rigidity), muscles (for movement), eyes and ears (for sensing), the nervous system (for transferring information) and skin (to bind it all together). At the next level of detail we might ask how the muscles work, and consider issues such as cell structure and chemical actions. But chemical actions are governed by their molecular structure. And to understand molecules we break them into their individual atoms.

Any explanation must be phrased at the right level of abstraction; trying to explain how a person can walk, for example, by understanding the atomic level details is almost certainly difficult, if not impossible.

2.2.2 Encapsulation and Interchangeability

A key step in the creation of large systems is the division into components. Suppose instead of writing software, we are part of a team working to create a new automobile. By separating the automobile into the parts *engine* and *transmission*, it is possible to assign people to work on the two aspects more or less independently of each other. We use the term *encapsulation* to mean that there is a strict division between the inner and the outer view; those members of the team working on the engine need only an abstract (outside, as it were) view of the transmission, while those actually working on the transmission need the more detailed inside view.

An important benefit of encapsulation is that it permits us to consider the possibility of *interchangeability*. When we divide a system into parts, a desirable goal is that the interaction between the parts is kept to a minimum. For example, by encapsulating the behavior of the engine from that of a transmission we permit the ability to exchange one type of engine with another without incurring an undue impact on the other portions of the system.

For these ideas to be applicable to software systems, we need a way to discuss the task that a software component performs, and separate this from the way in which the component fulfills this responsibility.

Catalogs

When the number of components in a system becomes large it is often useful to organize the items by means of a catalog. We use many different forms of catalog in everyday life. Examples include a telephone directory, a dictionary, or an internet search engine. Similarly, there are a variety of different catalogs used in software. One example is a simple list of classes. Another catalog might be the list of methods defined by a class. A reference book that describes the classes found in the Java standard library is a very useful form of catalog. In each of these cases the idea is to provide the user a mechanism to quickly locate a single part (be it class, object, or method) from a larger collection of items.

2.2.3 Interface and Implementation

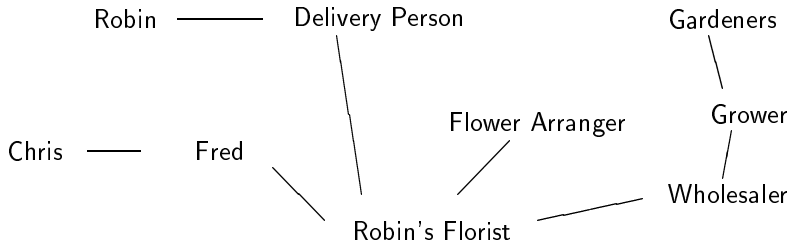
In software we use the terms *interface* and *implementation* to describe the distinction between the *what* aspects of a task, and the *how* features; between the outside view, and the inside view. An interface describes what a system is designed to do. This is the view that *users* of the abstraction must understand. The interface says nothing about how the assigned task is being performed. So to work, an interface is matched with an *implementation* that completes the abstraction. The designers of an engine will deal with the interface to the transmission, while the designers of the transmission must complete an implementation of this interface.

Similarly, a key step along the path to developing complex computer systems will be the division of a task into component parts. These parts can then be developed by different members of a team. Each component will have two faces, the interface that it shows to the outside world, and an implementation that it uses to fulfill the requirements of the interface.

The division between interface and implementation not only makes it easier to understand a design at a high level (since the description of an interface is much simpler than the description of any specific implementation), but also make possible the interchangeability of software components (as I can use any implementation that satisfies the specifications given by the interface).

2.2.4 The Service View

The idea that an interface describes the service provided by a software component without describing the techniques used to implement the service is at the heart of a much more general approach to managing the understanding of complex software systems. It was this sort of abstraction that we emphasized when we described the flower story in Chapter 1. Ultimately in that story a whole community of people became involved in the process of sending flowers:



Each member of the community is providing a service that is used by other members of the group. No member could solve the problem on their own, and it is only by working together that the desired outcome is achieved.

2.2.5 Composition

Composition is another powerful technique used to create complex structures out of simple parts. The idea is to begin with a few primitive forms, and add rules for combining forms to create new forms. The key insight in composition is to permit the combination mechanism to be used both on the new forms as well as the original primitive forms.

A good illustration of this technique is the concept of regular expressions. Regular expressions are a simple technique for describing sets of values, and have been extensively studied by theoretical computer scientists. The description of a regular expression begins by identifying a basic alphabet, for example the letters a, b, c and d. Any single example of the alphabet is a regular expression. We next add a rule that says the composition of two regular expressions is a regular expression. By applying this rule repeatedly we see that any finite string of letters is a regular expression:

abaccaba

The next combining rule says that the alternation (represented by the vertical bar |) of two regular expressions is a regular expression. Normally we give this rule a lower precedence that composition, so that the following pattern represents the set of three letter values that begin with ab, and and end with either an a, c or d:

aba | abc | abd

Parenthesis can be used for grouping, so that the previous set can also be described as follows:

ab(a|c|d)

Finally the $*$ symbol (technically known as the kleene-star) is used to represent the concept "zero or more repetitions". By combining these rules we can describe quite complex sets. For example, the following describes the set of values that begin with a run of a's and b's followed by a single c, or a two character sequence dd, followed by the letter a.

(((a|b)*c)|dd)a

This idea of composition is also basic to type systems. We begin with the primitive types, such as int and boolean. The idea of a class then permits the user to create new types. These new types can include data fields constructed out of previous types, either primitive or user-defined. Since classes can build on previously defined classes, very complex structure can be constructed piece by piece.

```
class  Box { // a box is a new data type
    ...
    private int value; // built out of the existing type int
}
```

Yet another application of the principle of composition is the way that many user interface libraries facilitate the layout of windows. A window is composed from a few simple data types, such as buttons, sliders, and drawing panels. Various different types of layout managers create simple structures. For example, a grid layout defines a rectangular grid of equal sized components, a border layout manger permits the specification of up to five components in the north, south, east, west, and center of a screen. As with regular expressions, the key is that windows can be structured as part of other windows. Imagine, for example, that we want to define a window that has three sliders on the left, a drawing panel in the middle, a bank of sixteen buttons organized four by four on the right, and a text output box running along the top. (We will develop just such an application in Chapter 22. A screen shot is shown in Figure 22.4.) We can do this by laying simple windows inside of more complex windows (Figure 2.3).

Many computer programs can themselves be considered a product of composition, where the method or procedure call is the mechanism of composition. We begin with the primitive statements in the language (assignments and the like). With these we can develop a library of useful functions. Using these functions as new primitives, we can then develop more complex functions. We continue, each layer being built on top of earlier layers, until eventually we have the desired application.

2.2.6 Layers of Specialization

Yet another approach to dealing with complexity is to structure abstraction using layers of specialization. This is sometimes referred to as a *taxonomy*. For example, in biology we divide living things into animals and plants. Living things are then divided into vertebrates and invertebrates. Vertebrates eventually includes mammals, which can be divided into (among other categories) cats and dogs, and so on.

The key difference between this and the earlier abstraction is that the more specialized layers of abstraction (for example, a cat) is indeed a representative

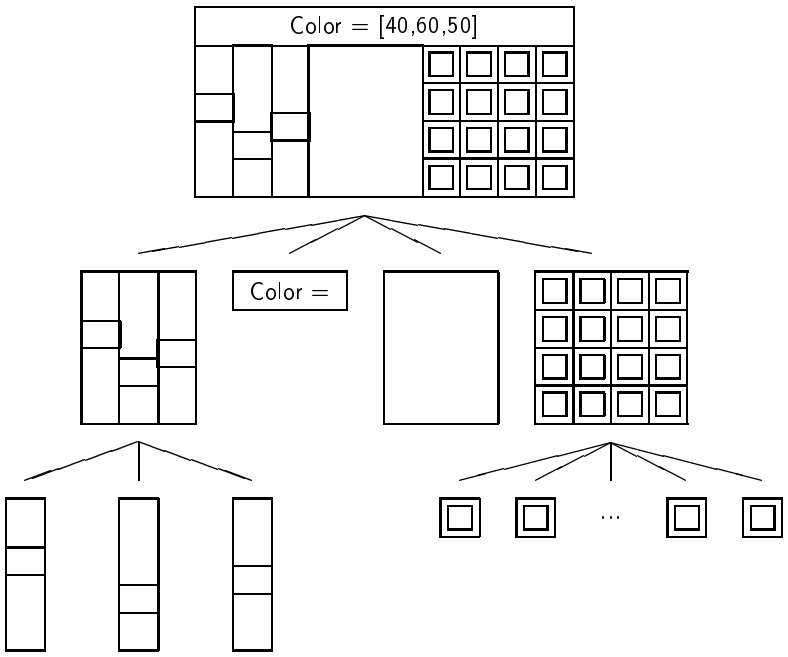


Figure 2.3: Composition in the Creation of User Interfaces

Nonstandard Behavior

Phyl and his friends remind us that there are almost never generalizations without their being exceptions. A platypus (such as phyl) is a mammal that lays eggs. Thus, while we might associate the tidbit of knowledge "gives birth to live young" with the category Mammal, we then need to amend this with the caveat "lays eggs" when we descend to the category Platypus.

Object-oriented languages will also need a mechanism to *override* information inherited from a more general category. We will explore this in more detail once we have developed the idea of class hierarchies.

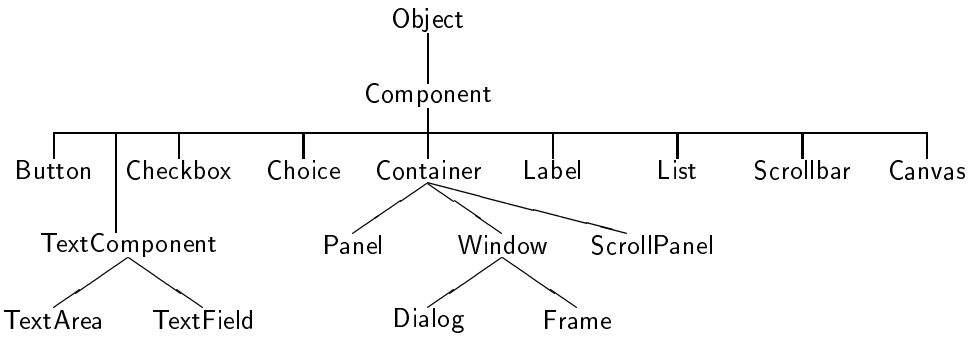


Figure 2.4: The AWT class hierarchy

of the more general layer of abstraction (for example, an animal). This was not true when, in an earlier example, we descended from the characterization of a muscle to the description of different chemical interactions. These two different types of relations are sometimes described using the heuristic keywords "is-a" and "has-a". The first relationship, that of parts to a whole, is a has-a relation, as in the sentence "a car has an engine". In contrast, the specialization relation is described using is-a, as in "a cat is a mammal".

But in practice our reason for using either type of abstraction is the same. The principle of abstraction permits us to suppress some details so that we can more easily characterize a fewer number of features. We can say that mammals are animals that have hair and nurse their young, for example. By associating this fact at a high level of abstraction, we can then apply the information to all more specialized categories, such as cats and dogs.

The same technique is used in object-oriented languages. New interfaces can be formed from existing interfaces. A class can be formed using inheritance from an existing class. In doing so, all the properties (data fields and behavior) we associate with the original class become available to the new class.

In a case study later in this book we will examine the Java AWT (Abstract Windowing Toolkit) library. When a programmer creates a new application using the AWT they declare their main class as a subclass of Frame, which in turn is linked to many other classes in the AWT library (Figure 2.4). A Frame is an special type of application window, but it is also a more specialized type of the general class Window. A Window can hold other graphical objects, and is hence a type of Container. Each level of the hierarchy provides methods used by those below. Even the simplest application will likely use the following:

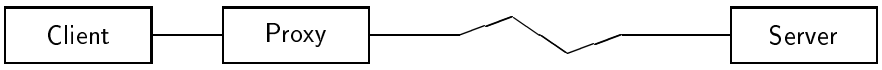
setTitle(String)	inherited from class Frame
setSize(int, int)	inherited from class Component
show()	inherited from class Window
repaint()	inherited from class Component
paint()	inherited from Component, then overridden in the programmers new application class

2.2.7 Patterns

When faced with a new problem, most people will first look to previous problems they have solved that seem to have characteristics in common with the new task. These previous problems can be used as a model, and the new problem attacked in a similar fashion, making changes as necessary to fit the different circumstances.

This insight lies behind the idea of a software *pattern*. A pattern is nothing more than an attempt to document a proven solution to a problem so that future problems can be more easily handled in a similar fashion. In the object-oriented world this idea has been used largely to describe patterns of interaction between the various members of an object community.

A simple example will illustrate this idea of a pattern. Imagine one is developing an application that will operate over a network. That means that part of the application will run on one computer, and part will run on another computer linked by a network connection. Creating the actual connection between the two computers, and transmitting information along this connection, are details that are perhaps not relevant to large portions of the application. One way to structure these relationships is to use a type of pattern termed a *proxy*. The proxy is an intermediary that hides the network connection. Objects can interact with the proxy, and not be aware that any type of network connection is involved at all. When the proxy receives a request for data or action, it bundles the request as a package, transmits the package over the network, recieves the response, unpackages the response and hands it back to the client. In this fashion the client is completely unaware of the details of the network protocol.



Notice how the description of the pattern has captured certain salient points of the interaction (the need to hide the communication protocol from the client) while omitting many other aspects of the interaction (for example, the particular information being communicated between client and server). We will have more to say about patterns later in Chapter 24.

2.3 A Short History of Abstraction Mechanisms*

Each of the abstraction mechanisms we have described in this chapter was the end product of a long process of searching for ways to deal with complexity. Another way to appreciate the role of object-oriented programming is to quickly review the history of mechanisms that computer languages have used to manage complexity. When seen in this perspective, object-oriented techniques are not at all revolutionary, but are rather a natural outcome of a progression from procedures, to modules, to abstract data types, and finally to objects.

2.3.1 Assembly Language

The techniques used to control the first computers were hardly what we would today term a language. Memory locations were described by address (i.e., location 372), not by name or purpose. Operations were similarly described by a numeric operation code. For example, an integer addition might be written as opcode 33, an integer subtraction as opcode 35. The following program might add the contents of location 372 to that of 376, then subtract from the result the value stored in location 377:

```
33   372   376
35   377   376
...  ...   ...
```

One of the earliest abstraction mechanisms was the creation of an assembler; a tool that could take a program written in a more human-friendly form, and translate it into a representation suitable for execution by the machine. The assembler permitted the use of symbolic names. The previous instructions might now be written as follows:

```
ADDI   a,x
SUBI   b,x
...    ...
```

This simple process was the first step in the long process of abstraction. Abstraction allowed the programmer to concentrait more effort on defining the task to be performed, and less on the steps necessary to complete the task.

2.3.2 Procedures

Procedures and functions represent the next improvement in abstraction in programming langauges. Procedures allowed tasks that were executed repeatedly, or executed with only slight variations, to be collected in one place and reused rather than being duplicated several times. In addition, the procedure gave the first possibility for *information hiding*. One programmer could write a procedure, or a set of procedures, that was used by many others. Other programmers

[0] Section headings followed by an asterisk indicate optional material.

```
int datastack[100];
int datatop = 0;

void init()
{
    datatop = 0;
}

void push(int val)
{
    if (datatop < 100)
        datastack [datatop++] = val;
}

int top()
{
    if (datatop > 0)
        return datastack [datatop - 1];
    return 0;
}

int pop()
{
    if (datatop > 0)
        return datastack [--datatop];
    return 0;
}
```

Figure 2.5: – Failure of procedures in information hiding.

did not need to know the exact details of the implementation–they needed only the necessary interface. But procedures were not an answer to all problems. In particular, they were not an effective mechanism for information hiding, and they only partially solved the problem of multiple programmers making use of the same names.

Example–A Stack

To illustrate these problems, we can consider a programmer who must write a set of routines to implement a simple stack. Following good software engineering principles, our programmer first establishes the visible interface to her work–say, a set of four routines: init, push, pop, and top. She then selects some suitable implementation technique. There are many choices here, such as an array with

a top-of-stack pointer, a linked list, and so on. Our intrepid programmer selects from among these choices, then proceeds to code the utilities, as shown in Figure 2.5.

It is easy to see that the data contained in the stack itself cannot be made local to any of the four routines, since they must be shared by all. But if the only choices are local variables or global variables (as they are in early programming languages, such as FORTRAN, or in C prior to the introduction of the static modifier), then the stack data must be maintained in global variables. However, if the variables are global, there is no way to limit the accessibility or visibility of these names. For example, if the stack is represented in an array named datastack, this fact must be made known to all the other programmers since they may want to create variables using the same name and should be discouraged from doing so. This is true even though these data values are important only to the stack routines and should not have any use outside of these four procedures. Similarly, the names init, push, pop, and top are now reserved and cannot be used in other portions of the program for other purposes, even if those sections of code have nothing to do with the stack routines.

2.3.3 Modules

The solution to the problem of global name space congestion was the introduction of the idea of a module. In one sense, modules can be viewed simply as an improved technique for creating and managing collections of names and their associated values. Our stack example is typical, in that there is some information (the interface routines) that we want to be widely and publicly available, whereas there are other data values (the stack data themselves) that we want restricted. Stripped to its barest form, a *module* provides the ability to divide a name space into two parts. The *public* part is accessible outside the module; the *private* part is accessible only within the module. Types, data (variables), and procedures can all be defined in either portion. A module encapsulation of the Stack abstraction is shown in Figure 2.6.

David Parnas, who popularized the notion of modules, described the following two principles for their proper use:

1. One must provide the intended user with all the information needed to use the module correctly, and *nothing more*.

2. One must provide the implementor with all the information needed to complete the module, and *nothing more*.

The philosophy is much like the military doctrine of "need to know"; if you do not need to know some information, you should not have access to it. This explicit and intentional concealment of information is what we have been calling *information hiding*.

Modules solve some, but not all, of the problems of software development. For example, they will permit our programmer to hide the implementation details of her stack, but what if the other users want to have two (or more) stacks?

```
module StackModule;
    export push, pop, top; (* the public interface *)

    var
        (* since data values are not exported, they are hidden *)
        datastack : array [ 1 .. 100 ] of integer;
        datatop : integer;

    procedure push(val : integer)...

    procedure top : integer ...

    procedure pop : integer ...

    begin      (* can perform initialization here *)
        datatop = 0;
    end;
end StackModule.
```

Figure 2.6: A Module for the Stack Abstraction

As a more extreme example, suppose a programmer announces that he has developed a new type of numeric abstraction, called Complex. He has defined the arithmetic operations for complex numbers–addition, subtraction, multiplication, and so on, and has defined routines to convert numbers from conventional to complex. There is just one small problem: only one complex number can be manipulated.

The complex number system would not be useful with this restriction, but this is just the situation in which we find ourselves with simple modules. Modules by themselves provide an effective method of information hiding, but they do not allow us to perform *instantiation*, which is the ability to make multiple copies of the data areas. To handle the problem of instantiation, computer scientists needed to develop a new concept.

2.3.4 Abstract Data Types

The development of the notion of an abstract data type was driven, in part, by two important goals. The first we have identified already. Programmers should be able to define their own new data abstractions that work much like the primitive system provided data types. This includes giving clients the ability to create multiple instances of the data type. But equally important, clients should be able to use these instances knowing only the operations that have been provided, without concern for how those operations were supported.

An *abstract data type* is defined by an abstract specification. The specification for our stack data type might list, for example, the trio of operations push, pop

and top. Matched with the ADT will be one or more different implementations. There might be several different implementation techniques for our stack; for example one using an array and another using a linked list. As long as the programmer restricts themselves to only the abstract specification, any valid implementation should work equally well.

The important advance in the idea of the ADT is to finally separate the notions of interface and implementation. Modules are frequently used as an implementation technique for abstract data types, although we emphasize that modules are an implementation technique and that the abstract data type is a more theoretical concept. The two are related but are not identical. To build an abstract data type, we must be able to:

1. Export a type definition.
2. Make available a set of operations that can be used to manipulate instances of the type.
3. Protect the data associated with the type so that they can be operated on only by the provided routines.
4. Make multiple instances of the type.

As we have defined them, modules serve only as an information-hiding mechanism and thus directly address only list items 2 and 3, although the others can be accommodated via appropriate programming techniques. *Packages*, found in languages such as CLU and Ada, are an attempt to address more directly the issues involved in defining abstract data types.

In a certain sense, an object is simply an abstract data type. People have said, for example, that Smalltalk programmers write the most "structured" of all programs because they cannot write anything but definitions of abstract data types. It is true that an object definition is an abstract data type, but the notions of object-oriented programming build on the ideas of abstract data types and add to them important innovations in code sharing and reusability.

2.3.5 A Service-Centered View

Assembly language and procedures as abstraction mechanisms concentrated the programmers view at the functional level–how a task should be accomplished. The movement towards modules and ADT are indicative of a shift from a function-centered conception of computation to a more data-centered view. Here it is the data values that are important, their structure, representation and manipulation.

Object-oriented programming starts from this data-centered view of the world and takes it one step further. It is not that data abstractions, per se, are important to computation. Rather, an ADT is a useful abstraction because it can be defined in terms of the *service* it offers to the rest of a program. Other types of abstractions can be similarly defined, not in terms of their particular actions or their data values, but in terms of the services they provide.

Assembly Language	*Function*
Functions and Procedures	Centered View
Modules	*Data*
Abstract Data Types	Centered View
Object-Oriented	*Service*
Programming	Centered View

Thus, object-oriented programming represents a third step in this sequence. From function centered, to data centered, and finally to service centered view of how to structure a computer program.

2.3.6 Messages, Inheritance, and Polymorphism

In addition to this service-centered view of computing, object-oriented programming adds several important new ideas to the concept of the abstract data type. Foremost among these is *message passing*. Activity is initiated by a *request* to a specific object, not by the invoking of a function.

Implicit in message passing is the idea that the *interpretation* of a message can vary with different objects. That is, the behavior and response that the message elicit will depend upon the object receiving it. Thus, `push` can mean one thing to a stack, and a very different thing to a mechanical-arm controller. Since names for operations need not be unique, simple and direct forms can be used, leading to more readable and understandable code.

Finally, object-oriented programming adds the mechanisms of *inheritance* and *polymorphism*. Inheritance allows different data types to share the same code, leading to a reduction in code size and an increase in functionality. Polymorphism allows this shared code to be tailored to fit the specific circumstances of individual data types. The emphasis on the independence of individual components permits an incremental development process in which individual software units are designed, programmed, and tested before being combined into a large system.

We will describe all of these ideas in more detail in subsequent chapters.

Chapter Summary

People deal with complex artifacts and situations every day. Thus, while many readers may not yet have created complex computer programs, they nevertheless will have experience in using the tools that computer scientists employ in managing complexity.

- The most basic tool is *abstraction*, the purposeful suppression of detail in order to emphasize a few basic features.
- *Information hiding* describes the part of abstraction in which we intentionally choose to ignore some features so that we can concentrate on others.

- Abstraction is often combined with a division into *components*. For example, we divided the automobile into the engine and the transmission. Components are carefully chosen so that they *encapsulate* certain key features, and interact with other components through a simple and fixed *interface*.

- The division into components means we can divide a large task into smaller problems that can then be worked on more-or-less independently of each other. It is the responsibility of a developer of a component to provide an *implementation* that satisfies the requirements of the interface.

- A point of view that turns out to be very useful in developing complex software system is the concept of a *service provider*. A software component is providing a service to other components with which it interacts. In real life we often characterize members of the communities in which we operate by the services they provide. (A delivery person is charged with transporting flowers from a florist to a recipient). Thus this metaphor allows one to think about a large software system in the same way that we think about situations in our everyday lives.

- Another form of abstraction is a taxonomy, in object-oriented languages more often termed an *inheritance hierarchy*. Here the layers are more detailed representatives of a general category. An example of this type of system is a biological division into categories such as Living Thing-Animal-Mammal-Cat. Each level is a more specialized version of the previous. This division simplifies understanding, since knowledge of more general levels is applicable to many more specific categories. When applied to software this technique also simplifies the creation of new components, since if a new component can be related to an existing category all the functionality of the older category can be used for free. (Thus, for example, by saying that a new component represents a Frame in the Java library we immediately get features such as a menu bar, as well as the ability to move and resize the window).

- Finally, a particular tool that has become popular in recent years is the *pattern*. A pattern is simply a generalized description of a solution to a problem that has been observed to occur in many places and in many forms. The pattern described how the problem can be addressed, and the reasons both for adopting the solution and for considering other alternatives. We will see several different types of patterns throughout this book.

Further Information

In the sidebar on page 33 we mention software catalogs. For the Java programmer a very useful catalog is *The Java Developers Almanac*, by Patrick Chan [Chan 2000].

The concept of *patterns* actually grew out of work in architecture, specifically the work of Christopher Alexander [Alexander 77]. The application of patterns to software is described by Gabriel [Gabriel 96]. The best-known catalog of software Patterns is by Gamma et al [Gamma 1995]. A more recent almanac that collects several hundred design patterns is [Rising 2000].

The criticism of procedures as an abstraction technique, because they fail to provide an adequate mechanism for information hiding, was first stated by William Wulf and Mary Shaw [Wulf 1973] in an analysis of many of the problems surrounding the use of global variables. These arguments were later expanded upon by David Hanson [Hanson 1981].

David Parnas originally described his principles in [Parnas 1972].

An interesting book that deals with the relationship between how people think and the way they form abstractions of the real word is Lakoff [Lakoff 87].

Self Study Questions

1. What is abstraction?
2. Give an example of how abstraction is used in real life.
3. What is information hiding?
4. Give an example of how information hiding is used in real life.
5. What are the layers of abstraction found in an object-oriented program?
6. What do the terms client and server mean when applied to simple object-oriented programs?
7. What is the distinction between an interface and an implementation?
8. How does an emphasis on encapsulation and the identification of interfaces facilitate interchangeability?
9. What are the basic features of composition as a technique for creating complex systems out of simple parts?
10. How does a division based on layers of specialization differ from a division based on separation into parts?
11. What goal motivates the collection of software patterns?
12. What key idea was first realized by the development of procedures as a programming abstraction?
13. What are the basic features of a module?
14. How is an abstract data type different from a module?
15. In what ways is an object similar to an abstract data type? In what ways are they different?

Exercises

1. Consider a relationship in real life, such as the interaction between a customer and a waiter in a resturant. Describe the interaction govering this relationship in terms of an interface for a customer object and a waiter object.

2. Take a relatively complex structure from real life, such as a building. Describe features of the building using the technique of division into parts, followed by a further refinement of each part into a more detailed description. Extend your description to at least three levels of detail.

3. Describe a collection of everyday objects using the technique of layers of specialization.

Chapter 3

Object-Oriented Design

A cursory explanation of object-oriented programming tends to emphasize the syntactic features of languages such as C++ or Delphi, as opposed to their older, non object-oriented versions, C or Pascal. Thus, an explanation usually turns rather quickly to issues such as classes and inheritance, message passing, and virtual and static methods. But such a description will miss the most important point of object-oriented programming, which has nothing to do with syntax.

Working in an object-oriented language (that is, one that supports inheritance, message passing, and classes) is neither a necessary nor sufficient condition for doing object-oriented programming. As we emphasized in Chapters 1 and 2, the most important aspect of OOP is the creation of a universe of largely autonomous interacting agents. But how does one come up with such a system? The answer is a design technique driven by the determination and delegation of responsibilities. The technique described in this chapter is termed *responsibility-driven design.*[1]

3.1 Responsibility Implies Noninterference

As anyone can attest who can remember being a child, or who has raised children, responsibility is a sword that cuts both ways. When you make an object (be it a child or a software system) responsible for specific actions, you expect a certain behavior, at least when the rules are observed. But just as important, responsibility implies a degree of independence or noninterference. If you tell a child that she is responsible for cleaning her room, you do not normally stand

[1] The past few years have seen a polifieration of object-oriented design techniques. See the section on further reading at the end of this chapter for pointers to some of the alternatives. I have selected Responsibility-driven design, developed by Rebecca Wirfs-brock [Wirfs-Brock 1989b, Wirfs-Brock 1990] because it is one of the simplest, and it facilitates the transition from design to programming. Also in this chapter I introduce some of the notational techniques made popular by the Unified Modelling Language, or UML. However, space does not permit a complete introduction to UML, nor is it necessary for an understanding of subsequent material in the book.

over her and watch while that task is being performed–that is not the nature of responsibility. Instead, you expect that, having issued a directive in the correct fashion, the desired outcome will be produced.

Similarly, in the flowers example from Chapter 1, when Chris gave the request to the Florist to deliver flowers to Robin, it was not necessary to stop to think about how the request would be serviced. The florist, having taken on the responsibility for this service, is free to operate without interference on the part of the customer Chris.

The difference between conventional programming and object-oriented programming is in many ways the difference between actively supervising a child while she performs a task, and delegating to the child responsibility for that performance. Conventional programming proceeds largely by doing something *to* something else–modifying a record or updating an array, for example. Thus, one portion of code in a software system is often intimately tied, by control and data connections, to many other sections of the system. Such dependencies can come about through the use of global variables, through use of pointer values, or simply through inappropriate use of and dependence on implementation details of other portions of code. A responsibility-driven design attempts to cut these links, or at least make them as unobtrusive as possible.

This notion might at first seem no more subtle than the concepts of information hiding and modularity, which are important to programming even in conventional languages. But responsibility-driven design elevates information hiding from a technique to an art. This principle of information hiding becomes vitally important when one moves from programming in the small to programming in the large.

One of the major benefits of object-oriented programming occurs when software subsystems are reused from one project to the next. For example, a simulation manager (such as the one we will develop in Chapter 7) might work for both a simulation of balls on a billiards table and a simulation of fish in a fish tank. This ability to reuse code implies that the software can have almost no domain-specific components; it must totally delegate responsibility for domain-specific behavior to application-specific portions of the system. The ability to create such reusable code is not one that is easily learned–it requires experience, careful examination of case studies (paradigms, in the original sense of the word), and use of a programming language in which such delegation is natural and easy to express. In subsequent chapters, we will present several such examples.

3.2 Programming in the Small and in the Large

The difference between the development of individual projects and of more sizable software systems is often described as programming in the small versus programming in the large.

Programming in the small characterizes projects with the following attributes:

- Code is developed by a single programmer, or perhaps by a very small

collection of programmers. A single individual can understand all aspects of a project, from top to bottom, beginning to end.

- The major problem in the software development process is the design and development of algorithms for dealing with the problem at hand.

Programming in the large, on the other hand, characterizes software projects with features such as the following:

- The software system is developed by a large team, often consisting of people with many different skills. There may be graphic artists, design experts, as well as programmers. Individuals involved in the specification or design of the system may differ from those involved in the coding of individual components, who may differ as well from those involved in the integration of various components in the final product. No single individual can be considered responsible for the entire project, or even necessarily understands all aspects of the project.

- The major problem in the software development process is the management of details and the communication of information between diverse portions of the project.

While the beginning student will usually be acquainted with programming in the small, aspects of many object-oriented languages are best understood as responses to the problems encountered while programming in the large. Thus, some appreciation of the difficulties involved in developing large systems is a helpful prerequisite to understanding OOP.

3.3 Why Begin with Behavior?

Why begin the design process with an analysis of behavior? The simple answer is that the behavior of a system is usually understood long before any other aspect.

Earlier software development methodologies (those popular before the advent of object-oriented techniques) concentrated on ideas such as characterizing the basic data structures or the overall structure of function calls, often within the creation of a formal specification of the desired application. But structural elements of the application can be identified only after a considerable amount of problem analysis. Similarly, a formal specification often ended up as a document understood by neither programmer nor client. But *behavior* is something that can be described almost from the moment an idea is conceived, and (often unlike a formal specification) can be described in terms meaningful to both the programmers and the client.

Responsibility-Driven Design (RDD), developed by Rebecca Wirfs-Brock, is an object-oriented design technique that is driven by an emphasis on behavior at all levels of development. It is but one of many alternative object-oriented

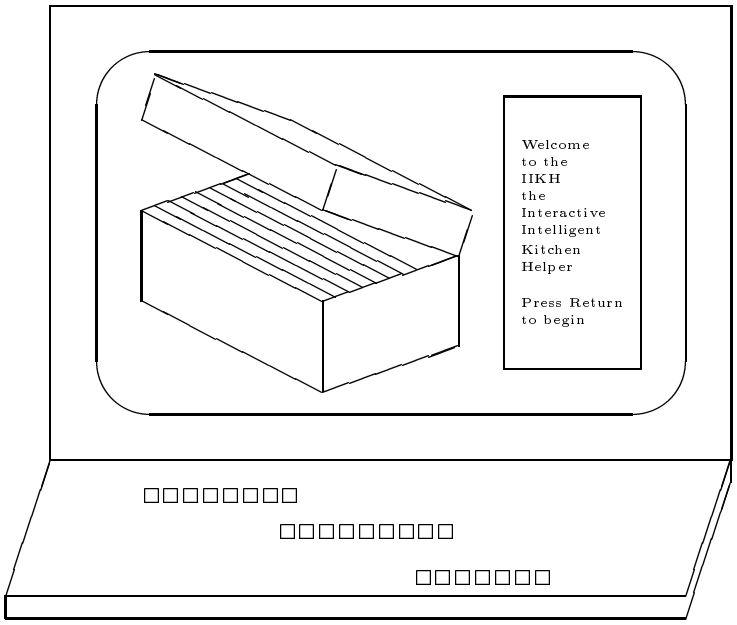


Figure 3.1: – View of the Interactive Intelligent Kitchen Helper.

design techniques. We will illustrate the application of Responsibility-Driven Design with a case study.

3.4 A Case Study in RDD

Imagine you are the chief software architect in a major computer firm. One day your boss walks into your office with an idea that, it is hoped, will be the next major success in your product line. Your assignment is to develop the *Interactive Intelligent Kitchen Helper* (Figure 3.1).

The task given to your software team is stated in very few words (written on what appears to be the back of a slightly-used dinner napkin, in handwriting that appears to be your boss's).

3.4.1 The Interactive Intelligent Kitchen Helper

Briefly, the Interactive Intelligent Kitchen Helper (IIKH) is a PC-based application that will replace the index-card system of recipes found in the average kitchen. But more than simply maintaining a database of recipes, the kitchen helper assists in the planning of meals for an extended period, say a week. The

user of the IIKH can sit down at a terminal, browse the database of recipes, and interactively create a series of menus. The IIKH will automatically scale the recipes to any number of servings and will print out menus for the entire week, for a particular day, or for a particular meal. And it will print an integrated grocery list of all the items needed for the recipes for the entire period.

As is usually true with the initial descriptions of most software systems, the specification for the IIKH is highly ambiguous on a number of important points. It is also true that, in all likelihood, the eventual design and development of the software system to support the IIKH will require the efforts of several programmers working together. Thus, the initial goal of the design team must be to clarify the ambiguities in the description and to outline how the project can be divided into components to be assigned for development to individual team members.

The fundamental cornerstone of object-oriented programming is to characterize software in terms of *behavior*; that is, actions to be performed. We will see this repeated on many levels in the development of the IIKH. Initially, the team will try to characterize, at a very high level of abstraction, the behavior of the entire application. This then leads to a description of the behavior of various software subsystems. Only when all behavior has been identified and described will the software design team proceed to the coding step. In the next several sections we will trace the tasks the software design team will perform in producing this application.

3.4.2 Working through Scenarios

The first task is to refine the specification. As we have already noted, initial specifications are almost always ambiguous and unclear on anything except the most general points. There are several goals for this step. One objective is to get a better handle on the "look and feel" of the eventual product. This information can then be carried back to the client (in this case, your boss) to see if it is in agreement with the original conception. It is likely, perhaps inevitable, that the specifications for the final application will change during the creation of the software system, and it is important that the design be developed to easily accommodate change and that potential changes be noted as early as possible. Equally important, at this point very high level decisions can be made concerning the structure of the eventual software system. In particular, the activities to be performed can be mapped onto components.

In order to uncover the fundamental behavior of the system, the design team first creates a number of *scenarios*. That is, the team acts out the running of the application just as if it already possessed a working system. An example scenario is shown in Figure 3.2.

Simple Browsing

Alice Smith sits down at her computer and starts the IIKH. When the program begins, it displays a graphical image of a recipe box, and identifies itself as the IIKH, product of IIKH incorporated. Alice presses the return button to begin.

In response to the key press, Alice is given a choice of a number of options. She elects to browse the recipe index, looking for a recipe for Salmon that she wishes to prepare for dinner the next day. She enters the keyword Salmon, and is shown in response a list of various recipes. She remembers seeing an interesting recipe that used dill-weed as a flavoring. She refines the search, entering the words Salmon and dill-weed. This narrows the search to two recipes.

She selects the first. This brings up a new window in which an attractive picture of the finished dish is displayed, along with the list of ingredients, preparation steps, and expected preparation time. After examining the recipe, Alice decides it is not the recipe she had in mind. She returns to the search result page, and selects the second alternative.

Examining this dish, Alice decides this is the one she had in mind. She requests a printing of the recipe, and the output is spooled to her printer. Alice selects "quit" from a program menu, and the application quits.

Figure 3.2: An Example Scenario

3.4.3 Identification of Components

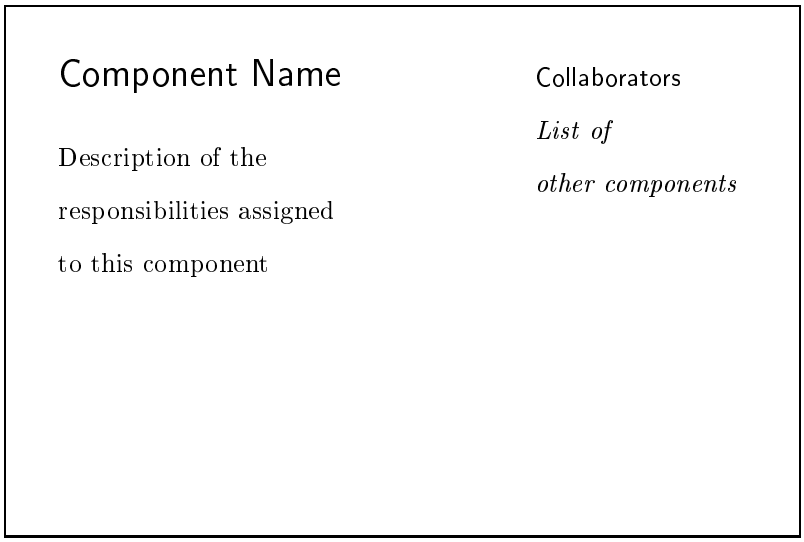
The engineering of a complex physical system, such as a building or an automobile engine, is simplified by dividing the design into smaller units. So, too, the engineering of software is simplified by the identification and development of software components. A *component* is simply an abstract entity that can perform tasks–that is, fulfill some responsibilities. At this point, it is not necessary to know exactly the eventual representation for a component or how a component will perform a task. A component may ultimately be turned into a function, a structure or class, or a collection of other components. At this level of development there are just two important characteristics:

- A component must have a small well-defined set of responsibilities.
- A component should interact with other components to the minimal extent possible.

We will shortly discuss the reasoning behind the second characteristic. For the moment we are simply concerned with the identification of component responsibilities.

3.5 CRC Cards–Recording Responsibility

As the design team walks through the various scenarios they have created, they identify the components that will be performing certain tasks. Every activity that must take place is identified and assigned to some component as a responsibility.



As part of this process, it is often useful to represent components using small index cards. Written on the face of the card is the name of the software component, the responsibilities of the component, and the names of other components with which the component must interact. Such cards are sometimes known as CRC (Component, Responsibility, Collaborator) cards, and are associated with each software component. As responsibilities for the component are discovered, they are recorded on the face of the CRC card.

3.5.1 Give Components a Physical Representation

While working through scenarios, it is useful to assign CRC cards to different members of the design team. The member holding the card representing a component records the responsibilities of the associated software component, and acts as the "surrogate" for the software during the scenario simulation. He or she describes the activities of the software system, passing "control" to another member when the software system requires the services of another component.

An advantage of CRC cards is that they are widely available, inexpensive, and erasable. This encourages experimentation, since alternative designs can be tried, explored, or abandoned with little investment. The physical separation of the cards encourages an intuitive understanding of the importance of the logical separation of the various components, helping to emphasize the cohesion and coupling (which we will describe shortly). The constraints of an index card are also a good measure of approximate complexity–a component that is expected to perform more tasks than can fit easily in this space is probably too complex, and the team should find a simpler solution, perhaps by moving some responsibilities elsewhere to divide a task between two or more new components.

3.5.2 The What/Who Cycle

As we noted at the beginning of this discussion, the identification of components takes place during the process of imagining the execution of a working system. Often this proceeds as a cycle of what/who questions. First, the design team identifies *what* activity needs to be performed next. This is immediately followed by answering the question of *who* performs the action. In this manner, designing a software system is much like organizing a collection of people, such as a club. Any activity that is to be performed must be assigned as a responsibility to some component.

A popular bumper sticker states that phenomena can and will spontaneously occur. (The bumper sticker uses a slightly shorter phrase.) We know, however, that in real life this is seldom true. If any action is to take place, there must be an agent assigned to perform it. Just as in the running of a club any action to be performed must be assigned to some individual, in organizing an object-oriented program all actions must be the responsibility of some component. The secret to good object-oriented design is to first establish an agent for each action.

3.5.3 Documentation

At this point the development of documentation should begin. Two documents should be essential parts of any software system: the user manual and the system design documentation. Work on both of these can commence even before the first line of code has been written.

The user manual describes the interaction with the system from the user's point of view; it is an excellent means of verifying that the development team's conception of the application matches the client's. Since the decisions made in creating the scenarios will closely match the decisions the user will be required to make in the eventual application, the development of the user manual naturally dovetails with the process of walking through scenarios.

Before any actual code has been written, the mindset of the software team is most similar to that of the eventual users. Thus, it is at this point that the developers can most easily anticipate the sort of questions to which a novice user will need answers. A user manual is also an excellent tool to verify that the programming team is looking at the problem in the same way that the client intended. A client seldom presents the programming team with a complete and formal specification, and thus some reassurance and two-way communication early in the process, before actual programming has begun, can prevent major misunderstandings.

The second essential document is the design documentation. The design documentation records the major decisions made during software design, and should thus be produced when these decisions are fresh in the minds of the creators, and not after the fact when many of the relevant details will have been forgotten. It is often far easier to write a general global description of the software system early in the development. Too soon, the focus will move to the level of individual components or modules. While it is also important to document the module level, too much concern with the details of each module will make it difficult for subsequent software maintainers to form an initial picture of the larger structure.

CRC cards are one aspect of the design documentation, but many other important decisions are not reflected in them. Arguments for and against any major design alternatives should be recorded, as well as factors that influenced the final decisions. A log or diary of the project schedule should be maintained. Both the user manual and the design documents are refined and evolve over time in exactly the same way the software is refined and evolves.

3.6 Components and Behavior

To return to the IIKH, the team decides that when the system begins, the user will be presented with an attractive informative window (see Figure 3.1). The responsibility for displaying this window is assigned to a component called the Greeter. In some as yet unspecified manner (perhaps by pull-down menus, button

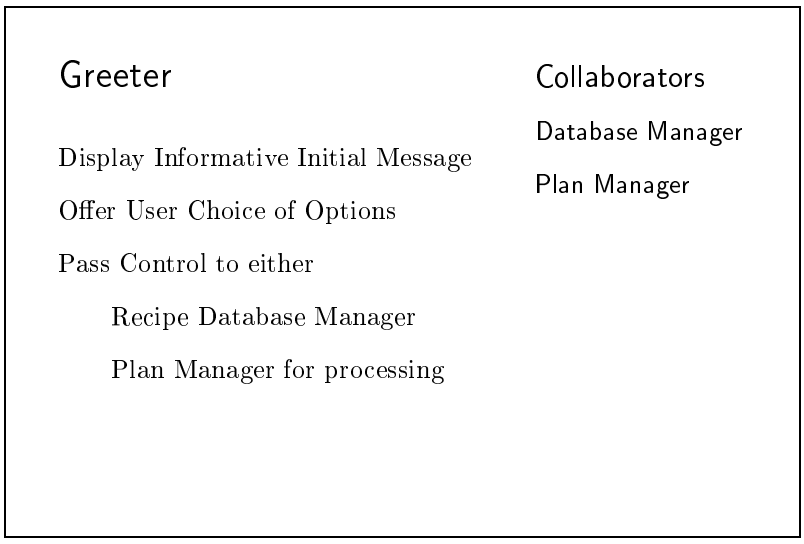


Figure 3.3: – CRC card for the Greeter.

or key presses, or use of a pressure-sensitive screen), the user can select one of several actions. Initially, the team identifies just five actions:

1. Casually browse the database of existing recipes, but without reference to any particular meal plan.
2. Add a new recipe to the database.
3. Edit or annotate an existing recipe.
4. Review an existing plan for several meals.
5. Create a new plan of meals.

These activities seem to divide themselves naturally into two groups. The first three are associated with the recipe database; the latter two are associated with menu plans. As a result, the team next decides to create components corresponding to these two responsibilities. Continuing with the scenario, the team elects to ignore the meal plan management for the moment and move on to refine the activities of the Recipe Database component. Figure 3.3 shows the initial CRC card representation of the Greeter.

Broadly speaking, the responsibility of the recipe database component is simply to maintain a collection of recipes. We have already identified three elements of this task: The recipe component database must facilitate browsing the library of existing recipes, editing the recipes, and including new recipes in the database.

3.6.1 Postponing Decisions

There are a number of decisions that must eventually be made concerning how best to let the user browse the database. For example, should the user first be presented with a list of categories, such as "soups," "salads," "main meals," and "desserts"? Alternatively, should the user be able to describe keywords to narrow a search, perhaps by providing a list of ingredients, and then see all the recipes that contain those items ("Almonds, Strawberries, Cheese"), or a list of previously inserted keywords ("Bob's favorite cake")? Should scroll bars be used or simulated thumb holes in a virtual book? These are fun to think about, but the important point is that such decisions do not need to be made at this point (see Section 3.6.2, "Preparing for Change"). Since they affect only a single component, and do not affect the functioning of any other system, all that is necessary to continue the scenario is to assert that by some means the user can select a specific recipe.

3.6.2 Preparing for Change

It has been said that all that is constant in life is the inevitability of uncertainty and change. The same is true of software. No matter how carefully one tries to develop the initial specification and design of a software system, it is almost certain that changes in the user's needs or requirements will, sometime during the life of the system, force changes to be made in the software. Programmers and software designers need to anticipate this and plan accordingly.

- The primary objective is that changes should affect as few components as possible. Even major changes in the appearance or functioning of an application should be possible with alterations to only one or two sections of code.

- Try to predict the most likely sources of change and isolate the effects of such changes to as few software components as possible. The most likely sources of change are interfaces, communication formats, and output formats.

- Try to isolate and reduce the dependency of software on hardware. For example, the interface for recipe browsing in our application may depend in part on the hardware on which the system is running. Future releases may be ported to different platforms. A good design will anticipate this change.

- Reducing coupling between software components will reduce the dependence of one upon another, and increase the likelihood that one can be changed with minimal effect on the other.

- In the design documentation maintain careful records of the design process and the discussions surrounding all major decisions. It is almost certain

that the individuals responsible for maintaining the software and designing future releases will be at least partially different from the team producing the initial release. The design documentation will allow future teams to know the important factors behind a decision and help them avoid spending time discussing issues that have already been resolved.

3.6.3 Continuing the Scenario

Each recipe will be identified with a specific recipe component. Once a recipe is selected, control is passed to the associated recipe object. A recipe must contain certain information. Basically, it consists of a list of ingredients and the steps needed to transform the ingredients into the final product. In our scenario, the recipe component must also perform other activities. For example, it will display the recipe interactively on the terminal screen. The user may be given the ability to annotate or change either the list of ingredients or the instruction portion. Alternatively, the user may request a printed copy of the recipe. All of these actions are the responsibility of the Recipe component. (For the moment, we will continue to describe the Recipe in singular form. During design we can think of this as a prototypical recipe that stands in place of a multitude of actual recipes. We will later return to a discussion of singular versus multiple components.)

Having outlined the actions that must take place to permit the user to browse the database, we return to the recipe database manager and pretend the user has indicated a desire to add a new recipe. The database manager somehow decides in which category to place the new recipe (again, the details of how this is done are unimportant for our development at this point), requests the name of the new recipe, and then creates a new recipe component, permitting the user to edit this new blank entry. Thus, the responsibilities of performing this new task are a subset of those we already identified in permitting users to edit existing recipes.

Having explored the browsing and creation of new recipes, we return to the Greeter and investigate the development of daily menu plans, which is the Plan Manager's task. In some way (again, the details are unimportant here) the user can save existing plans. Thus, the Plan Manager can either be started by retrieving an already developed plan or by creating a new plan. In the latter case, the user is prompted for a list of dates for the plan. Each date is associated with a separate Date component. The user can select a specific date for further investigation, in which case control is passed to the corresponding Date component. Another activity of the Plan Manager is printing out the recipes for the planning period. Finally, the user can instruct the Plan Manager to produce a grocery list for the period.

The Date component maintains a collection of meals as well as any other annotations provided by the user (birthday celebrations, anniversaries, reminders, and so on). It prints information on the display concerning the specified date. By some means (again unspecified), the user can indicate a desire to print all the information concerning a specific date or choose to explore in more detail a

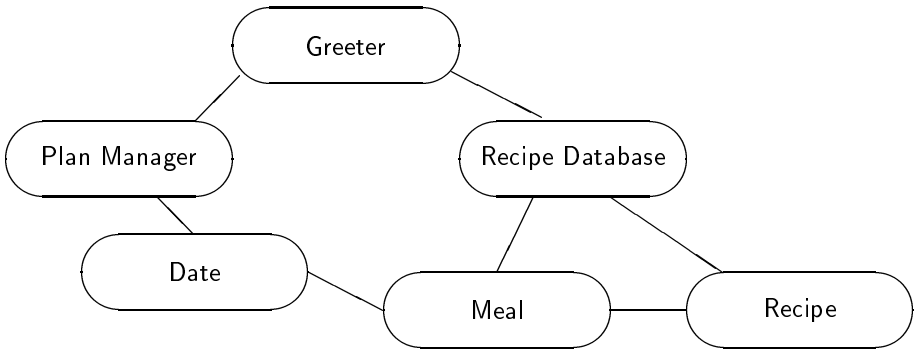


Figure 3.4: – Communication between the six components in the IIKH.

specific meal. In the latter case, control is passed to a Meal component.

The Meal component maintains a collection of augmented recipes, where the augmentation refers to the user's desire to double, triple, or otherwise increase a recipe. The Meal component displays information about the meal. The user can add or remove recipes from the meal, or can instruct that information about the meal be printed. In order to discover new recipes, the user must be permitted at this point to browse the recipe database. Thus, the Meal component must interact with the recipe database component. The design team will continue in this fashion, investigating every possible scenario. The major category of scenarios we have not developed here is exceptional cases. For example, what happens if a user selects a number of keywords for a recipe and no matching recipe is found? How can the user cancel an activity, such as entering a new recipe, if he or she decides not to continue? Each possibility must be explored, and the responsibilities for handling the situation assigned to one or more components.

Having walked through the various scenarios, the software design team eventually decides that all activities can be adequately handled by six components (Figure 3.4). The Greeter needs to communicate only with the Plan Manager and the Recipe Database components. The Plan Manager needs to communicate only with the Date component; and the Date agent, only with the Meal component. The Meal component communicates with the Recipe Manager and, through this agent, with individual recipes.

3.6.4 Interaction Diagrams

While a description such as that shown in Figure 3.4 may describe the static relationships between components, it is not very good for describing their dynamic interactions during the execution of a scenario. A better tool for this purpose is an *interaction diagram*. Figure 3.5 shows the beginning of an interaction diagram for the interactive kitchen helper. In the diagram, time moves forward from the

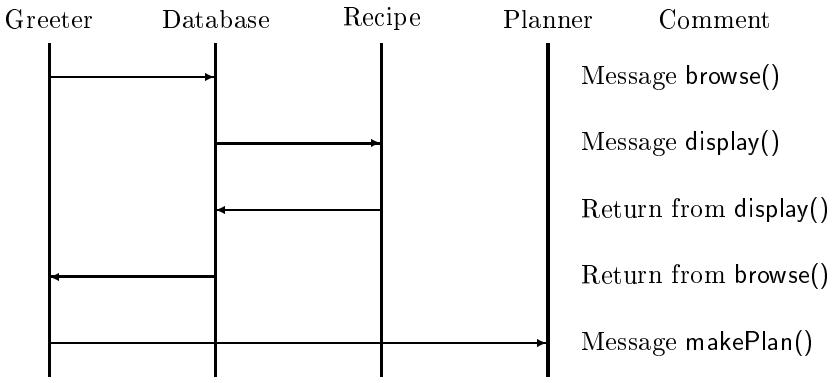


Figure 3.5: – An Example interaction diagram.

top to the bottom. Each component is represented by a labeled vertical line. A component sending a message to another component is represented by a horizontal arrow from one line to another. Similarly, a component returning control and perhaps a result value back to the caller is represented by an arrow. (Some authors use two different arrow forms, such as a solid line to represent message passing and a dashed line to represent returning control.) The commentary on the right side of the figure explains more fully the interaction taking place.

With a time axis, the interaction diagram is able to describe better the sequencing of events during a scenario. For this reason, interaction diagrams can be a useful documentation tool for complex software systems.

3.7 Software Components

In this section we will explore a software component in more detail. As is true of all but the most trivial ideas, there are many aspects to this seemingly simple concept.

3.7.1 Behavior and State

We have already seen how components are characterized by their behavior, that is, by what they can do. But components may also hold certain information. Let us take as our prototypical component a Recipe structure from the IIKH. One way to view such a component is as a pair consisting of *behavior* and *state*.

- The *behavior* of a component is the set of actions it can perform. The complete description of all the behavior for a component is sometimes

called the *protocol*. For the Recipe component this includes activities such as editing the preparation instructions, displaying the recipe on a terminal screen, or printing a copy of the recipe.

- The *state* of a component represents all the information held within it at a given point of time. For our Recipe component the state includes the ingredients and preparation instructions. Notice that the state is not static and can change over time. For example, by editing a recipe (a behavior) the user can make changes to the preparation instructions (part of the state).

It is not necessary that all components maintain state information. For example, it is possible that the Greeter component will not have any state since it does not need to remember any information during the course of execution. However, most components will consist of a combination of behavior and state.

3.7.2 Instances and Classes

The separation of state and behavior permits us to clarify a point we avoided in our earlier discussion. Note that in the real application there will probably be many different recipes. However, all of these recipes will *perform* in the same manner. That is, the behavior of each recipe is the same; it is only the state–the individual lists of ingredients and instructions for preparation–that differs between individual recipes. In the early stages of development our interest is in characterizing the behavior common to all recipes; the details particular to any one recipe are unimportant.

The term *class* is used to describe a set of objects with similar behavior. We will see in later chapters that a class is also used as a syntactic mechanism in almost all object-oriented languages. An individual representative of a class is known as an *instance*. Note that behavior is associated with a class, not with an individual. That is, all instances of a class will respond to the same instructions and perform in a similar manner. On the other hand, state is a property of an individual. We see this in the various instances of the class Recipe. They can all perform the same actions (editing, displaying, printing) but use different data values.

3.7.3 Coupling and Cohesion

Two important concepts in the design of software components are coupling and cohesion. Cohesion is the degree to which the responsibilities of a single component form a meaningful unit. High cohesion is achieved by associating in a single component tasks that are related in some manner. Probably the most frequent way in which tasks are related is through the necessity to access a common data value. This is the overriding theme that joins, for example, the various responsibilities of the Recipe component.

Coupling, on the other hand, describes the relationship between software components. In general, it is desirable to reduce the amount of coupling as much as possible, since connections between software components inhibit ease of development, modification, or reuse.

In particular, coupling is increased when one software component must access data values–the state–held by another component. Such situations should almost always be avoided in favor of moving a task into the list of responsibilities of the component that holds the necessary data. For example, one might conceivably first assign responsibility for editing a recipe to the Recipe Database component, since it is while performing tasks associated with this component that the need to edit a recipe first occurs. But if we did so, the Recipe Database agent would need the ability to directly manipulate the state (the internal data values representing the list of ingredients and the preparation instructions) of an individual recipe. It is better to avoid this tight connection by moving the responsibility for editing to the recipe itself.

3.7.4 Interface and Implementation–Parnas's Principles

The emphasis on characterizing a software component by its behavior has one extremely important consequence. It is possible for one programmer to know how to *use* a component developed by another programmer, without needing to know how the component is *implemented*. For example, suppose each of the six components in the IIKH is assigned to a different programmer. The programmer developing the Meal component needs to allow the IIKH user to browse the database of recipes and select a single recipe for inclusion in the meal. To do this, the Meal component can simply invoke the browse behavior associated with the Recipe Database component, which is defined to return an individual Recipe. This description is valid regardless of the particular implementation used by the Recipe Database component to perform the actual browsing action.

The purposeful omission of implementation details behind a simple interface is known as *information hiding*. We say the component *encapsulates* the behavior, showing only how the component can be used, not the detailed actions it performs. This naturally leads to two different views of a software system. The interface view is the face seen by other programmers. It describes *what* a software component can perform. The implementation view is the face seen by the programmer working on a particular component. It describes *how* a component goes about completing a task.

The separation of interface and implementation is perhaps *the* most important concept in software engineering. Yet it is difficult for students to understand, or to motivate. Information hiding is largely meaningful only in the context of multiperson programming projects. In such efforts, the limiting factor is often not the amount of coding involved, but the amount of communication required between the various programmers and between their respective software systems. As we will describe shortly, software components are often developed in parallel by different programmers, and in isolation from each other.

There is also an increasing emphasis on the reuse of general-purpose software components in multiple projects. For this to be successful, there must be minimal and well-understood interconnections between the various portions of the system. As we noted in the previous chapter, these ideas were captured by computer scientist David Parnas in a pair of rules, known as *Parnas's principles:*

- The developer of a software component must provide the intended user with all the information needed to make effective use of the services provided by the component, and should provide *no* other information.
- The developer of a software component must be provided with all the information necessary to carry out the given responsibilities assigned to the component, and should be provided with *no* other information.

A consequence of the separation of interface from implementation is that a programmer can experiment with several different implementations of the same structure without affecting other software components.

3.8 Formalize the Interface

We continue with the description of the IIKH development. In the next several steps the descriptions of the components will be refined. The first step in this process is to formalize the patterns and channels of communication.

A decision should be made as to the general structure that will be used to implement each component. A component with only one behavior and no internal state may be made into a function–for example, a component that simply takes a string of text and translates all capital letters to lowercase. Components with many tasks are probably more easily implemented as classes. Names are given to each of the responsibilities identified on the CRC card for each component, and these will eventually be mapped onto method names. Along with the names, the types of any arguments to be passed to the function are identified. Next, the information maintained within the component itself should be described. All information must be accounted for. If a component requires some data to perform a specific task, the source of the data, either through argument or global value, or maintained internally by the component, must be clearly identified.

3.8.1 Coming up with Names

Careful thought should be given to the names associated with various activities. Shakespeare said that a name change does not alter the object being described, but certainly not all names will conjure up the same mental images in the listener. As government bureaucrats have long known, obscure and idiomatic names can make even the simplest operation sound intimidating. The selection of useful names is extremely important, as names create the vocabulary with which the eventual design will be formulated. Names should be internally consistent, meaningful, preferably short, and evocative in the context of the problem. Often a

considerable amount of time is spent finding just the right set of terms to describe the tasks performed and the objects manipulated. Far from being a barren and useless exercise, proper naming early in the design process greatly simplifies and facilitates later steps.

The following general guidelines have been suggested:

- Use pronounceable names. As a rule of thumb, if you cannot read a name out loud, it is not a good one.
- Use capitalization (or underscores) to mark the beginning of a new word within a name, such as "CardReader" or "Card_reader," rather than the less readable "cardreader."
- Examine abbreviations carefully. An abbreviation that is clear to one person may be confusing to the next. Is a "TermProcess" a terminal process, something that terminates processes, or a process associated with a terminal?
- Avoid names with several interpretations. Does the empty function tell whether something is empty, or empty the values from the object?
- Avoid digits within a name. They are easy to misread as letters (0 as O, 1 as l, 2 as Z, 5 as S).
- Name functions and variables that yield Boolean values so they describe clearly the interpretation of a true or false value. For example, "PrinterIsReady" clearly indicates that a true value means the printer is working, whereas "PrinterStatus" is much less precise.
- Take extra care in the selection of names for operations that are costly and infrequently used. By doing so, errors caused by using the wrong function can be avoided.

Once names have been developed for each activity, the CRC cards for each component are redrawn, with the name and formal arguments of the function used to elicit each behavior identified. An example of a CRC card for the Date is shown in Figure 3.6. What is not yet specified is how each component will perform the associated tasks.

Once more, scenarios or role playing should be carried out at a more detailed level to ensure that all activities are accounted for, and that all necessary information is maintained and made available to the responsible components.

3.9 Designing the Representation

At this point, if not before, the design team can be divided into groups, each responsible for one or more software components. The task now is to transform the description of a component into a software system implementation. The

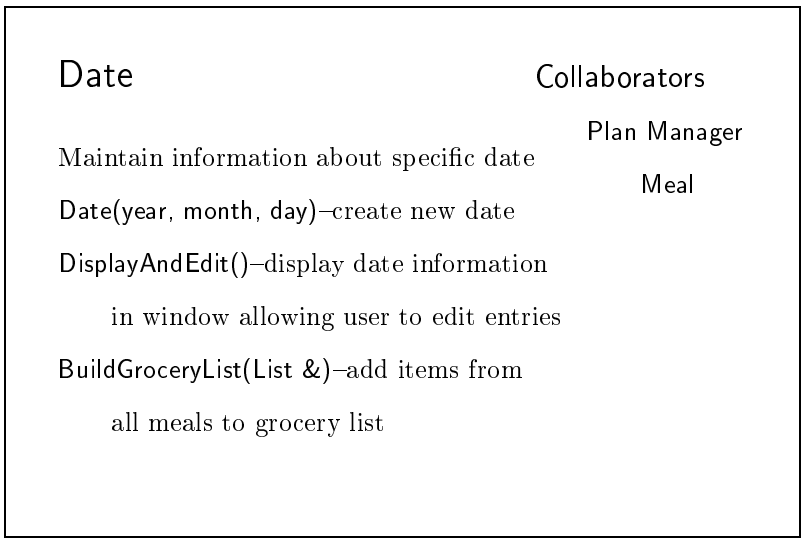


Figure 3.6: – Revised CRC card for the Date component.

major portion of this process is designing the data structures that will be used by each subsystem to maintain the state information required to fulfill the assigned responsibilities.

It is here that the classic data structures of computer science come into play. The selection of data structures is an important task, central to the software design process. Once they have been chosen, the code used by a component in the fulfillment of a responsibility is often almost self-evident. But data structures must be carefully matched to the task at hand. A wrong choice can result in complex and inefficient programs, while an intelligent choice can result in just the opposite.

It is also at this point that descriptions of behavior must be transformed into algorithms. These descriptions should then be matched against the expectations of each component listed as a collaborator, to ensure that expectations are fulfilled and necessary data items are available to carry out each process.

3.10 Implementing Components

Once the design of each software subsystem is laid out, the next step is to implement each component's desired behavior. If the previous steps were correctly addressed, each responsibility or behavior will be characterized by a short description. The task at this step is to implement the desired activities in a computer language. In a later section we will describe some of the more common

heuristics used in this process.

If they were not determined earlier (say, as part of the specification of the system), then decisions can now be made on issues that are entirely self-contained within a single component. A decision we saw in our example problem was how best to let the user browse the database of recipes.

As multiperson programming projects become the norm, it becomes increasingly rare that any one programmer will work on all aspects of a system. More often, the skills a programmer will need to master are understanding how one section of code fits into a larger framework and working well with other members of a team.

Often, in the implementation of one component it will become clear that certain information or actions might be assigned to yet another component that will act "behind the scene," with little or no visibility to users of the software abstraction. Such components are sometimes known as *facilitators*. We will see examples of facilitators in some of the later case studies.

An important part of analysis and coding at this point is characterizing and documenting the necessary preconditions a software component requires to complete a task, and verifying that the software component will perform correctly when presented with legal input values.

3.11 Integration of Components

Once software subsystems have been individually designed and tested, they can be integrated into the final product. This is often not a single step, but part of a larger process. Starting from a simple base, elements are slowly added to the system and tested, using *stubs*–simple dummy routines with no behavior or with very limited behavior–for the as yet unimplemented parts.

For example, in the development of the IIKH, it would be reasonable to start integration with the Greeter component. To test the Greeter in isolation, stubs are written for the Recipe Database manager and the daily Meal Plan manager. These stubs need not do any more than print an informative message and return. With these, the component development team can test various aspects of the Greeter system (for example, that button presses elicit the correct response). Testing of an individual component is often referred to as *unit testing*.

Next, one or the other of the stubs can be replaced by more complete code. For example, the team might decide to replace the stub for the Recipe Database component with the actual system, maintaining the stub for the other portion. Further testing can be performed until it appears that the system is working as desired. (This is sometimes referred to as *integration testing*.)

The application is finally complete when all stubs have been replaced with working components. The ability to test components in isolation is greatly facilitated by the conscious design goal of reducing connections between components, since this reduces the need for extensive stubbing.

During integration it is not uncommon for an error to be manifested in one

software system, and yet to be caused by a coding mistake in another system. Thus, testing during integration can involve the discovery of errors, which then results in changes to some of the components. Following these changes the components should be once again tested in isolation before an attempt to reintegrate the software, once more, into the larger system. Reexecuting previously developed test cases following a change to a software component is sometimes referred to as *regression testing*.

3.12 Maintenance and Evolution

It is tempting to think that once a working version of an application has been delivered the task of the software development team is finished. Unfortunately, that is almost never true. The term *software maintenance* describes activities subsequent to the delivery of the initial working version of a software system. A wide variety of activities fall into this category.

- Errors, or *bugs*, can be discovered in the delivered product. These must be corrected, either in updates or corrections to existing releases or in subsequent releases.
- Requirements may change, perhaps as a result of government regulations or standardization among similar products.
- Hardware may change. For example, the system may be moved to different platforms, or input devices, such as a pen-based system or a pressure-sensitive touch screen, may become available. Output technology may change–for example, from a text-based system to a graphical window-based arrangement.
- User expectations may change. Users may expect greater functionality, lower cost, and easier use. This can occur as a result of competition with similar products.
- Better documentation may be requested by users.

A good design recognizes the inevitability of changes and plans an accommodation for them from the very beginning.

Chapter Summary

In this chapter we have presented a very abbreviated introduction to the basic ideas of object-oriented modeling and design. References in the following section can be consulted for more detailed discussion of this topic.

Object-oriented design differs from conventional software design in that the driving force is the assignment of responsibilities to different software components. No action will take place without an agent to perform the action, and

hence every action must be assigned to some member of the object community. Conversely, the behavior of the members of the community taken together must be sufficient to achieve the desired goal.

The emphasis on *behavior* is a hall-mark of object-oriented programming. Behavior can be identified in even the most rudimentary descriptions of a system, long before any other aspect can be clearly discerned. By constantly being driven by behavior, responsibility driven design moves smoothly from problem description to software architecture to code development to finished application.

Further Reading

Responsibility-driven design was developed and first described by Rebecca Wirfs-Brock [Wirfs-Brock 1989b, Wirfs-Brock 1990]. There are many other object-oriented design techniques, such as that of Jacobson [Jacobson 1994] or Rumbaugh [Rumbaugh 1991], but I like responsibility-driven design because it is among the simplest to explain, and is therefore a good introduction to object-oriented design and modeling.

Much of the most recent work in the field of object-oriented design has centered on UML, the *Unified Modeling Language*. I are not going to discuss UML in detail in this book, although I do use some of their notation in describing class diagrams. A good introduction to UML is [Booch 1999]. A slightly simpler explanation is found in [Alhir 1998].

Other good books on object-oriented design include [Rumbaugh 1991] and [Henderson-Sellers 1992].

CRC cards were developed by Beck [Beck 1989]. A more in-depth book-length treatment of the idea is [Bellin 1997].

Parnas's principles were first presented in [Parnas 1972].

The guidelines on names presented in Section 3.8.1 are from [Keller 1990]. The Shakespeare reference in that same section is to *Romeo and Juliet*, Act II, Scene 2:

> What's in a name?
> That which we call a rose,
> by any other name would smell as sweet;
> So Romeo would, were he not Romeo call'd,
> retain that dear perfection which he owes without that title.

Self Study Questions

1. What are the key features of responsibility-driven design?

2. What are some key differences between programming in-the-small and programming in-the-large?

3. Why can a design technique based on behavior be applied more easily to poorly-defined problems than can, say, a design approach based on data structures?
4. What is a scenario?
5. What are the basic elements of a component?
6. What is a CRC card? What do the letters stand for?
7. What is the what/who cycle?
8. Why should a user manual be developed before coding begins?
9. What are the major sources of change that can be expected during the lifetime of most long-lived software applications?
10. What information is conveyed by an interaction diagram?
11. What are Parnas's principles?
12. Why is the selection of good names an important aspect of a successful software design effort? What are some guidelines for choosing names?
13. What is integration testing?
14. What is software maintenance?

Exercises

1. Describe the responsibilities of an organization that includes at least six types of members. Examples of such organizations are a school (students, teachers, principal, janitor), a business (secretary, president, worker), and a club (president, vice-president, member). For each member type, describe the responsibilities and the collaborators.
2. Create a scenario for the organization you described in Exercise 1 using an interaction diagram.
3. For a common game such as solitaire or twenty-one, describe a software system that will interact with the user as an opposing player. Example components include the deck and the discard pile.
4. Describe the software system to control an ATM (Automated Teller Machine). Give interaction diagrams for various scenarios that describe the most common uses of the machine.

Chapter 4

Classes and Methods

Although the terms they use may be different, all object-oriented languages have in common the concepts introduced in Chapter 1: *classes*, *instances*, *message passing*, *methods*, and *inheritance*. As noted already, the use of different terms for similar concepts is rampant in object-oriented programming languages. We will use a consistent and, we hope, clear terminology for all languages, and we will note in language-specific sections the various synonyms for our terms. Readers can refer to the glossary at the end of the book for explanations of unfamiliar terms.

This chapter will describe the definition or creation of classes, and Chapter 5 will outline their dynamic use. Here we will illustrate the mechanics of declaring a class and defining methods associated with instances of the class. In Chapter 5 we will examine how instances of classes are created and how messages are passed to those instances. For the most part we will defer an explanation of the mechanics of inheritance until Chapter 8.

4.1 Encapsulation

In Chapter 1, we noted that object-oriented programming, and objects in particular, can be viewed from many perspectives. In Chapter 2 we described the many levels of abstraction from which one could examine a program. In this chapter, we wish to view objects as examples of *abstract data types*.

Programming that makes use of data abstractions is a methodological approach to problem solving where information is consciously hidden in a small part of a program. In particular, the programmer develops a series of abstract data types, each of which can be viewed as having two faces. This is similar to the dichotomy in Parnas's principles, discussed in Chapter 3. From the outside, a client (user) of an abstract data type sees only a collection of operations that define the behavior of the abstraction. On the other side of the interface, the programmer defining the abstraction sees the data variables that are used to maintain the internal state of the object.

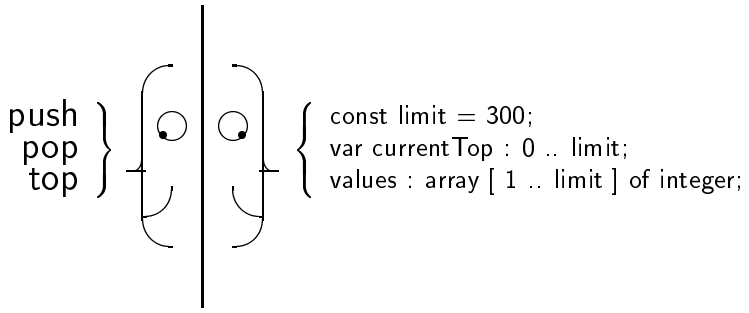


Figure 4.1: – The interface and implementation faces of a stack.

For example, in an abstraction of a stack data type, the user would see only the description of the legal operations–say, push, pop, and top. The implementor, on the other hand, needs to know the actual concrete data structures used to implement the abstraction (Figure 4.1). The concrete details are encapsulated within a more abstract framework.

We have been using the term *instance* to mean a representative, or example, of a class. We will accordingly use the term *instance variable* to mean an internal variable maintained by an instance. Other terms we will occasionally use are data field, or data members. Each instance has its own collection of instance variables. These values should not be changed directly by clients, but rather should be changed only by methods associated with the class.

A simple view of an object is, then, a combination of *state* and *behavior*. The state is described by the instance variables, whereas the behavior is characterized by the methods. From the outside, clients can see only the behavior of objects; from the inside, the methods provide the appropriate behavior through modifications of the state as well as by interacting with other objects.

4.2 Class Definitions

Throughout this chapter and the next we will use as an example the development of a playing card abstraction, such as would be used in a card game application. We will develop this abstraction through a sequence of refinements, each refinement incorporating a small number of new features.

We start by imagining that a playing card can be abstracted as a container for two data values; the card rank and card suit. We can use a number between 1 and 13 to represent the rank (1 is ace, 11, 12 and 13 are Jack, Queen and King). To represent the suit we can use an enumerated data type, if our language provides such facilities. In languages that do not have enumerated data types we can use symbolic constants and integer values from 1 to 4. (The advantage of the enumerated data type is that type errors are avoided, as we can guaranteed the

suit is one of the four specified values. If we use integers for this purpose than nothing prevents a programmer from assigning an invalid integer number, for example 42, to the suit variable.)

4.2.1 C++, Java and C#

We begin by looking at class definitions in three very similar languages; C++, Java and C#. The syntax used by these three languages in shown in Figure 4.2. There are some superficial differences, for example a class definition is terminated by a semicolon in C++, and not in the other two. Visibility modifiers (that is, public) mark an entire block of declarations in C++, and are placed on each declaration independently in the other two languages. In C++ and C# a programmer can define an enumerated data type for representing the playing card suits. By placing the definition inside the class in C++ the programmer makes clear the link between the two data types. (This is not possible in C#). Outside of the class definition the symbolic constants that represent the suits must be prefixed by the class name, as in C++:

```
if (aCard.suit() == PlayingCard::Diamond) ...
```

or by the type name as in C#:

```
if (aCard.suit() == Suits.Diamond) ...
```

Here aCard is the name of an instance of Playing Card, and we are invoking the method named suit in order to test the suit of the card. The data fields suitValue and rankValue represent the instance data for this abstraction. Each instance of the class PlayingCard will have their own separate fields, maintaining their own suit and rank values. Notice that the value of the suit is obtained by invoking a method named suit, which simply returns the data field named suitValue.

The first letter of the class name has here been capitalized. This is a convention that is followed by many langauges, although not universally (in particular, many C++ programmers prefer to use names that are all small letters). Normally instance variables are given names that are not capitalized, so as to make it easier to distinguish between the two categories. Some languages, such as Delphi Pascal, have other conventions.

Enumerated data types are not provided by the language Java, and so the programmer typically resorts to defining a series of symbolic constants. A symbolic constant is characterized by the two modifiers final and static. In Java the modifier final means that the assignment of the name cannot subsequently be changed. The modifier static means that there exists only one instance of a variable, regardless of how many instances of the class are created. Taken together, the two define a unique variable that cannot change; that is, a constant.

C++

```
class PlayingCard {
public:
    enum Suits {Spade, Diamond, Club, Heart};

    Suits suit () { return suitValue; }
    int   rank () { return rankValue; }

private:
    Suits suitValue;
    int   rankValue;
};
```

Java

```
class PlayingCard {
    public  int suit () { return suitValue; }
    public  int rank () { return rankValue; }

    private int suitValue;
    private int rankValue;

    public static final int Spade = 1;
    public static final int Diamond = 2;
    public static final int Club = 3;
    public static final int Heart = 4;
}
```

C#

```
enum Suits {Spade, Diamond, Club, Heart};

class PlayingCard {
    public Suits suit () { return suitValue; }
    public int   rank () { return rankValue; }

    private Suits suitValue;
    private int   rankValue;
}
```

Figure 4.2: A Simple Class Definition in C++, Java and C#

Note the essential difference between the data fields suitValue and rankValue and the constants Heart, Spade, Diamond and Club in the Java definition. Because the latter are declared as static they exist outside of any one instance of the class, and are shared by all instances. The suit and rank fields, on the other hand, are not static and hence each instance of the class will have their own copy of these values.

Visibility Modifiers

Note the use of the terms public and private in several places in these examples. These are *visibility* modifiers. All three languages, as well as a number of other object-oriented programming languages, provide a way of describing features that are known and can be used *outside* the class definition, and distinguishing those from features that can only be used *within* a class definition. The later are indicated by the keyword private.

4.2.2 Apple Object Pascal and Delphi Pascal

The next two languages we will consider are also very similar. Both Apples Object Pascal language and Borlands Delphi Pascal (called Kylix on Linux platforms) were based on an earlier language named simply Pascal. Thus, many features derived from the original language are the same. However, the two vendors have extended the language in slightly different ways.

Both languages permit the creation of enumerated data types, similar to C++. The Apple version of the language uses the keyword object to declare a new class (and hence classes are sometimes termed *object types* in that language). The Delphi language uses the keyword class, and furthermore requires that every class inherit from some existing class. We have here used the class TObject for this purpose. It is conventional in Delphi that all classes must have names that begin with the letter T. Delphi uses visibility modifiers, the Apple language does not. Finally the Delphi language requires the creation of a constructor, a topic we will return to shortly.

4.2.3 Smalltalk

Smalltalk does not actually have a textual representation of a class. Instead, classes are described using an interactive interface called the browser. A screen shot of the browser is shown in Figure 4.4. Using the browser, the programmer can define a new class using a message sent to the parent class Object. As in Delphi Pascal, all classes in Smalltalk must name a specific parent class from which they will inherit. Figure 4.4 illustrates the creation of the class PlayingCard with two instance data fields.

Object Pascal

```
type
    Suits = (Heart, Club, Diamond, Spade);

    PlayingCard = object
        suit : Suits;
        rand : integer;
    end;
```

Delphi Pascal

```
type
    Suits = (Heart, Club, Diamond, Spade);

    TPlayingCard = class (TObject)
        public
            constructor Create (r : integer; s : Suits);

            function suit : Suits;
            function rank : int;

        private
            suitValue : Suits;
            rankValue : integer;
    end;
```

Figure 4.3: Class Definitions in Object Pascal and Delphi Pascal

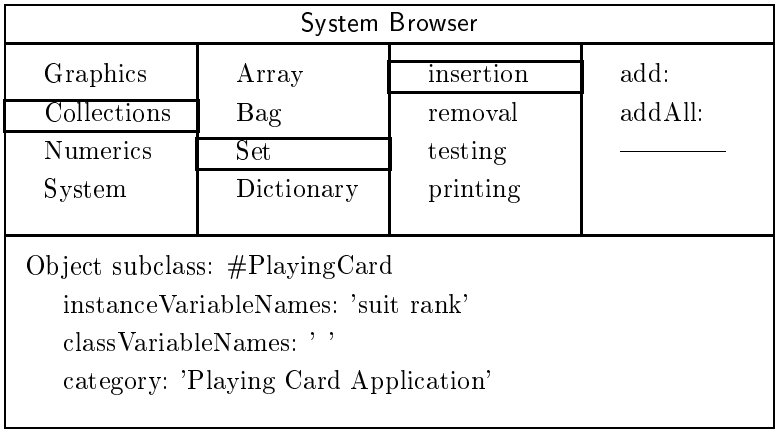


Figure 4.4: – A view of the Smalltalk browser.

CLOS	`(defclass PlayingCard ( ) (rank suit) )`

Eiffel	`class PlayingCard`
	`feature`
	`  Spade, Diamond, Heart, Club : Integer is Unique;`
	
	`  suit : integer;`
	`  rank : integer;`
	`end`

Objective-C	`enum suits {Heart, Club, Diamond, Spade};`
	
	`@ interface PlayingCard : Object`
	`{`
	`  suits suit;`
	`  int rank;`
	`}`
	`@end`

Python	`class PlayingCard:`
	`  "A playing card class"`
	`  def __init__ (self, s, r):`
	`  self.suit = s`
	`  self.rank = r`

Figure 4.5: Class Definitions in Other Object-Oriented Languages

4.2.4 Other Languages

We will periodically throughout the book refer to a number of languages, particularly when they include features that are unique or not widely found in alternative languages. Some of these include Objective-C, CLOS, Eiffel, Dylan and Python. Class definitions for some of these are shown in Figure 4.5. Python is interesting in that indentation levels, rather than beginning and ending tokens, are used to indicate class, function and statement nesting.

4.3 Methods

In the next revision of our playing card abstraction we make the following changes:

- We add a method that will return the face color of the card, either red or black.

```
class PlayingCard {
        // constructor, initialize new playing card
    public PlayingCard (Suits is, int ir)
        { suit = is; rank = ir; faceUp = true; }

        // operations on a playing card
    public boolean isFaceUp  ()           { return faceUp; }
    public int     rank      ()           { return rankValue; }
    public Suits   suit      ()           { return suitValue; }
    public void    setFaceUp (boolean up) { faceUp = up; }
    public void    flip      ()           { setFaceUp( !faceUp);}
    public Color   color     ()           {
        if ((suit() == Suits.Diamond) || (suit() == Suits.Heart))
            return Color.Red;
        return Color.Black;
    }
        // private data values
    private Suits suitValue;
    private int rankValue;
    private boolean faceUp;
}
```

Figure 4.6: The Revised PlayingCard class in C#

- We add a data field to maintain whether the card is face up or face down, and methods both to test the state of this value and to flip the card.

A typical class that illustrates these changes is the C# definition shown in Figure 4.6. Some features to note are that we have added a second enumerated data type to represent the colors, and the data fields (including the third data field representing the face up state of the card) are declared private. By declaring the data fields private it means that access outside the class definition is not permitted. This guarantees that the only way the data fields will be modified is by methods associated with the class. Most object-oriented style guidelines will instruct that data fields should never be declared public, and should always be private or protected, the latter a third level of protection we will discuss after we introduce inheritance in Chapter 8.

The constructor is a special method that has the same name as the class, and is used to initialize the data fields in an object. As we noted earlier, we will discuss constructors in more detail in the next chapter.

Where access to data fields must be provided, good object-oriented style says that access should be mediated by methods defined in the class. A method that does nothing more than return the value of a data field is termed an *accessor*, or sometimes a *getter*. An example is the method isFaceUp, which returns the value of the data field faceUp. Another example is the method rank, which return the

value of the rankValue data field.

Why is it better to use a method for this simple action, rather than permitting access to the data field directly? One reason is that the method makes the data field *read-only*. A function can only be called, whereas a data field can be both read and written. By the combination of a private data field and a public accessor we ensure the rank of the playing card cannot change once it has been created.

The naming conventions for the methods shown here are typical. It is good practice to name a method that returns a boolean value with a term that begins with is and indicates the meaning when a true value is returned. Following this convention makes it easy to understand the use of the method in a conditional statement, such as the following:

```
if (aCard.isFaceUp()) ...
```

Here aCard is once again an instance of class PlayingCard. Many style guidelines suggest that all other accessor methods should begin with the word get, so as to to most clearly indicate that the most important purpose of the method is to simply get the value of a data field. Again this convention makes it easy to understand statements that use this method:

```
int cardRank = aCard.getRank();
```

However, this convention is not universally advocated. In particular, we will continue to use the simpler names rank and suit for our methods.

Methods whose major purpose is simply to set a value are termed *mutator methods* or *setters*. As the name suggests, a setter most generally is named beginning with the word set. An example setter is the method setFaceUp which sets the value for the faceUp accessor:

```
class PlayingCard {
    ...
    void setFaceUp (boolean up) { faceUp = up; }
    ...
}
```

The method flip is neither a getter nor a setter, since it neither gets nor sets a data field. It is simply a method. The method color is not technically a getter, since it is not getting a data field held by the class. Nevertheless, because it is returning an attribute of the object, some style guidelines would suggest that a better name would be getColor.

Visibility modifiers are not found in the language Smalltalk. By default all data fields are private, that is accessible only within the class definition itself. To allow access to a data field an accessor method must be provided:

```
rank
```

```
" return the face value of a card "
↑ rank
```

The convention of using get names is not widely followed in Smalltalk. Instead, it is conventional for accessor methods to have the same name as the data field they are returning. No confusion arises in the Smalltalk system; we say nothing of confusion that can arise in the programmers mind.

4.3.1 Order of Methods in a Class Declaration

For the most part, programming languages do not specify the order that methods are declared within a class definition. However, the order can have a significant impact on readability, an issue that is often of critical importance to programmers. Many style guidelines offer differing advice on this issue, often conflicting with each other. The following are some of the most significant considerations:

- Important features should be listed earlier in the class definition, less important features listed later.
- Constructors are one of the most important aspects of an object definition, and hence should appear very near the top of a class definition.
- The declaration of methods should be grouped so as to facilitate rapidly finding the body associated with a given message selector. Ways of doing this include listing methods in alphabetical order, or grouping methods by their purpose.
- Private data fields are mostly important only to the class developer. They should be listed near the end of a class definition.

4.3.2 Constant or Immutable Data Fields

As an alternative to accessor methods, some programming languages provide a way to specify that a data field is constant, or *immutable*. This means that once set the value of the data field can not subsequently be changed. With this restriction there is less need to hide access to a data value behind a method.

Two different ways of describing constant data fields are shown in Figure 4.7. Such a field is declared as final in Java. The modifier const is used in C++ for much the same purpose.

4.3.3 Separating Definition and Implementation

Some languages, such as Java and C#, place the body of a method directly in the class definition, as shown in Figure 4.6. Other languages, such as C++ and Object Pascal, separate these two aspects. In C++ the programmer has a choice. Small methods can be defined in the class, while larger methods are

C++

```
class PlayingCard {
public:
    ....
    const int rank; // since immutable, can allow
    const Suits suit; // public access to data field
};
```

Java

```
class PlayingCard {
    ....
    public final int rank;
    public final int suit;
}
```

Figure 4.7: Syntax for Defining Immutable Data Values

defined outside. A C++ class definition for our playing card abstraction might look something like the following:

```
class PlayingCard {
public:
        // enumerated types
    enum Suits {Spade, Diamond, Club, Heart};
    enum Colors {Red, Black};

        // constructor, initialize new playing card
    PlayingCard (Suits is, int ir)
        { suit = is; rank = ir; faceUp = true; }

        // operations on a playing card
    boolean isFaceUp  ()        { return faceUp; }
    void    setFaceUp (bool up) { faceUp = up; }
    void    flip      ()        { setFaceUp( ! faceUp); }
    int     rank      ()        { return rankValue; }
    Suits   suit      ()        { return suitValue; }
    Colors  color     () ;
private:        // private data values
    Suits suitValue;
    int rankValue;
    boolean faceUp;
};
```

Notice the body of the method color has been omitted, as it is longer than the other methods defined in this class. A subsequent method definition (sometimes

termed a *function member*) provides the body of the function:

```
PlayingCard::Colors PlayingCard::color ( )
{
        // return the face color of a playing card
    if ((suit == Diamond) || (suit == Heart))
        return Red;
    return Black;
}
```

The method heading is very similar to a normal C style function definition, except that the name has been expanded into a *fully-qualified* name. The qualified name provides both the class name and the method name for the method being defined. This is analogous to identifying a person by both their given and family names (for example, "Chris Smith").

C++ programmers have the choice between defining methods in-line as part of the class definition, or defining them in a separate section of the program. Typically only methods that are one or two statements long are placed in-line, and anything more complex than one or two lines is defined outside the class.

There are two reasons for playing a method body outside the class definition. Method bodies that are longer than one statement can obscure other features of the class definition, and thus removing long method bodies can improve readability. (Readability, however, is in the eye of the beholder. Not all programmers think that this separation improves readability, since the programmer must now look in two different places to find a method body). A second reason involves semantics. When method bodies are declared within a class definition a C++ compiler is permitted (although not obligated) to expand invocations of the method directly in-line, without creating a function call. An inline definition can be executed much faster than the combination of function call and method body.

Often the class definition and the larger method bodies in a C++ program will not even be found in the same file. The class heading will be given in an *interface file* (by convention a file with the extension .h on Unix systems, or .hpp on Windows systems) while the function bodies will be found in an implementation file (by convention a file with the extension .cpp or .C).

Objective-C also separates a class definition from a class implementation. The definition includes a description of methods for the class. These are indicated by a + or − sign followed by the return type in parenthesis followed by a description of the method:

```
@ interface PlayingCard : Object
{
    int suit;
    int rank;
    int faceUp;
```

```
}

+ suit: (int) s rank: (int) i
- (int) color;
- (int) rank;
- (int) suit;
- (int) isFaceUp;
- (void) flip;
@ end
```

The implementation section then provides the body of the methods:

```
@ implementation PlayingCard

- (int) color
{
    if ((suit == Diamond) || (suit == Heart))
        return red;
    return black;
}

- (int) rank
{
    return rank;
}
    ... /* other method bodies */
@ end
```

Object Pascal and Delphi similarly separate the class definition from the method function bodies, however the two parts remain in the same file. The class definitions are described in a section labeled with the name interface, while the implementations are found in a section labeled, clearly enough, implementation. The following is a Delphi example:

```
interface

type
    Suits = (Heart, Club, Diamond, Spade);

    Colors = (Red, Black);

    TPlayingCard = class (TObject)
        public
            constructor Create (r : integer; s : Suits);
            function color : Colors;
```

```
            function isFaceUp : boolean;
            procedure flip;
            function rank : integer;
            function suit : Suits;
        private
            suit : Suits;
            rank : integer;
            faceUp : boolean;
    end;

implementation
    function TPlayingCard.color : Colors;
    begin
        case suit of
            Diamond: color := Red;
            Heart: color := Red;
            Spade: color := Black;
            Club: color := Black;
    end

    ... (* other methods similarly defined *)
end.
```

Note that fully qualified names in Pascal are formed using a period between the class name and the method name, instead of the double colon used by C++.

In CLOS accessor functions can be automatically created when a class is defined, using the :accessor keyword followed by the name of the accessor function:

```
(defclass PlayingCard ()
    ((rank :accessor getRank) (suit :accessor getSuit) ))
```

Other methods are defined using the function defmethod. Unlike Java or C++, the receiver for the method is named as an explicit parameter:

```
(defmethod color ((card PlayingCard))
    (cond
        ((eq (getSuit card) 'Diamond) 'Red)
        ((eq (getSuit card) 'Heart) 'Red)
        (t 'Black)))
```

The receiver must also be named as an explicit parameter in Python:

```
class PlayingCard:
    "A playing card class"
    def __init__ (self, s, r):
```

```
        self.suit = s
        self.rank = r
    def rank (self)
        return self.rank
    def color (self)
        if self.suit == 1 or self.suit == 2
            return 1
        return 0
```

4.4 Variations on Class Themes*

While the concept of a class is fundamental to object-oriented programming, some languages go further in providing variations on this basic idea. In the following sections we will describe some of the more notable among these variations.

4.4.1 Methods without Classes in Oberon

The language Oberon does not have classes in the sense of other object oriented languages, but only the more traditional concept of data records. Nevertheless, it does support message passing, including many of the dynamic method binding features found in object-oriented langauges.

A method in Oberon is not defined inside a record, but is instead declared using a special syntax where the receiver is described in an argument list separately from the other arguments. Often the receiver is required to be a pointer type, rather than the data record type:

```
TYPE
    PlayingCard = POINTER TO PlayingCardDesc;

    PlayingCardDesc = RECORD
        suit : INTEGER;
        rank : INTEGER;
        faceUp: BOOLEAN;
    END

PROCEDURE (aCard: PlayingCard) setFaceUp (b : BOOLEAN);
BEGIN
    aCard.faceUp = b;
END
```

[0] Section headings followed by an asterisk indicate optional material. Instructors should feel free to select subsections of this section that seem most appropriate for their own situation.

The record PlayingCardDesc contains the data fields, which can be modified by the procedure setFaceUp, which must take a pointer to a playing card as a receiver.

4.4.2 Interfaces

Some object-oriented languages, such as Java, support a concept called an *interface*. An interface defines the protocol for certain behavior but does not provide an implementation. The following is an example interface, describing objects that can read from and write to an input/output stream.

```
public interface Storing {
    void writeOut (Stream s);
    void readFrom (Stream s);
};
```

Like a class, an interface defines a new type. This means that variables can be declared simply by the interface name.

```
    Storing storableValue;
```

A class can indicate that it implements the protocol defined by an interface. Instances of the class can be assigned to variables declared as the interface type.

```
public class BitImage implements Storing {
    void writeOut (Stream s) {
        // ...
    }
    void readFrom (Stream s) {
        // ...
    }
};

    storableValue = new BitImage();
```

The use of interfaces is very similar to the concept of inheritance, and thus we will return to a more detailed consideration of interfaces in Chapter 8.

4.4.3 Properties

Delphi, Visual Basic, C# and other programming languages (both object-oriented and not) incorporate an idea called a property. A property is manipulated syntactically in the fashion of a data field, but operates internally like a method. That is, a property can be read as an expression, or assigned to as a value:

```
    writeln ('rank is ', aCard.rank); (* rank is property of card *)
    aCard.rank = 5; (* changing the rank property *)
```

However, in both cases the value assigned or set will be mediated by a function, rather than a simple data value. In Delphi a property is declared using the keyword property and the modifiers read and write. The values following the read and write keyword can be either a data field or a method name. The read attribute will be invoked when a property is used in the fashion of an expression, and the write attribute when the property is the target of an assignment. Having a read attribute and no write makes a property read only. We could recast our rank and suit values as properties as follows:

```
type
    TPlayingcard = class (TObject)
        public
            ...
            property rank : Integer read rankValue;
            property suit : Suits read suitValue write suitValue;
        private
            rankValue : Integer;
            suitValue : Suits;
    end;
```

Here we have made rank read only, but allowed suit to be both read and written. It is also possible to make a property write-only, although this is not very common.

In C# a property is defined by writing a method without an argument list, and including either a get or a set section.

```
public class PlayingCard {
    public int rank {
        get
        {
            return rankValue;
        }
        set
        {
            rankValue = value;
        }
    }
    ...
    private int rankValue;
}
```

A get section must return a value. A set section can use the pseudo-variable value to set the property. If no set section is provided the property is read-only. If no get section is given the property is write only. Properties are commonly used in C# programs for functions that take no arguments and return a value.

4.4.4 Forward Definitions

A program can sometimes require that two or more classes each have references to the other. This situation is termed *mutual recursion*. We might need to represent the horse and buggy trade, for example, where every horse is associated with their own buggy, and every buggy with one horse. Some languages will have little trouble with this. Java, for example, scans an entire file before it starts to generate code, and so classes that are referenced later in a file can be used earlier in a file with no conflict.

Other languages, such as C++, deal with classes and methods one by one as they are encountered. A name must have at least a partial definition before it can be used. In C++ this often results in the need for a *forward definition*. A definition that serves no other purpose than to place a name in circulation, leaving the completion of the definition until later. Our horse and buggy example, for instance, might require something like the following:

```
class Horse; // forward definition

class Buggy {
    ...
    Horse * myHorse;
};

class Horse {
    ...
    Buggy * myBuggy;
};
```

The first line simply indicates that Horse is the name of a class, and that the definition will be forthcoming shortly. Knowing only this little bit of information, however, is sufficient for the C++ compiler to permit the creation of a pointer to the unknown class.

Of course, nothing can be done with this object until the class definition has been read. Solving this problem requires a careful ordering of the class definitions and the the implementations of their associated methods. First one class definition, then the second class definition, then methods from the first class, finally methods from the second.

4.4.5 Inner or Nested Classes

Both Java and C++ allow the programmer to write one class definition inside of another. Such a definition is termed an *inner class* in Java, and a *nested class* in C++. Despite the similar appearances, there is a major semantic difference between the two concepts. An inner class in Java is linked to a specific instance of the surrounding class (the instance in which it was created), and is permitted access to data fields and methods in this object. A nested class in C++ is simply a naming device, it restricts the visibility of features associated with the inner class, but otherwise the two are not related.

To illustrate the use of nested classes, let us imagine that a programmer wants to write a doubly linked list abstraction in Java. The programmer might decide to place the Link class inside the List abstraction:

```
    // Java List class
class List {
    private Link firstElement = null;

    public void push_front(Object val)
    {
        if (firstElement == null)
            firstElement = new Link(val, null, null);
        else
            firstElement.addBefore (val);
    }

    ... // other methods omitted

    private class Link { // inner class definition
        public Object value;
        public Link forwardLink;
        public Link backwardLink;

        public Link (Object v, Link f, Link b)
            { value = v; forwardLink = f; backwardLink = b; }

        public void addBefore (Object val)
        {
            Link newLink = new Link(val, this, backwardLink);
            if (backwardLink == null)
                firstElement = newLink;
            else {
                backwardLink.forwardLink = newLink;
                backwardLink = newLink;
            }
```

```
        }
        ... // other methods omitted
    }
}
```

Note that the method addBefore references the data field firstElement, in order to handle the special case where an element is being inserted into the front of a list. A direct translation of this code into C++ will produce the following:

```
    // C++ List class
class List {
private:
    class Link;     // forward definition
    Link * firstElement;

    class Link { // nested class definition
    public:
        int value;
        Link * forwardLink;
        Link * backwardLink;

        Link (int v, Link * f, Link * b)
            { value = v; forwardLink = f; backwardLink = b; }

        void addBefore (int val)
        {
            Link * newLink = new Link(val, this, backwardLink);
            if (backwardLink == 0)
                firstElement = newLink; // ERROR !
            else {
                backwardLink->forwardLink = newLink;
                backwardLink = newLink;
            }
        }

        ... // other methods omitted
        };


public:
    void push_front(int val)
    {
        if (firstElement == 0)
            firstElement = new Link(val, 0, 0);
        else
```

```
            firstElement->addBefore (val);
    }
    ... // other methods omitted
};
```

It has been necessary to introduce a forward reference for the Link class, so that the pointer firstElement could be declared before the class was defined. Also C++ uses the value zero for a null element, rather than the pseudo-constant null. Finally links are pointers, rather than values, and so the pointer access operator is necessary. But the feature to note occurs on the line marked as an error. The class Link is not permitted to access the variable firstElement, because the scope for the class is not actually nested in the scope for the surrounding class. In order to access the List object, it would have to be explicitly available through a variable. In this case, the most reasonable solution would probably be to have the List method pass itself as argument, using the pseudo-variable this, to the inner Link method addBefore. (An alternative solution, having each Link maintain a reference to its creating List, is probably too memory intensive).

```
class List {
    Link * firstElement;

    class Link {
        void addBefore (int val, List * theList)
        {
            ...
            if (backwardLink == 0)
                theList->firstElement = newLink;
            ...
        }
    };
public:
    void push_front(int val)
    {
            ...
                // pass self as argument
            firstElement->addBefore (val, this);
    }
    ... // other methods omitted
};
```

When nested class methods are defined outside the class body, the name may require multiple levels of qualification. The following, for example, would be how the method addBefore would be written in this fashion:

```
    void List::Link::addBefore (int val, List * theList)
```

```
{
    Link * newLink = new Link(val, this, backwardLink);
    if (backwardLink == 0)
        theList->firstElement = newLink;
    else {
        backwardLink->forwardLink = newLink;
        backwardLink = newLink;
    }
}
```

The name of the function indicates that this is the method addBefore that is part of the class Link, which is in turn defined as part of the class List.

4.4.6 Class Data Fields

In many problems it is useful to have a common data field that is shared by all instances of a class. However, the manipulation of such an object creates a curious paradox for the object oriented language designer. To understand this problem, consider that the reason for the invention of the concept of a class was to reduce the amount of work necessary to create similar objects; every instance of a class has exactly the same behavior as every other instance. Now imagine that we have somehow defined a common data area shared by all instances of a class, and think about the task of initializing this common area. There seem to be two choices, neither satisfactory. Either everybody performs the initialization task (and the field is initialized and reinitialized over and over again), or nobody does (leaving the data area uninitialized).

Resolving this paradox requires moving outside of the simple class/method/instance paradigm. Another mechanism, not the objects themselves, must take responsibility for the initialization of shared data. If objects are automatically initialized to a special value (such as zero) by the memory manager, then every instance can test for this special value, and perform initialization if they are the first. However, there are other (and better) techniques.

In both C++ and Java shared data fields are created using the static modifier. We have seen a use of this already in the creation of symbolic constants in Java. In Java the intialization of a static data field is accomplished by a *static block*, which is executed when the class is loaded. For example, suppose we wanted to keep track of how many instances of a class have been created:

```
class CountingClass {

    CountingClass () {
        count = count + 1; // increment count
        ...
    }
```

```
    ...

    private static int count; // shared by all

    static {    // static block
        count = 0;
    }
}
```

In C++ there are two different mechanisms. Data fields that are static (or const) and represented by primitive data types can be initialized in the class body, as we have seen already. Alternatively, a global initialization can be defined that is separate from the class:

```
class CountingClass {
public:
    CountingClass () { count++; ... }

private:
    static int count;
};

// global initialization is separate from class
int CountingClass::count = 0;
```

In C# static data fields can be initialized by a static constructor, a constructor method that is declared static. This constructor is not permitted to have any arguments.

In Python class data fields are simply named at the level of methods, while instance variables are named inside of methods (typically inside the constructor method):

```
class CountingClass:
    count = 0
    def __init__ (self)
        self.otherField = 3
```

4.4.7 Classes as Objects

In a number of languages (Smalltalk, Java, many others) a class is itself an object. Of course, one must then ask what class represents the category to which this object belongs, that is, what class is the class? In most cases there is a special class, typically Class, that is the class for classes.

Since objects are classes, they have behavior. What can you do with a class? Frequently the creation of an instance of the class is simply a message given to a class object. This occurs in the following example from Smalltalk, for instance:

```
aCard <- PlayingCard new. "message new given to object PlayingCard"
```

Other common behaviors include returning the name of the class, the size of instances of the class, or a list of messages that instances of the class will recognize. The following bit of Java code illustrates one use:[1]

```
    Object obj = new PlayingCard();
    Class c = obj.getClass();
    System.out.println("class is " + c.getName());
PlayingCard
```

We wil return to an exploration of classes as objects when we investigate the concept of *reflection* in Chapter 25.

Chapter Summary

In this chapter we have started our exploration of the concept of *class* in object-oriented languages. We have described the syntax for class and method definitions in various languages, including Java, C++, C#, Object Pascal, Objective-C, and Eiffel. Throughout the text we will occasional refer to other example languages as well.

Some of the features of classes that we have seen in this chapter include the following:

- Visibility modifiers. The keywords public and private that are used to control the visibility, and hence the manipulation, of class features.
- Getter and Setter functions. Sometimes termed accessors and mutators, these are methods that provide access to data fields. By using methods rather than providing direct access, programmers have greater control over the way data is modified, and where it can used.
- Constant, or immutable data fields. Data fields that are guaranteed to not change during the course of execution.
- Interfaces. Class like entities that describe behavior, but do not provide an implementation.

[1] In non interactive languages it is sometimes difficult to show the relationship between program statements and their output. Throughout the rest of the book we will use the convention illustrated by this example, indenting a sequence of statement and then showing the resulting output without indentation. The reader will hopefully be able to distinguish the executable statements from the non-executable output.

- Nested classes. Class definitions that appear inside of other class definitions.
- Class data fields. The particular paradox that arises over the initialization data fields that are shared in common among all instances of a class.

Further Reading

The Apple Object Pascal language was original defined by Larry Tesler of Apple Computer [Tesler 1985]. The Borland language was originally known as Turbo Pascal [Turbo 1988]. More recent descriptions of the Delphi version of this language can be found in [Lischner 2000, Kerman 2002].

The classic definition of the language Smalltalk is [Goldberg 1983]. More recent treatments of the language include [LaLonde 1990b, Smith 1995]. A popular public-domain version of Smalltalk is Squeak [Guzdial 2001].

The Java language is described in [Arnold 2000]. A Good tutorial on Java can be found in [Campione 1998]. In Java and C++ the concept of *interfaces* is closely related to the concept of a class. We will discuss interfaces when we examine inheritance in Chapter 8. A good style guidebook for Java programmers is [Vermeulen 2000].

Since C# is a relatively recent language there are still only a few published references. Two recent sources are [Gunnerson 2000, Albahari 2001].

Objective-C was created as an extension to C at about the same time that C++ was developed. A good introduction Objective-C is the book written by its creator, Brad Cox [Cox 1986]. Python is described in [Beazley 2000].

An interesting question is whether classes are necessary for object-oriented programming. It turns out that you can achieve much of the desirable characteristics of object-oriented languages without using classes, by means of an idea termed *delegation* [Lieberman 1986]. However, in the period between when delegation languages were first proposed and the present there has not been a ground-swell of support for this idea, so most people seem to prefer classes.

Self Study Questions

1. What is the difference between a class declaration and an object declaration (the latter also known as an instantiation)?
2. What is an instance variable?
3. What are the two most basic aspects of a class?
4. What is the meaning of the modifiers final and static in Java? How do these two features combine to form a symbolic constant?
5. What does the term public mean? What does the term private mean?

6. What is a constructor?

7. What is an accessor method? What is the advantage of using accessor methods instead of providing direct access to a data field?

8. What is a mutator, or setter method?

9. What are some guidelines for selecting the order of features in a class definition?

10. What is an immutable data field?

11. What is a fully qualified name?

12. How is an interface different from a class? How is it similar?

13. What is an inner or nested class?

14. Explain the paradox arising the the initialization of common data fields or class data fields.

Exercises

1. Suppose you were required to program in a non-object-oriented language, such as Pascal or C. How would you simulate the notion of classes and methods?

2. In Smalltalk and Objective-C, methods that take multiple arguments are described using a keyword to separate each argument; in C++ the argument list follows a single method name. Describe some of the advantages and disadvantages of each approach; in particular, explain the effect on readability and understandability.

3. A digital counter is a bounded counter that turns over when its integer value reaches a certain maximum. Examples include the numbers in a digital clock and the odometer in a car. Define a class description for a bounded counter. Provide the ability to set maximum and minimum values, to increment the counter, and to return the current counter value.

4. Write a class description for complex numbers. Write methods for addition, subtraction, and multiplication of complex numbers.

5. Write a class description for a fraction, a rational number composed of two integer values. Write methods for addition, subtraction, multiplication, and division of fractions. How do you handle the reduction of fractions to lowest-common-demoninator form?

6. Consider the following two combinations of class and function in C++. Explain the difference in using the function addi as the user would see it.

```
class example1 {
public:
    int i;
};

int addi(example1 & x, int j)
{
    x.i = x.i + j;
    return x.i;
}


class example2 {
public:
    int i;
    int addi(int j)
        { i = i + j; return i; }
};
```

7. In both the C++ and Objective-C versions of the playing card abstraction, the modular division instruction is used to determine the color of a card based on the suit value. Is this a good practice? Discuss a few of the advantages and disadvantages. Rewrite the methods to remove the dependency on the particular values associated with the suits.

8. Do you think it is better to have the access modifiers private and public associated with every individual object, as in Java, or used to create separate areas in the declaration, as in C++, Objective-C, and Delphi Pascal? Give reasons to support your view.

9. Contrast the encapsulation provided by the class mechanism with the encapsulation provided by the module facility. How are they different? How are they the same?

Chapter 5

Messages, Instances, and Initialization

In Chapter 4 we briefly outlined some of the compile-time features of object-oriented programming languages. That is, we described how to create new types, new classes, and new methods. In this chapter, we continue our exploration of the mechanics of object-oriented programming by examining the *dynamic* features. These include how values are instantiated (or created), how they are initialized, and how they communicate with each other by means of message passing.

In the first section, we will explore the mechanics of message passing. Then we will investigate creation and initialization. By *creation* we mean the allocation of memory space for a new object and the binding of that space to a name. By *initialization* we mean not only the setting of initial values in the data area for the object, similar to the initialization of fields in a record, but also the more general process of establishing the initial conditions necessary for the manipulation of an object. The degree to which this latter task can be hidden from clients who use an object in most object-oriented languages is an important aspect of *encapsulation*, which we identified as one of the principle advantages of object-oriented techniques over other programming styles.

5.1 Message-Passing Syntax

We are using the term *message passing* (sometimes also called *method lookup*) to mean the dynamic process of asking an object to perform a specific action. In Chapter 1 we informally described message passing and noted how a message differs from an ordinary procedure call. In particular:

- A *message* is always given *to* some object, called the *receiver*.
- The action performed in response to the message is not fixed, but may differ depending upon the class of the receiver. That is, different objects

C++, C#, Java, Python, Ruby	`aCard.flip ();`
	`aCard.setFaceUp(true);`
	`aGame.displayCard(aCard, 45, 56);`

Pascal, Delphi, Eiffel, Oberon	`aCard.flip;`
	`aCard.setFaceUp(true);`
	`aGame.displayCard(aCard, 45, 56);`

Smalltalk	`aCard flip.`
	`aCard setFaceUp: true.`
	`aGame display: aCard atLocation: 45 and: 56.`

Objective-C	`[ aCard flip ].`
	`[ aCard setFaceUp: true ].`
	`[ aGame display: aCard atLocation: 45 and: 56 ]`

CLOS	`(flip aCard)`
	`(setFaceUp aCard true)`
	`(displayCard aGame 45 56)`

Figure 5.1: Message Passing Syntax in various Languages

may accept the same message, and yet perform different actions.

There are three identifiable parts to any message-passing expression. These are the *receiver* (the object to which the message is being sent), the *message selector* (the text that indicates the particular message being sent), and the *arguments* used in responding to the message.

$$\underbrace{\mathsf{aGame}}_{receiver}.\underbrace{\mathsf{displayCard}}_{selector}\underbrace{(\mathsf{aCard}, 42, 27)}_{arguments}$$

As Figure 5.1 indicates, the most common syntax for message passing uses a period to separate the receiver from the message selector. Minor variations include features such as whether an empty pair of parenthesis are required when a method has no arguments (they can be omitted in Pascal and some other languages).

Smalltalk and Objective-C use a slightly different syntax. In these languages a space is used as a separator. Unary messages (messages that take no argument) are simply written following the receiver. Messages that take arguments are written using *keyword notation*. The message selector is split into parts, one part before each argument. A colon follows each part of the key.

```
aGame display: aCard atLocation: 45 and: 56.
```

In Smalltalk even binary operations, such as addition, are interpreted as a message sent to the left value with the right value as argument:

```
z <- x + y.  " message to x to add y to itself and return sum "
```

It is possible to define binary operators in C++ to have similar meanings. In Objective-C a Smalltalk-like message is enclosed in a pair of square braces, termed a *message passing expression*. The brackets only surround the message itself. They do not, for example, surround an assignment the places the result of a message into a variable:

```
int cardrank = [ aCard getRank ];
```

The syntax used in CLOS follows the traditional Lisp syntax. All expressions in Lisp are written as parenthesis-bounded lists. The operation is the first element of the list, followed by the arguments. The receiver is simply the first argument.

5.2 Statically and Dynamically Typed Languages

Languages can be divided into two groups depending upon whether they are statically or dynamically typed. Fundamentally, a statically typed language associates types with variables (usually the binding is established by means of declaration statements), while a dynamically typed language treats variables simply as names, and associates types with values. Java, C++, C#, and Pascal are statically typed languages, while Smalltalk, CLOS and Python are dynamically typed.

Objective-C holds a curious middle ground between the two camps. In Objective-C a variable can be declared with a fixed type, and if so the variable is statically typed. On the other hand, a variable can also be declared using the object type id. A variable declared in this fashion can hold any object value, and hence is dynamically typed.

```
PlayingCard aCard;  /* a statically typed variable */
id anotherCard; /* a dynamically typed variable */
```

The difference between statically typed languages and dynamically typed languages is important in regards to message passing because a statically typed language will use the type of the receiver to check, at compile time, that a receiver will understand the message it is being presented. A dynamically typed language, on the other hand, has no way to verify this information at compile time. Thus in a dynamically typed language a message can generate a run-time

error if the receiver does not understand the message selector. Such a run-time error can never occur in a statically typed language.

5.3 Accessing the Receiver from Within a Method

As we indicated at the beginning of this chapter, a message is always passed to a receiver. In most object-oriented languages, however, the receiver does not appear in the argument list for the method. Instead, the receiver is only implicitly involved in the method definition. In those rare situations when it is necessary to access the receiver value from within a method body a *pseudo-variable* is used. A pseudo-variable is like an ordinary variable, only it need not be declared and cannot be modified. (The term *pseudo constant* might therefore seem more appropriate, but this term does not seem to be used in any language definitions).

The pseudo-variable that designates the receiver is named this in Java and C++, Current in Eiffel, and self in Smalltalk, Objective-C, Object Pascal and many other languages. The pseudo-variable can be used as if it refers to an instance of the class. For example, the method color could be written in Pascal as follows:

```
function PlayingCard.color : colors;
begin
    if (self.suit = Heart) or (self.suit = Diamond) then
        color := Red
    else
        color := Black;
end
```

In most languages the majority of uses of the receiver pseudo-variable can be omitted. If a data field is accessed or a method is invoked without reference to a receiver it is implicitly assumed that the receiver pseudo-variable is the intended basis for the message. We saw this earlier in the method flip, which acted by invoking the method setFaceUp:

```
class PlayingCard {
    ...
    public void flip () { setFaceUp( ! faceUp ); }
    ...
}
```

The method could be rewritten to make the receivers explicit as follows:

```
class PlayingCard {
    ...
```

```
    public void flip () { this.setFaceUp( ! this.faceUp); }
    ...
}
```

One place where the use of the variable often cannot be avoided is when a method wishes to pass itself as an argument to another function, as in the following bit of Java:

```
class QuitButton extends Button implements ActionListener {
    public QuitButton () {
        ...
            // install ourselves as a listener for button events
        addActionListener(this);
    }
    ...
};
```

Some style guidelines for Java suggest the use of this when arguments in a constructor are used to initialize a data member. The same name can then be used for the argument and the data member, with the explicit this being used to distinguish the two names:

```
class PlayingCard {
    public PlayingCard (int suit, int rank) {
        this.rank = rank; // this.rank is the data member
        this.suit  = suit;  // rank is the argument value
        this.faceUp = true;
    }
    ...
    private int suit;
    private int rank;
    private boolean faceUp;
}
```

A few object-oriented languages, such as Python, CLOS, or Oberon, buck the trend and require that the receiver be declared explicitly in a method body. In Python, for example, a message might appear to have two arguments, as follows:

```
    aCard.moveTo(27, 3)
```

but the corresponding method would declare three parameter values:

```
class PlayingCard:
    def moveTo (self, x, y):
        ...
```

While the first argument could in principle be named anything, it is common to name it self or this, so as to indicate the association with the receiver pseudo-variables in other languages. Examples in the previous chapter illustrated the syntax used by CLOS and Oberon, which also must name the receiver as an argument.

5.4 Object Creation

In most conventional programming languages, variables are created by means of a declaration statement, as in the following Pascal example:

```
var
    sum : integer;
begin
    sum := 0.0;
    ...
end;
```

Some programming languages allow the user to combine declaration with initialization, as in the following Java example:

```
int sum = 0.0; // declare and initialize variable with zero
...
```

A variable declared within the bounds of a function or procedure generally exists only as long as the procedure is executing. The same is true for some object-oriented languages. The following declaration statement, for example, can be used to create a variable in C++:

```
PlayingCard aCard(Diamond, 4);  // create 4 of diamonds
```

Most object oriented languages, however, separate the process of variable naming from the process of object creation. The declaration of a variable only creates the name by which the variable will be known. To create an object value the programmer must perform a separate operation. Often this operation is denoted by the operator new, as in this Smalltalk example:

```
| aCard |  " name a new variable named aCard "

aCard <- PlayingCard new.  " allocate memory space to variable "
```

C++	`PlayingCard * aCard = new PlayingCard(Diamond, 3);`
Java, C#	`PlayingCard aCard = new PlayingCard(Diamond, 3);`

Object Pascal	`var`
	`  aCard : ^ PlayingCard;`
	`begin`
	`  new (aCard);`
	`  ...`
	`end`

Objective-C	`aCard = [ PlayingCard new ];`
Python	`aCard = PlayingCard(2, 3)`
Ruby	`aCard = PlayingCard.new`
Smalltalk	`aCard <- PlayingCard new.`

Figure 5.2: Syntax Used for Object Creation

The syntax used in object creation for various different languages is shown in Figure 5.2. Python does not use the `new` operator explicitly, instead, in Python creation occurs when a class name is used in the fashion of a function.

5.4.1 Creation of Arrays of Objects

The creation of an array of objects presents two levels of complication. There is the allocation and creation of the array itself, and then the allocation and creation of the objects that the array will hold.

In C++ these features are combined, and an array will consist of objects that are each initialized using the default (that is, no-argument) constructor (see Section 5.6):

```
    // create an array of 52 cards, all the same
PlayingCard cardArray [52];
```

In Java, on the other hand, a superficially similar statement has a very different effect. The `new` operator used to create an array creates only the array. The values held by the array must be created separately, typically in a loop:

```
PlayingCard cardArray[ ] = new PlayingCard[13];
for (int i = 0; i < 13; i++)
```

```
cardArray[i] = new PlayingCard(Spade, i+1);
```

A frequent source of error for C or C++ programmers moving to Java is to forget that in Java the allocation of an array occurs separately from the allocation of the elements the array will contain.

5.5 Pointers and Memory Allocation

All object-oriented languages use pointers in their underlying representation. Not all languages expose this representation to the programmer. It is sometimes said that "Java has no pointers" as a point of contrast to C++. A more accurate statement would be that Java has no pointers that the programmer can see, since all object references are in fact pointers in the internal representation.

The issue is important for three reasons. Pointers normally reference memory that is *heap allocated*, and thus does not obey the normal rules associated with variables in conventional imperative languages. In an imperative language a value created inside a procedure will exist as long as the procedure is active, and will disappear when the procedure returns. A heap allocated value, on the other hand, will continue to exist as long as there are references to it, which often will be much longer than the lifetime of the procedure in which it is created.

The second reason is that heap based memory must be recovered in one fashion or another, a topic we will address in the next section.

A third reason is that some languages, notably C++, distinguish between conventional values and pointer values. In C++ a variable that is declared in the normal fashion, a so-called *automatic* variable, has a lifetime tied to the function in which it is created. When the procedure exits, the memory for the variable is recovered:

```
void exampleProcedure
{
    PlayingCard ace(Diamond, 1);
    ...
    // memory is recovered for ace
    // at end of execution of the procedure
}
```

Values that are assigned to pointers (or as *references*, which are another form of pointers) are not tied to procedure entry. Such values differ from automatic variables in a number of important respects. As we will note in the next section, memory for such values must be explicitly recovered by the programmer. When we introduce inheritance in Chapter 8 we will see that such values also differ in the way they use that feature.

5.5.1 Memory Recovery

Memory created using the new operator is known as *heap-based* memory, or simply *heap* memory. Unlike ordinary variables, heap-based memory is not tied to procedure entry and exit. Nevertheless, memory is always a finite commodity, and hence some mechanism must be provided to recover memory values. Memory that has been allocated to object values is then recycled and used to satisfy subsequent memory requests.

There are two general approaches to the task of memory recovery. Some languages (such as C++ and Delphi Pascal) insist the programmer indicate when an object value is no longer being used by a program, and hence can be recovered and recycled. The keywords used for this purpose vary from one language to another. In Object Pascal the keyword is free, as in the following example:

```
free aCard;
```

Objective-C uses the same keyword, but written as a message, with the receiver first:

```
[ aCard free ];
```

In C++ the keyword is delete:

```
delete aCard;
```

When an array is deleted a pair of square braces must be placed after the keyword:

```
delete [ ] cardArray;
```

The alternative to having the programmer explicitly manage memory is an idea termed *garbage collection*. A language that uses garbage collection (such as Java, C#, Smalltalk or CLOS) monitors the manipulation of object values, and will automatically recover memory from objects that are no longer being used. Generally garbage collection systems wait until memory is nearly exhausted, then will suspend execution of the running program while they recover the unused space, before finally resuming execution. Garbage collection uses a certain amount of execution time, which may make it more costly than the alternative of insisting that programmers free their own memory. But garbage collection prevents a number of common programming errors:

- It is not possible for a program to run out of memory because the programmer forgot to free up unused memory. (Programs can still run out of memory if the total memory required at any one time exceeds the available memory, of course).

- It is not possible for a programmer to try to use memory after it has been freed. Freed memory can be reused, and hence the contents of the memory values may be overwritten. Using a value after it has been freed can therefore cause unpredictable results.

```
PlayingCard * aCard = new PlayingCard(Spade, 1);
    ...
delete aCard;
    ...
cout << aCard.rank(); // attempt to use after deletion
```

- It is not possible for a programmer to try and free the same memory value more than once. Doing this can also cause unpredictable results.

```
Playingcard * aCard = new PlayingCard(Space, 1);
    ...
delete aCard;
    ...
delete aCard; // deleting already deleted value
```

When a garbage collection system is not available, to avoid these problems it is often necessary to ensure that every dynamically allocated memory object has a designated *owner*. The owner of the memory is responsible for ensuring that the memory location is used properly and is freed when it is no longer required. In large programs, as in real life, disputes over the ownership of shared resources can be a source of difficulty.

When a single object cannot be designated as the owner of a shared resource, another common technique is to use *reference counts*. A reference count is a count of the number of pointers that reference the shared object. Care is needed to ensure that the count is accurate; whenever a new pointer is added the count is incremented, and whenever a pointer is removed the count is decremented. When the count reaches zero it indicates that no pointers refer to the object, and its memory can be recovered.

As with the arguments for and against dynamic typing, the arguments for and against garbage collection tend to pit efficiency against flexibility. Automatic garbage collection can be expensive, as it necessitates a run-time system to manage memory. On the other hand, the cost of memory errors can be equally expensive.

5.6 Constructors

As we indicated in Chapter 4, a constructor is a method that is used to initialize a newly created object. Linking creation and initialization together has many beneficial consequences. Most importantly, it guarantees that an object can never

be used before it has been properly initialized. When creation and initialization are separated (as they must be in languages that do not have constructors), a programmer can easily forget to call an initialization routine after creating a new value, often with unfortunate consequences. A less common problem, although often just as unfortunate, is to invoke an initialization procedure twice on the same value. This problem, too, is avoided by the use of constructors.

In Java and C++ a constructor can be identified by the fact that it has the same name as the class in which it appears. Another small difference is that constructors do not declare a return type:

```
class PlayingCard {     // a Java constructor
    public PlayingCard (int s, int r) {
        suit = s;
        rank = r;
        faceUp = true;
    }
    ...
}
```

When memory is allocated using the new operator, any arguments required by the constructor appear following the class name:

```
    aCard = new PlayingCard(PlayingCard.Diamond, 3);
```

Data fields in Java (as well as in C#) that are initialized with a simple value, independent of any constructor argument, can be assigned a value at the point they are declared, even if they are subsequently reassigned:

```
class Complex { // complex numbers
    public Complex (double rv) { realPart = rv; }

    public double realPart = 0.0;  // initialize data areas
    public double imagPart = 0.0; // to zero
}
```

A similar syntax can be used in C++ if the data members are are declared to be static and/or const.

In C++ and Java there can be more than one function definition that uses the same name, as long as the number, type and order of arguments are sufficient to distinguish which function is intended in any invocation. This facility is frequently used with constructors, allowing the creation of one constructor to be used when no arguments are provided, and another to be used with arguments:

```
class PlayingCard {
public:
```

```
    PlayingCard ( )  // default constructor,
            // used when no arguments are given
        { suit = Diamond; rank = 1; faceUp = true; }

    PlayingCard (Suit is) // constructor with one argument
        { suit = is; rank = 1; faceUp = true; }

    PlayingCard (Suit is, int ir) // constructor with two arguments
        { suit = is; rank = ir; faceUp = true; }
};
```

The combination of number, type and order of arguments is termed a function *type signature*. We say that the meaning of an overloaded constructor (or any other overloaded function, for that matter) is resolved by examining the type signature of the invocation.

```
    PlayingCard cardOne; // invokes default
    PlayingCard * cardTwo = new PlayingCard;
    PlayingCard cardThree(PlayingCard.Heart);
    PlayingCard * cardFour = new PlayingCard(PlayingCard.Spade, 6);
```

In C++ one must be careful to omit the parenthesis from an invocation of the default constructor. Using a parenthesis in this situation is legal, but has an entirely different meaning:

```
    PlayingCard cardFive; // creates a new card
    PlayingCard cardSix(); // forward definition for function
                // named cardSix that returns a PlayingCard
```

On the other hand, when using the `new` operator and no arguments, parenthesis are omitted in C++ but not in Java or C#:

```
    PlayingCard cardSeven = new PlayingCard(); // Java
    PlayingCard * cardEight = new PlayingCard; // C++
```

Constructors in C++ can also use a slightly different syntax to specify the initial value for data members. A colon, followed by a named value in parenthesis, is termed an *initializer*. Our constructor written using initializer syntax would look as follows:

```
Class PlayingCard {
public:
    PlayingCard (Suits is, int ir)
        : suit(is), rank(ir), faceUp(true) { }
    ...
```

```
};
```

For simple values such as integers there is no difference between the use of an initializer and the use of an assignment statement within the body of the constructor. We will subsequently encounter different forms of initialization that can only be performed in C++ by means of an initializer.

Constructors in Objective-C need not have the same name as the class, and are signified by the use of a plus sign, rather than a minus sign, in the first column of their definition. Such a function is termed a *factory method*. The factory method uses the new operator to perform the actual memory allocation, then performs whatever actions are necessary to initialize the object.

```
@ implementation PlayingCard

+  suit: (int) s rank: (int) r {
    self = [ Card new ];
    suit = s;
    rank = r;
    return self;
}

@end
```

Factory methods are invoked using the class as the receiver, rather than an instance object:

```
PlayingCard aCard = [ PlayingCard suit: Diamond rank: 3 ];
```

Constructors in Python all have the unusual name `__init__`. When an object is created, the init function is implicitly invoked, passing as argument the newly created object, and any other arguments used in the creation expression:

```
aCard = PlayingCard(2, 3)
    # invokes PlayingCard.__init__(aCard, 2 3)
```

In Apple Object Pascal there are no constructors. New objects are created using the operator new, and often programmers define their own initialization routines that should be invoked using the newly created object as receiver. The Delphi version of Object Pascal is much closer to C++. In Delphi programmers can define a constructor, although unlike C++ this function need not have the same name as the class. It is typical (although not required) to use the name Create as a constructor name:

```
interface
    type
```

```
        TPlayingCard = class (TObject)
            constructor Create (is : Suits, ir : integer);
            ...
        end;
implementation
    constructor TPlayingCard.Create (is : Suits, ir : integer);
    begin
        suit = is;
        rank = ir;
        faceUp = true;
    end;
```

New objects are then created using the constructor method with the class as receiver:

```
    aCard := TPlayingCard.Create (Spade, 4);
```

5.6.1 The Orthodox Canonical Class Form*

Several authors of style guides for C++ have suggested that almost all classes should define four important functions. This has come to be termed the *orthodox canonical class* form. The four important functions are:

- A default constructor. This is used internally to initialize objects and data members when no other value is available.
- A copy constructor. This is used, among other places, in the implementation of call-by-value parameters.
- An assignment operator. This is used to assign one value to another.
- A destructor. This is invoked when an object is deleted. (We will shortly give an example to illustrate the use of destructors).

A default constructor we have seen already. This is simply a constructor that takes no arguments. A copy constructor takes a reference to an instance of the class as argument, and initializes itself as a copy of the argument:

```
class PlayingCard {
public:
    ...
    PlayingCard (PlayingCard & aCard)
    {
            // initialize ourself as copy of argument
```

[0] Section headings followed by an asterisk indicate optional material.

```
        rank = aCard.getRank();
        suit = aCard.getSuit();
        faceUp = aCard.isFaceUp();
    }
    ...
};
```

The system will implicitly create default versions of each of these if the user does not provide an alternative. However, in many situations (particuarly those involving the management of dynanically allocated memory) the default versions are not what the programmer might wish. Even if empty bodies are supplied for these functions, writing the class body will at least suggest that the program designer has *thought* about the issues involved in each of these. Furthermore, appropriate use of visibility modifiers give the programmer great power is allowing or disallowing different operations used with the class.

5.6.2 Constant Values

In Chapter 4 we pointed out that some languages, such as C++ and Java, permit the creation of data fields that can be assigned once and thereafter are not allowed to change. Having introduced constructors, we can now complete that discussion by showing how such values can be initialized.

In Java an immutable data field is simply declared as final and can be initialized directly:

```
class ListofImportantPeople {
public:
    final int max = 100; // maximum number of people
    ...
}
```

Alternatively, a final value can be assigned in the constructor. If there is more than one constructor, each constructor must initialize the data field:

```
class PlayingCard {
    public PlayingCard ( )
        { suit = Diamond; rank = 1; faceUp = true; }
    public PlayingCard ( int is, int ir)
        { suit = is; rank = ir; faceUp = true; }
    ...
    public final int suit;  // suit and rank are
    public final int rank; // immutable
    private boolean faceUp; // faceUp is not
}
```

Immutable values in C++ are designated using the keyword const. They can only be given a value using an initializer clause in a constructor:

```
class PlayingCard {
public:
    PlayingCard () : suit(Diamond), rank(1) { faceUp = true; }
    PlayingCard (Suits is, int ir) : suit(is), rank(ir)
        { faceUp = true; }
    ...
    const Suits suit;
    const int rank;
private:
    boolean faceUp;
};
```

There is one subtle but nevertheless important difference between const and final values. The const modifier in C++ says that the associated value is truly constant, and is not allowed to change. The final modifier in Java only asserts that the associated variable will not be assigned a new value. Nothing prevents the value itself from changing its own internal state, for example in response to messages. To illustrate this, consider the following definition for a data type named Box.

```
class Box {
    public void setValue (int v);
    public int getValue () { return v; }
    private int v = 0;
}
```

Declaring a variable using the final modifier simply means it will not be reassigned, it does not mean it will not change:

```
final aBox = new Box(); // can be assigned only once
aBox.setValue(8); // but can change
aBox.setValue(12); // as often as you like
```

A variable declared using the const modifier in C++, on the other hand, is not allowed to change in any way, not even in its internal state. (Individual fields can be named as mutable, in which case they are allowed to change even within a const object. However, use of this facility is rare.)

5.7 Destructors and Finalizers

A constructor allows the programmer to perform certain actions when an object value is created, when it is being born (so to speak). Occasionally it is useful to also be able to specify actions that should be performed at the other end of a values lifetime, when the variable is about to die and have its memory recovered.

This can be performed in C++ using a method termed a *destructor*. The destructor is invoked automatically whenever memory space for an object is released. For automatic variables, space is released when the function containing the declaration for the variable is existed. For dynamically allocated variable space is released with the operator delete. The destructor function is written as the name of the class preceded by a tilde (~). It does not take any arguments and is never directly invoked by the user.

A simple but clever function will illustrate the use of constructors and destructors. The class Trace defines a simple class that can be used to trace the flow of execution. The constructor class takes as argument a descriptive string and prints a message when space for the associated variable is allocated (which is when the procedure containing the declaration is entered). A second message is printed by the destructor when space for the variable is released, which occurs when the procedure is exited.

```
class Trace {
public:
    // constructor and destructor
    Trace    (string);
    ~Trace    ();
private:
    string text;
};

Trace::Trace (string t) : text(t)
{     cout << "entering " << text << endl; }

Trace::~Trace ()
{     cout << "exiting " << text << endl; }
```

To trace the flow of execution, the programmer simply creates a declaration for a dummy variable of type Trace in each procedure to be traced. Consider the following pair of routines:

```
void procedureA ()
{
    Trace dummy ("procedure A");
    procedureB (7);
}
```

```
void procedureB (int x)
{
    Trace dummy ("procedure B");
    if (x < 5) {
        Trace aaa("true case in Procedure B");
        ...
        }
    else {
        Trace bbb("false case in Procedure B");
        ...
        }
}
```

By their output, the values of type Trace will trace out the flow of execution. A typical output might be:

```
entering procedure A
entering procedure B
entering false case in Procedure B
...
exiting false case in Procedure B
exiting procedure B
exiting procedure A
```

Delphi Pascal also supports a form of destructor. A destructor function (usually called Destroy) is declared by the keyword destructor. When a dynamically allocated object is freed, the memory management system will call the destructor function.

```
type
    TPlayingCard = class (TObject)
        ...
        destructor Destroy;
    end;

destructor PlayingCard.Destroy;
begin
    (* whatever housekeeping is necessary *)
    ...
end;
```

Java and Eiffel have similar facilities, although since both languages use garbage collection their utilization is different. A method named finalize in Java will be invoked just before the point where a variable is recovered by the garbage

collection system. Since this can occur at any time, or may never occur, the use of this facility is much less common than the use of destructors in C++.

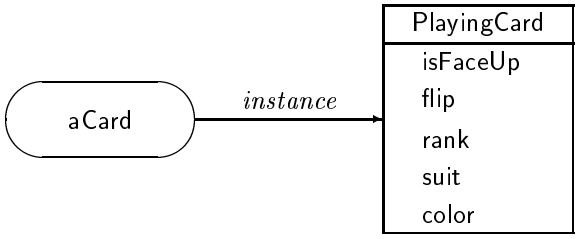
```
class FinalizeExample {
    public void finalize () {
        System.out.println("finally doing finalization");
        System.exit(0);
    }
}

    ...

        // first create an instance
    Object x = new FinalizeExample();
        // redefining x releases memory
    x = new Integer(3);
        // now do lots of memory allocations
        // at some indeterminent point garbage collection
        // will occur and final method will be called
    for (int i = 0; i < 1000; i++) {
        System.out.println("i is " + i);
        for (int j = 0; j < 1000; j++)
            x = new Integer(j);
    }
```

In Eiffel the same effect is achieved by inheriting from the class Memory and overriding the method dispose. (We will discuss inheritance and overriding later in Chapter 8.)

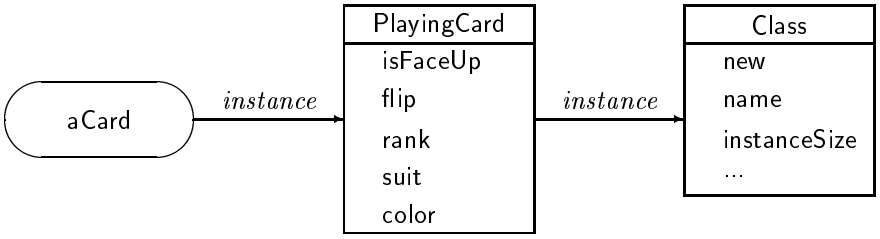
5.8 Metaclasses in Smalltalk*

The discussion of object creation provides an excuse to introduce a curious concept found in Smalltalk and a few similar languages, termed *metaclasses*. To understand metaclasses, note first that methods are associated not with objects, but with classes. That is, if we create a playing card, the methods associated with the card are found not in the object itself, but in the class PlayingCard.

[0] Section headings followed by an asterisk indicate optional material.



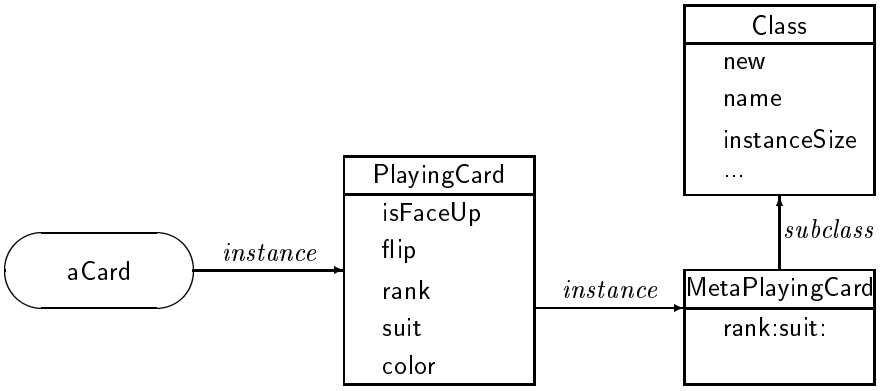
But in Smalltalk classes *are* objects. We explored this briefly in the previous chapter. Thus classes themselves respond to certain messages, such as the object creation message new.



Given this situation, let us now imagine that we want to create a method that can be used in the fashion of a constructor. That is, we want a method, let us call it rank:suit:, that can be given to a specific class object, say PlayingCard, and when executed it will both create a new instance and ensure it is properly initialized. Where in the picture just given can this method be placed? It cannot be part of the methods held by PlayingCard, as those are methods that are to be executed by *instances* of the class, and at the time of creation we do not yet have an instance. Nor can it be part of the methods held by Class, since those represent behavior common to all classes, and our initialization is something we want to do only for this one class.

The solution is to create a new "hidden" class, termed a metaclass. The object named PlayingCard is not actually an instance of Class, but is in reality an instance of MetaPlayingCard, which is formed from inheritance from Class.[1] Initialization specific behavior can then be placed in this new class.

[1] We are being slightly premature in presenting the discussion here, since inheritance will not be discussed in detail until Chapter 8. But the intuitive description of inheritance given in Chapter 1 is sufficient to understand the concept of metaclasses.



The behavior placed in the class MetaPlayingCard is understood by the object PlayingCard, and by no other object. This object is the only instance of the class.

Smalltalk browsers generally hide the existence of metaobjects from programmers, calling such methods by the term *class methods*, and acting as if they were associated with the same class as other methods. But behind the browser, class methods are simply ordinary methods associated with metaclasses.

Chapter Summary

In this chapter we have examined the syntax and techniques used on object creation and initialization for each of the different languages we are considering.

- We have examined the syntax used for message passing.
- We introduced the two major categories of programming languages, statically and dynamically typed languages. In a statically typed language types are associated with variables, while in a dynamically typed language a variable is simply a name, and types are associated with values.
- In many languages the receiver of a message can be accessed from within the body of the method used to respond to the message. The receiver is represented by a pseudo-variable. This variable can be named this, self, or current (depending upon the language being used).
- Automatic memory allociates the lifetime of an object with the procedure in which it is declared. Heap-based memory is explicitly allocated (in most languages using an operator named new) and is either explicitly deallocated or recovered by a garbage collection system.
- A constructor ties together the two tasks of memory allocation and initialization. This ensures that all objects which are allocated are properly initialized.
- A destructor is executed with a value is deleted.

- Finally, we have examined how metaclasses in Smalltalk address the problem of creation and initialization in that language.

Further Reading

The works cited at the end of the previous chapter should be consulted for more detailed information on any of the languages we are considering in this book.

Cohen [Cohen 1981] provides a good overview of garbage collection techniques. An interesting comparison between garbage collection and automatic memory allocation is given by Appel [Appel 1987]. Techniques for Reference counting in C++ are described in [Budd 1999].

Metaclasses in Smalltalk will be examined in detail later in Chapter 25.

Self Study Questions

1. In what ways is a message passing operation different from a procedure call?

2. What are the three parts of a message passing expression?

3. How does Smalltalk style keyword notation differ from Java or C++ style notation?

4. What is the difference between a statically typed language and a dynamically typed language?

5. Why are run-time errors of the form "receiver does not understand message" not common in statically typed languages? Why are they more common in dynamically typed languages?

6. What does the pseudo-variable this (or self in Smalltalk) refer to?

7. What is the difference between stack allocated and heap allocated memory?

8. What are the two general approaches to recovery of heap allocated memory?

9. What common programming errors does the use of a garbage collection system eliminate?

10. What two tasks are brought together by a constructor?

11. When is a destructor method executed?

12. What is a metaclass? What problem is solved through the use of metaclasses?

Exercises

1. Write a method copy for the class Card of Chapter 4. This method should return a new instance of the class Card with the suit and rank fields initialized to be the same as the receiver.

2. In a language that does not provide direct support for immutable instance variables, how might you design a software tool that would help to detect violations of access? (Hint: The programmer can provide directives in the form of comments that tell the tool which variables should be considered immutable.)

3. We have seen two styles for invoking methods. The approach used in C++ is similar to a conventional function call. The Smalltalk and Objective-C approaches separate arguments with keyword identifiers. Which do you think is more readable? Which is more descriptive? Which is more error-prone? Present short arguments to support your opinions.

4. How might you design a tool to detect the different types of memory allocation and free problems described in Section 5.5.1?

5. Andrew Appel [Appel 1987] argues that under certain circumstances heap-based memory allocation can be more efficient than stack-based memory allocation. Read this article and summarize the points of Appel's argument. Are the situations in which this is true likely to be encountered in practice?

6. Write a short (two- or three-paragraph) essay arguing for or against automatic memory-management (garbage-collection) systems.

Chapter 6

A Case Study: The Eight Queens Puzzle

This chapter presents the first of several case studies (or paradigms, in the original sense of the word) of programs written in an object-oriented style. The programs in this Chapter will be rather small so that we can present versions in several different langauges. Later case studies will be presented in only one language.

After first describing the problem, we will discuss how an object-oriented solution would differ from another type of solution. The chapter then concludes with a solution written in each language.

6.1 The Eight-Queens Puzzle

In the game of chess, the queen can attack any piece that lies on the same row, on the same column, or along a diagonal. The *eight-queens* is a classic logic puzzle. The task is to place eight queens on a chessboard in such a fashion that no queen can attack any other queen. A solution is shown in Figure 6.1, but this solution is not unique. The eight-queens puzzle is often used to illustrate problem-solving or backtracking techniques.

How would an object-oriented solution to the eight-queens puzzle differ from a solution written in a conventional imperative programming language? In a conventional solution, some sort of data structure would be used to maintain the positions of the pieces. A program would then solve the puzzle by systematically manipulating the values in these data structures, testing each new position to see whether it satisfied the property that no queen can attack any other.

We can provide an amusing but nevertheless illustrative metaphor for the difference between a conventional and an object-oriented solution. A conventional program is like a human being sitting above the board, and moving the chess pieces, which have no animate life of their own. In an object-oriented solution,

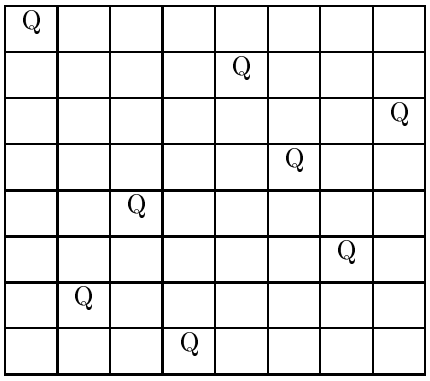


Figure 6.1: – One solution to the eight-queens puzzle.

on the other hand, we will empower the *pieces* to solve the problem themselves. That is, instead of a single monolithic entity controlling the outcome, we will distribute responsibility for finding the solution among many interacting agents. It is as if the chess pieces themselves are animate beings who interact with each other and take charge of finding their own solution.

Thus, the essence of our object-oriented solution will be to create objects that represent each of the queens, and to provide them with the abilities to discover the solution. With the computing-as-simulation view of Chapter 1, we are creating a model universe, defining behavior for the objects in this universe, then setting the universe in motion. When the activity of the universe stabilizes, the solution has been found.

6.1.1 Creating Objects That Find Their Own Solution

How might we define the behavior of a queen object so that a group of queens working together can find a solution on their own? The first observation is that, in any solution, no two queens can occupy the same column, and consequently no column can be empty. At the start we can therefore assign a specific column to each queen and reduce the problem to the simpler task of finding an appropriate row.

To find a solution it is clear that the queens will need to communicate with each other. Realizing this, we can make a second important observation that will greatly simplify our programming task–namely, each queen needs to know only about the queens to her immediate left. Thus, the data values maintained for each queen will consist of three values: a column value, which is *immutable*; a row value, which is altered in pursuit of a solution; and the neighboring queen to the immediate left.

Let us define an *acceptable solution for column n* to be a configuration of

columns 1 through n in which no queen can attack any other queen in those columns. Each queen will be charged with finding acceptable solutions between herself and her neighbors on her left. We will find a solution to the entire puzzle by asking the right most queen to find an acceptable solution. A CRC-card description of the class Queen, including the data managed by each instance (recall that this information is described on the back side of the card), is shown in Figure 6.2.

6.2 Using Generators

As with many similar problems, the solution to the eight-queens puzzle involves two interacting steps: *generating* possible partial solutions and *filtering out* solutions that fail to satisfy some later goal. This style of problem solving is sometimes known as the *generate and test* paradigm.

Let us consider the filter step first, as it is easier. For the system to test a potential solution it is sufficient for a queen to take a coordinate (row-column) pair and produce a Boolean value that indicates whether that queen, or any queen to her left, can attack the given location. A pseudo code algorithm that checks to see whether a queen can attack a specific position is given below. The procedure canAttack uses the fact that, for a diagonal motion, the differences in rows must be equal to the differences in columns.

```
function queen.canAttack(testRow , testColumn) -> boolean
    /* test for same row */
    if row = testRow then
        return true

    /* test diagonals */
    columnDifference := testColumn - column
    if (row + columnDifference = testRow) or
        (row - columnDifference = testRow)
            then return true

    /* we can't attack, see if neighbor can */
    return neighbor.canAttack(testRow, testColumn)
end
```

6.2.1 Initialization

We will divide the task of finding a solution into parts. The method initialize establishes the initial conditions necessary for a queen object, which in this case simply means setting the data values. This is usually followed immediately by a call on findSolution to discover a solution for the given column. Because such a

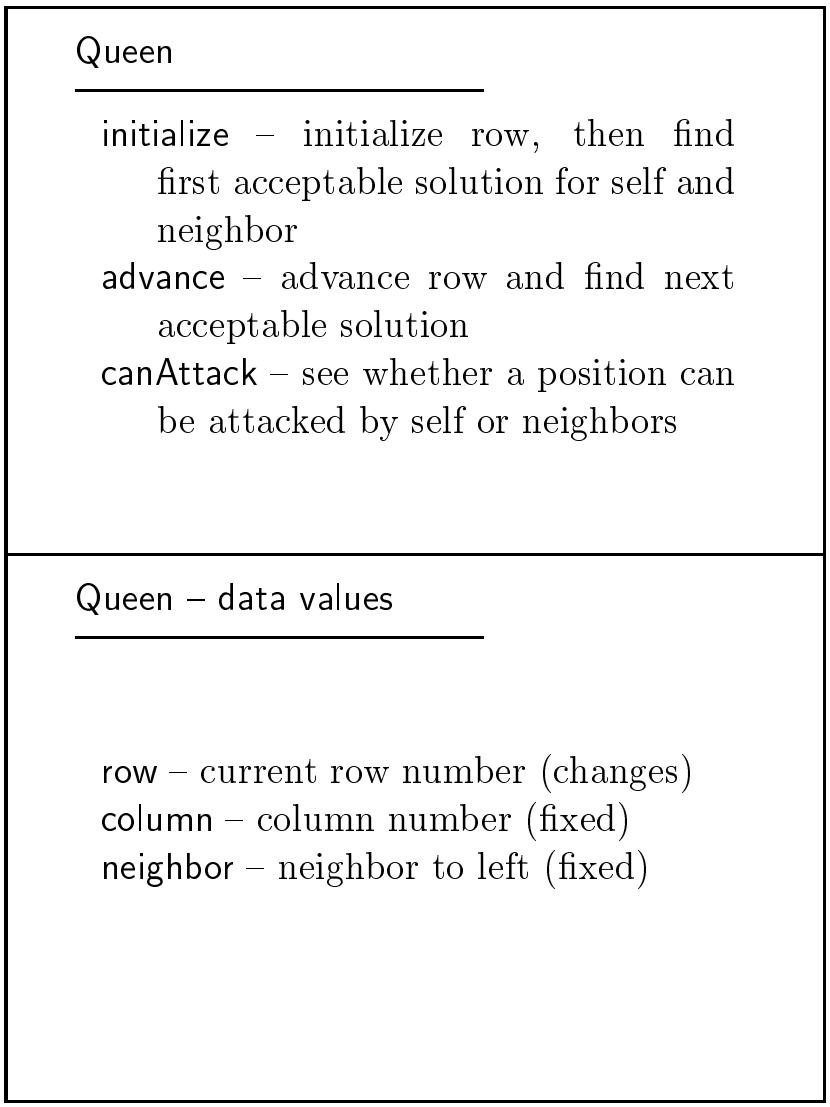


Figure 6.2: – Front and back sides of the Queen CRC card.

solution will often not be satisfactory to subsequent queens, the message advance is used to advance to the next solution.

A queen in column n is initialized by being given a column number, and the neighboring queen (the queen in column $n-1$). At this level of analysis, we will leave unspecified the actions of the leftmost queen, who has no neighbor. We will explore various alternative actions in the example problems we subsequently present. We will assume the neighbor queens (if any) have already been initialized, which includes their having found a mutually satisfactory solution. The queen in the current column simply places herself in row 1. A pseudo-code description of the algorithm is shown below.

```
function queen.initialize(col, neigh) -> boolean

        /* initialize our column and neighbor values */
    column := col
    neighbor := neigh

        /* start in row 1 */
    row := 1
    return findSolution;
end
```

6.2.2 Finding a Solution

To find a solution, a queen simply asks its neighbors if they can attack. If so, then the queen advances herself, if possible (returning failure if she cannot). When the neighbors indicate they cannot attack, a solution has been found.

```
function queen.findSolution -> boolean

        /* test positions */
    while neighbor.canAttack (row, column) do
        if not self.advance then
            return false

        /* found a solution */
    return true
end
```

As we noted in Chapter 5, the pseudo-variable self denotes the receiver for the current message. In this case we want the queen who is being asked to find a solution to pass the message advance to herself.

6.2.3 Advancing to the Next Position

The procedure advance divides into two cases. If we are not at the end, the queen simply advances the row value by 1. Otherwise, she has tried all positions and not found a solution, so nothing remains but to ask her neighbor for a new solution and start again from row 1.

```
function queen.advance -> boolean

        /* try next row */
    if row < 8 then begin
        row := row + 1
        return self.findSolution
    end

        /* cannot go further */
        /* move neighbor to next solution */
    if not neighbor.advance then
        return false

        /* start again in row 1 */
    row := 1
    return self.findSolution
end
```

The one remaining task is to print out the solution. This is most easily accomplished by a simple method, print that is rippled down the neighbors.

```
procedure print
    neighbor.print
    write row, column
end
```

6.3 The Eight-Queens Puzzle in Several Languages

In this section we present solutions to the eight-queens puzzle in several of the programming languages we are considering. Examine each variation, and compare how the basic features provided by the language make subtle changes to the final solution. In particular, examine the solutions written in Smalltalk and Objective-C, which use a special class for a sentinel value, and contrast this with the solutions given in Object Pascal, C++, or Java, all of which use a null pointer for the leftmost queen and thus must constantly test the value of pointer variables.

6.3.1 The Eight-Queens Puzzle in Object Pascal

The class definition for the eight-queens puzzle in Apple Object Pascal is shown below. A subtle but important point is that this definition is recursive; objects of type Queen maintain a data field that is itself of type Queen. This is sufficient to indicate that declaration and storage allocation are not necessarily linked; if they were, an infinite amount of storage would be required to hold any Queen value. We will contrast this with the situation in C++ when we discuss that language.

```
type
    Queen = object
            (* data fields *)
        row :        integer;
        column :    integer;
        neighbor :    Queen;

            (* initialization *)
        procedure initialize (col : integer; ngh : Queen);

            (* operations *)
        function     canAttack
                    (testRow, testColumn : integer) : boolean;
        function     findSolution : boolean;
        function     advance : boolean;
        procedure    print;
    end;
```

The class definition for the Delphi language differs only slightly, as shown below. The Borland language allows the class declaration to be broken into public and private sections, and it includes a constructor function, which we will use in place of the initialize routine.

```
TQueen = class (TObject)
public
    constructor Create (initialColumn : integer; nbr : TQueen);
    function findSolution : boolean;
    function advance : boolean;
    procedure print;

private
    function canAttack (testRow, testColumn : integer) : boolean;
    row : integer;
    column : integer;
    neighbor : TQueen;
```

```
end;
```

The pseudo-code presented in the earlier sections is reasonably close to the Pascal solution, with two major differences. The first is the lack of a return statement in Pascal, and the second is the necessity to first test whether a queen has a neighbor before passing a message to that neighbor. The functions findSolution and advance, shown below, illustrate these differences. (Note that Delphi Pascal differs from standard Pascal in permitting short-circuit interpretation of the and and or directives, in the fashion of C++. Thus, the code for the Delphi language could in a single expression combine the test for neighbor being non-null and the passing of a message to the neighbor).

```
function Queen.findSolution : boolean;
var
    done : boolean;
begin
    done := false;
    findsolution := true;

        (* test positions *)
    if neighbor <> nil then
        while not done and neighbor.canAttack(row, column) do
            if not self.advance then begin
                findSolution := false;
                done := true;
            end;
end;

function Queen.advance : boolean;
begin
    advance := false;
        (* try next row *)
    if row < 8 then begin
        row := row + 1;
        advance := self.findSolution;
    end
    else begin
            (* cannot go further *)
            (* move neighbor to next solution *)
        if neighbor <> nil then
            if not neighbor.advance then
                advance := false
            else begin
                row := 1;
                advance := self.findSolution;
```

```
            end;
    end;
end;
```

The main program allocates space for each of the eight queens and initializes the queens with their column number and neighbor value. Since during initialization the first solution will be discovered, it is only necessary for the queens to print their solution. The code to do this in Apple Object Pascal is shown below. Here, neighbor and i are temporary variables used during initialization and lastQueen is the most recently created queen.

```
begin
    neighbor := nil;
    for i := 1 to 8 do begin
            (* create and initialize new queen *)
        new (lastQueen);
        lastQueen.initial (i, neighbor);
        if not lastQueen.findSolution then
            writeln('no solution');
            (* newest queen is next queen neighbor *)
        neighbor := lastQueen;
    end;

        (* print the solution *)
    lastQueen.print;

    end;
end.
```

By providing explicit constructors that combine new object creation and initialization, the Delphi language allows us to eliminate one of the temporary variables. The main program for the Delphi language is as follows:

```
begin
    lastQueen := nil;
    for i := 1 to 8 do begin
            // create and initialize new queen
        lastQueen := Queen.create(i, lastQueen);
        lastQueen.findSolution;
        end;

        // print the solution
    lastQueen.print;
end;
```

6.3.2 The Eight-Queens Puzzle in C++

The most important difference between the pseudo-code description of the algorithm presented earlier and the eight-queens puzzle as actually coded in C++ is the explicit use of pointer values. The class description for the class Queen is shown below. Each instance maintains, as part of its data area, a pointer to another queen value. Note that, unlike the Object Pascal solution, in C++ this value must be declared explicitly as a pointer rather than an object value.

```
class Queen {
public:
        // constructor
    Queen (int, Queen *);

        // find and print solutions
    bool findSolution();
    bool advance();
    void print();

private:
        // data fields
    int row;
    const int column;
    const Queen * neighbor;

        // internal method
    bool canAttack (int, int);
};
```

As in the Delphi Pascal solution, we have subsumed the behavior of the method initialize in the constructor. We will describe this shortly.

There are three data fields. The integer data field column has been marked as const. This identifies the field as an immutable value, which cannot change during execution. The third data field is a pointer value, which either contains a null value (that is, points at nothing) or points to another queen.

Since initialization is performed by the constructor, the main program can simply create the eight queen objects, and then print their solution. The variable lastQueen will point to the most recent queen created. This value is initially a null pointer–it points to nothing. A loop then creates the eight values, initializing each with a column value and the previous queen value. When the loop completes, the leftmost queen holds a null value for its neighbor field while every other queen points to its neighbor, and the value lastQueen points to the rightmost queen.

```
void main() {
    Queen * lastQueen = 0;

    for (int i = 1; i <= 8; i++) {
        lastQueen = new Queen(i, lastQueen);
        if (! lastQueen->findSolution())
            cout << "no solution\n";
        }

    lastQueen->print();
}
```

We will describe only those methods that illustrate important points. The complete solution can be examined in Appendix A.

The constructor method must use the initialization clauses on the heading to initialize the constant value column, as it is not permitted to use an assignment operator to initialize instance fields that have been declared const. An initialization clause is also used to assign the value neighbor, although we have not declared this field as constant.

```
Queen::Queen(int col, Queen * ngh) : column(col), neighbor(ngh)
{
    row = 1;
}
```

Because the value of the neighbor variable can be either a queen or a null value, a test must be performed before any messages are sent to the neighbor. This is illustrated in the method findSolution. The use of short-circuit evaluation in the logical connectives and the ability to return from within a procedure simplify the code in comparison to the Object Pascal version, which is otherwise very similar.

```
bool Queen::findSolution()
{
    while (neighbor && neighbor->canAttack(row, column))
            if (! advance())
                return false;
    return true;
}
```

The advance method must similarly test to make certain there is a neighbor before trying to advance the neighbor to a new solution. When passing a message to oneself, as in the recursive message findSolution, it is not necessary to specify a receiver.

```
bool Queen::advance()
```

```
{
    if (row < 8) {
        row++;
        return findSolution();
        }

    if (neighbor && ! neighbor->advance())
        return false;

    row = 1;
    return findSolution();
}
```

6.3.3 The Eight-Queens Puzzle in Java

The solution in Java is in many respects similar to the C++ solution. However, in Java the bodies of the methods are written directly in place, and public or private designations are placed on the class definitions themselves. The following is the class description for the class Queen, with some of the methods omitted.

```
class Queen {
        // data fields
    private int row;
    private int column;
    private Queen neighbor;

        // constructor
    Queen (int c, Queen n) {
            // initialize data fields
        row = 1;
        column = c;
        neighbor = n;
        }

    public boolean findSolution() {
        while (neighbor != null &&
                neighbor.canAttack(row, column))
            if (! advance())
                return false;
        return true;
        }

    public boolean advance() { ...  }
```

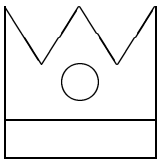
```
    private boolean canAttack(int testRow, int testColumn) { ...  }

    public void paint (Graphics g) { ...  }
}
```

Unlike in C++, in Java the link to the next queen is simply declared as an object of type Queen and not as a pointer to a queen. Before a message is sent to the neighbor instance variable, an explicit test is performed to see if the value is null.

Since Java provides a rich set of graphics primitives, this solution will differ from the others in actually drawing the final solution as a board. The method paint will draw an image of the queen, then print the neighbor images.

```
class Queen {
    ...
    public void paint (Graphics g) {
            // x, y is upper left corner
            // 10 and 40 give slight margins to sides
        int x = (row - 1) * 50 + 10;
        int y = (column - 1) * 50 + 40;
        g.drawLine(x+5, y+45, x+45, y+45);
        g.drawLine(x+5, y+45, x+5, y+5);
        g.drawLine(x+45, y+45, x+45, y+5);
        g.drawLine(x+5, y+35, x+45, y+35);
        g.drawLine(x+5, y+5, x+15, y+20);
        g.drawLine(x+15, y+20, x+25, y+5);
        g.drawLine(x+25, y+5, x+35, y+20);
        g.drawLine(x+35, y+20, x+45, y+5);
        g.drawOval(x+20, y+20, 10, 10);
            // then draw neighbor
        if (neighbor != null)
            neighbor.paint(g);
        }
}
```

The graphics routines draw a small crown, which looks like this:



Java does not have global variables nor functions that are not member functions. As we will describe in more detail in Chapter 22, a program is created by the defining of a subclass of the system class JFrame, and then the overriding of certain methods. Notably, the constructor is used to provide initialization for the application, while the method paint is used to redraw the screen. Mouse events and window events are handled by creating *listener objects* that will execute when their associated event occurs. We will describe listeners in much greater detail in later sections. We name the application class QueenSolver and define it as follows:

```
public class QueenSolver extends JFrame {

    public static void main(String [ ] args) {
        QueenSolver world = new QueenSolver();
        world.show();
    }

    private Queen lastQueen = null;

    public QueenSolver() {
        setTitle("8 queens");
        setSize(600, 500);
        for (int i = 1; i <= 8; i++) {
            lastQueen = new Queen(i, lastQueen);
            lastQueen.findSolution();
            }
        addMouseListener(new MouseKeeper());
        addWindowListener(new CloseQuit());
        }

    public void paint(Graphics g) {
        super.paint(g);
            // draw board
        for (int i = 0; i <= 8; i++) {
            g.drawLine(50 * i + 10, 40, 50*i + 10, 440);
            g.drawLine(10, 50 * i + 40, 410, 50*i + 40);
        }
        g.drawString("Click Mouse for Next Solution", 20, 470);
            // draw queens
        lastQueen.paint(g);
        }

    private class CloseQuit extends WindowAdapter {
        public void windowClosing (WindowEvent e) {
            System.exit(0);
```

```
        }
    }

    private class MouseKeeper extends MouseAdapter {
        public void mousePressed (MouseEvent e) {
            lastQueen.advance();
            repaint();
        }
    }
}
```

Note that the application class must be declared as public, because it must be accessible to the main program.

6.3.4 The Eight-Queens Puzzle in Objective-C

The interface description for our class Queen is as follows:

```
@interface Queen : Object
{    /* data fields */
    int row;
    int column;
    id neighbor;
}

    /* methods */
- (void) initialize: (int) c neighbor: ngh;
- (int)  advance;
- (void) print;
- (int)  canAttack: (int) testRow column: (int) testColumn;
- (int)  findSolution;

@end
```

Each queen will maintain three data fields: a row value, a column, and the neighbor queen. The last is declared with the data type id. This declaration indicates that the value being held by the variable is an object type, although not necessarily a queen.

In fact, we can use this typeless nature of variables in Objective-C to our advantage. We will employ a technique that is not possible, or at least not as easy, in a more strongly typed language such as C++ or Object Pascal. Recall that the leftmost queen does not have any neighbor. In the C++ solution, this was indicated by the null, or empty value, in the neighbor pointer variable in the leftmost queen. In the current solution, we will instead create a new type

of class, a *sentinel value*. The leftmost queen will point to this sentinel value, thereby ensuring that every queen has a valid neighbor.

Sentinel values are frequently used as endmarkers and are found in algorithms that manipulate linked lists, such as our linked list of queen values. The difference between an object-oriented sentinel and a more conventional value is that an object-oriented sentinel value can be *active*–it can have *behavior*–which means it can respond to requests.

What behaviors should our sentinel value exhibit? Recall that the neighbor links in our algorithm were used for two purposes. The first was to ensure that a given position could not be attacked; our sentinel value should always respond negatively to such requests, since it cannot attack any position. The second use of the neighbor links was in a recursive call to print the solution. In this case our sentinel value should simply return, since it does not have any information concerning the solution.

Putting these together yields the following implementation for our sentinel queen.

```
@implementation SentinelQueen : Object
- (int) advance
{
    /* do nothing */
    return 0;
}

- (int) findSolution
{
    /* do nothing */
    return 1;
}

- (void) print
{
    /* do nothing */
}

- (int) canAttack: (int) testRow column: (int) testColumn;
{
    /* cannot attack */
    return 0;
}
@end
```

In the full solution there is an implementation section for SentinelQueen, but no interface section. This omission is legal, although the compiler will provide a warning since it is somewhat unusual.

The use of the sentinel allows the methods in class Queen to simply pass messages to their neighbor without first determining whether or not she is the leftmost queen. The method for canAttack, for example, illustrates this use:

```
- (int) canAttack: (int) testRow column: (int) testColumn
{    int columnDifference;

    /* can attack same row */
    if (row == testRow)
        return 1;

    columnDifference = testColumn - column;
    if ((row + columnDifference == testRow) ||
        (row - columnDifference == testRow))
            return 1;

    return [ neighbor canAttack:testRow column: testColumn ];
}
```

Within a method, a message sent to the receiver is denoted by a message sent to the pseudo-variable self.

```
- (void) initialize: (int) c neighbor: ngh
{
    /* set the constant fields */
    column = c;
    neighbor = ngh;
    row = 1;
}

- (int) findSolution
{
    /* loop until we find a solution */
    while ([neighbor canAttack: row and: column ])
        if (! [self advance])
            return 0; /* return false */
    return 1; /* return true */
}
```

Other methods are similar, and are not described here.

6.3.5 The Eight-Queens Puzzle in Smalltalk

The solution to the eight-queens puzzle in Smalltalk is in most respects very similar to the solution given in Objective-C. Like Objective-C, Smalltalk handles

the fact that the leftmost queen does not have a neighbor by defining a special *sentinel* class. The sole purpose of this class is to provide a target for the messages sent by the leftmost queen.

The sentinel value is the sole instance of the class SentinelQueen, a subclass of class Object, which implements the following three methods:

```
advance
        " sentinels do not attack "
    ↑ false


canAttack: row column: column
        " sentinels cannot attack "
    ↑ false

result
        " return empty list as result "
    ↑ List new
```

One difference between the Objective-C and Smalltalk versions is that the Smalltalk code returns the result as a list of values rather than printing it on the output. The techniques for printing output are rather tricky in Smalltalk and vary from implementation to implementation. By returning a list we can isolate these differences in the calling method.

The class Queen is a subclass of class Object. Instances of class Queen maintain three instance variables: a row value, a column value, and a neighbor. Initialization is performed by the method setColumn:neighbor:

```
setColumn: aNumber neighbor: aQueen
        " initialize the data fields "
    column := aNumber.
    neighbor := aQueen.
    row := 1.
```

The canAttack method differs from the Objective-C counterpart only in syntax:

```
canAttack: testRow column: testColumn | columnDifference |
    columnDifference := testColumn - column.
    (((row = testRow) or:
        [ row + columnDifference = testRow]) or:
        [ row - columnDifference = testRow])
            ifTrue: [ ↑ true ].
    ↑ neighbor canAttack: testRow column: testColumn
```

Rather than testing for the negation of a condition, Smalltalk provides an explicit ifFalse statement, which is used in the method advance:

```
advance
        " first try next row "
    (row < 8)
        ifTrue: [ row := row + 1. ↑ self findSolution ].
        " cannot go further, move neighbor "
    (neighbor advance) ifFalse: [ ↑ false ].
        " begin again in row 1 "
    row := 1.
    ↑ self findSolution
```

The while loop in Smalltalk must use a block as the condition test, as in the following:

```
findSolution
    [ neighbor canAttack: row column: column ]
        whileTrue: [ self advance ifFalse: [ ↑ false ] ].
    ↑ true
```

A recursive method is used to obtain the list of answer positions. Recall that an empty list is created by the sentinel value in response to the message result.

```
result
    ↑ neighbor result; addLast: row
```

A solution can be found by invocation of the following method, which is not part of class Queen but is instead attached to some other class, such as Object.

```
solvePuzzle | lastQueen |
    lastQueen := SentinelQueen new.
    1 to: 8 do: [:i | lastQueen := (Queen new)
        setColumn: i neighbor: lastQueen.
        lastQueen findSolution ].
    ↑ lastQueen result
```

6.3.6 The Eight-Queens Puzzle in Ruby

Ruby is a recent scripting language, similar in spirit to Python or Perl. There are only functions in Ruby, every method returns a value, which is simply the value of the last statement in the body of the method. A feel for the syntax for

Ruby can be found by the definition of the sentinel queen, which can be written as follows:

```
class NullQueen

  def canAttack(row, column)
    false
  end

  def first?
    true
  end

  def next?
    false
  end

  def getState
    Array.new
  end

end
```

The class Queen handles all but the last case. In Ruby instance variables must begin with an at-sign. Thus the initialization method is written as follows:

```
class Queen

  def initialColumn(column, neighbour)
    @column = column
    @neighbour = neighbour
    nil
  end
  ...

end
```

Conditional statements are written in a curious form where the expression is given first, followed by the if keyword. This is illustrated by the method canAttack:

```
  def canAttack(row, column)
    return true if row == @row

    cd = (column - @column).abs
```

```
    rd = (row - @row).abs
    return true if cd == rd

    @neighbour.canAttack(row, column)
  end
```

The remainder of the Ruby solution can be found in the appendix.

Chapter Summary

In this first case study we have examined a classic puzzle, how to place eight queens on a chessboard in such a way that no queen can attack any of the others. While the problem is moderately intriguing, our interest is not so much in the problem itself, but in the way the solution to the problem has been structured. We have addressed the problem by making the queens into independent agents, who then work among themselves to discover a solution.

Further Reading

A solution to the eight-queens puzzle constructed without the use of a sentinel value was described in my earlier book on Smalltalk [Budd 1987].

The eight queens puzzle is found in many computing texts. See [Griswold 1983, Budd 1987, Berztiss 1990], for some representative examples.

For further information on the general technique termed generate and test, see [Hanson 1981], or [Berztiss 1990].

The solution in Ruby was written by Mike Stok. Further information on Ruby can be found in [Thomas 2001].

Self Study Questions

1. What is the eight queens puzzle?

2. In what way is the object-oriented solution presented here different from a conventional solution?

3. What is the generate and test approach to finding a solution in a space of various alternative possibilities?

4. What is a sentinel? (The term is introduced in the solution presented in Objective-C).

Exercises

1. Modify any one of the programs to produce all possible solutions rather than just one. How many possible solutions are there for the eight-queens puzzle? How many of these are rotations of other solutions? How might you filter out rotations?

2. Can you explain why the sentinel class in the Objective-C and Smalltalk versions of the eight-queens puzzle do not need to provide an implementation for the method findSolution, despite the fact that this message is passed to the neighbor value in the method advance?

3. Suppose we generalize the eight-queens problem to the N-queens problem, where the task is to place N queens on an N by N chessboard. How must the programs be changed?

   It is clear that there are values for N for which no solution exists (consider N=2 or N=3, for example). What happens when your program is executed for these values? How might you produce more meaningful output?

4. Using whatever graphics facilities your system has, alter one of the programs to display dynamically on a chessboard the positions of each queen as the program advances. What portions of the program need to know about the display?

Chapter 7

A Case Study: A Billiards Game

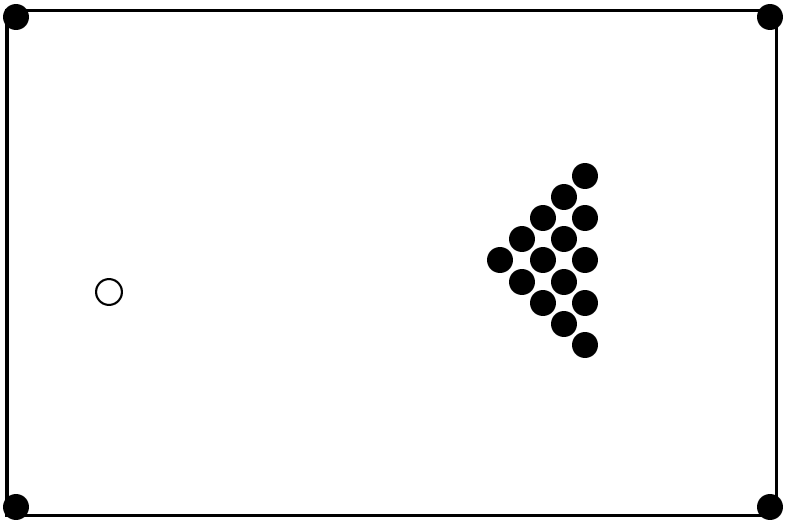
In our second case study, we will develop a simple simulation of a billiard table.[1] The program is written in Delphi Pascal.[2] As with the eight-queens program, the design of this program will stress the creation of autonomous interacting agents working together to produce the desired outcome.

7.1 The Elements of Billiards

The billiard table as the user sees it consists of a window containing a rectangle with holes (pockets) in the corners, 15 colored balls, and 1 white cue ball. By clicking the mouse the user simulates striking the cue ball, imparting a certain amount of energy to it. The direction of motion for the cue ball will be opposite to that of the mouse position in relation to the cue. Once a ball has energy it will start to move, reflecting off of walls, falling into holes, and potentially striking other balls. When a ball strikes another ball some of the energy of the first is given to the second, while the direction of movement of the two balls is changed by the collision.

[1] The game implemented by the program described in this chapter does not correspond to any actual game. It is not pool, it is not billiards, it is simply balls moving around a table consisting of walls and holes.

[2] Discussion of Delphi Pascal is complicated by the fact that graphical user interface elements of Delphi programs are constructed visually, using the integrated development environment. This style of design will be familiar to users of Visual Basic. However, the user interface aspects are not relevant to our purposes, which is the investigation of Delphi as an object-oriented programming language. The references at the end of the chapter provide pointers to further information regarding these other aspects of Delphi.



7.2 Graphical Objects

The heart of the simulation are three linked lists of *graphical objects*, which comprise the walls, holes, and balls. Each graphical object will include a link field and a field indicating the region of the screen occupied by the object.[3]

A simplifying assumption we have made is that all graphical objects occupy rectangular regions. This is, of course, quite untrue for a round object such as a ball. A more realistic alternative would have been to write a procedure that determined whether two balls have intersected based on the geometry of the ball rather than on the intersection of their regions. Once again, the complexity of the procedure would only have detracted from the issues we wish to address in our case study.

The primary objective in this case study is the way in which responsibility for behavior has been vested in the objects themselves. Every graphical object knows not only how to draw itself but how to move and how to interact with the other objects in the simulation.

[3] There are clear conflicts in ordering in the presentation of this case study. On the one hand, it is important for the reader to see examples of object-oriented principles as soon as possible; thus, placing this particular case study early in the book is desirable. On the other hand, this program, like almost all object-oriented programs, would benefit from more advanced techniques, which we will not discuss until later. In particular, the graphical objects might be better described by an inheritance hierarchy, such as we will describe in Chapter 8. Similarly, it is generally considered poor programming practice for the objects being maintained on a linked list to hold the link fields as part of their data area; a better design would separate the container from the elements in the list. Solving this problem is non-trivial and introduces complications not particularly relevant to the points addressed here. We will discuss container classes in Chapter 19.

7.2.1 The Wall Graphical Object

The first of our three graphical objects is a wall. It is defined by the following class description:

```
TWall = class(TObject)
public
    constructor create
        (ix, iy, iw, ih : Integer; cf : Real; ilink : TWall);
    procedure draw (canvas : TCanvas);
    function hasIntersected(aBall : Tball) : Boolean;
    procedure hitBy (aBall : TBall);
private
    x, y : Integer;
    height, width : Integer;
    convertFactor : Real;
    link : TWall;
end;
```

The x and y fields represent the upper left corner of the wall, while the height and width fields maintain the size. The link field maintains a linked list of wall objects. The constructor simply defines the region of the wall and sets the convert factor:

```
constructor TWall.create
    (ix, iy, iw, ih : Integer; cf : Real; ilink : Twall);
begin
    x := ix;
    y := iy;
    height := ih;
    width := iw;
    convertFactor := cf;
    link := ilink;
end;
```

A wall can be drawn simply by printing a solid rectangle. A graphics library routine performs this task:

```
procedure TWall.draw(canvas: TCanvas);
begin
        with canvas do begin
                Brush.Style := bsSolid;
                Brush.Color := clBlack;
                fillRect(Rect(x, y, x + width, y + height));
        end;
```

```
end;
```

The most interesting behavior of a wall occurs when it has been struck by a ball. The direction of the ball is modified by use of the convert factor for the wall. (Convert factors are either zero or pi, depending upon whether the wall is horizontal or vertical). The ball subsequently moves off in a new direction.

```
procedure TWall.hitBy (aBall : TBall);
begin
        { bounce the ball off the wall }
    aBall.direction := convertFactor - aBall.direction;
end;
```

7.2.2 The Hole Graphical Object

A hole is defined by the following class description:

```
THole = class(TObject)
public
    constructor create (ix, iy : Integer; ilink : THole);
    procedure draw (canvas : TCanvas);
    function hasIntersected(aBall : TBall) : Boolean;
    procedure hitBy (aBall : TBall);
private
    x, y : Integer;
    link : THole;
end;
```

As with walls, the initialization and drawing of holes is largely a matter of invoking the correct library routines:

```
constructor THole.create(ix, iy : Integer; ilink : THole);
begin
    x := ix;
    y := iy;
    link := ilink;
end;

procedure THole.draw(canvas : TCanvas);
begin
        with canvas do begin
                Brush.Style := bsSolid;
                Brush.Color := clBlack;
                Ellipse(x-5, y-5, x+5, y+5);
        end;
```

```
end;
```

Of more interest is what happens when a hole is struck by a ball. There are two cases. If the ball happens to be the cue ball (which is identified with a global variable, CueBall), it is placed back into play at a fixed location. Otherwise, all the energy is drained from the ball and it is moved off the table to a special display area.

```
procedure THole.hitBy (aBall : TBall);
begin
        { drain enery from ball }
    aBall.energy := 0.0;

        { move ball }
    if aBall = CueBall then
        aBall.setCenter(50, 100)
    else begin
        saveRack := saveRack + 1;
        aBall.setCenter (10 + saveRack * 15, 250);
    end;
end;
```

7.2.3 The Ball Graphical Object

Our final graphical object is the ball, defined by the following class description:

```
TBall = class(TObject)
public
    constructor create (ix, iy : Integer; iLink : TBall);
    procedure draw (canvas : TCanvas);
    function hasIntersected(aBall : Tball) : Boolean;
    procedure hitBy (aBall : TBall);
    procedure update;
    procedure setCenter (nx, ny : Integer);
    procedure setDirection (nd : Real);
private
        x, y : Integer;
        direction : Real;
        energy : Real;
        link : TBall;
end;
```

In addition to the link and rectangle regions common to the other objects, a ball maintains two new data fields; a direction, measured in radians, and an energy, which is an arbitrary real value. Like a hole, a ball is initialized by

arguments that specify the center of the ball. Initially a ball has no energy and a direction of zero.

```
constructor TBall.create(ix, iy : Integer; iLink : TBall);
begin
        setCenter(ix, iy);
        setDirection(0.0);
        energy := 0.0;
        link := iLink;
end;

procedure TBall.setCenter(nx, ny : Integer);
begin
        x := nx;
        y := ny;
end;

procedure TBall.setDirection(nd : Real);
begin
        direction := nd;
end;
```

A ball is drawn either as a frame or as a solid circle, depending upon whether or not it represents the cue ball.

```
procedure TBall.draw(canvas : TCanvas);
begin
        with canvas do begin
                Brush.Style := bsSolid;
                if (self = cueBall) then
                        Brush.Color := clWhite
                else
                        Brush.Color := clBlack;
                Ellipse(x-5, y-5, x+5, y+5);
        end;
end;
```

The method update is used to update the position of the ball. If the ball has a nontrivial amount of energy, it moves slightly, then checks to see if it has hit another object. A global variable named ballMoved is set true if any ball on the table has moved. If the ball has hit another object, it notifies the second object that it has been struck. This notification process is divided into three steps, corresponding to hitting holes, walls, and other balls. Inheritance, which we will study in Chapter 8, will provide a means by which these three tests can be combined into a single loop.

```
procedure TBall.update;
var
    hptr : THole;
    wptr : TWall;
    bptr : TBall;
    dx, dy : integer;
begin
    if energy > 0.5 then begin
        ballMoved := true;
            { decrease energy }
        energy := energy - 0.05;
            { move ball }
        dx := trunc(5.0 * cos(direction));
        dy := trunc(5.0 * sin(direction));
                x := x + dx;
                y := y + dy;

            { see if we hit a hole }
        hptr := listOfHoles;
        while (hptr <> nil) do
            if hptr.hasIntersected(self) then begin
                hptr.hitBy(self);
                hptr := nil;
            end
            else
                hptr := hptr.link;

            { see if we hit a wall }
        wptr := listOfWalls;
        while (wptr <> nil) do
            if wptr.hasIntersected(self) then begin
                wptr.hitBy(self);
                wptr := nil;
            end
            else
                wptr := wptr.link;

            { see if we hit a ball }
        bptr := listOfBalls;
        while (bptr <> nil) do
            if (bptr <> self) and bptr.hasIntersected(self) then begin
                bptr.hitBy(self);
                bptr := nil;
            end
```

```
            else
                bptr := bptr.link;
    end;
end;
```

When one ball strikes another ball, the energy of the first one is split and half is given to the second one. The angles of both are also changed. (The physics is not exactly correct, but the results look reasonably realistic.)

```
procedure TBall.hitBy (aBall : TBall);
var
    da : real;
begin
        { cut the energy of the hitting ball in half }
    aBall.energy := aBall.energy / 2.0;

        { and add it to our own }
    energy := energy + aBall.energy;

        { set our new direction }
    direction := hitAngle(self.x - aBall.x, self.y - aBall.y);

        { and set the hitting balls direction }
    da := aBall.direction - direction;
    aBall.direction := aBall.direction + da;

        { continue our update }
    update;
end;

function hitAngle (dx, dy : real) : real;
    const
        PI = 3.14159;
    var
        na : real;
begin
    if (abs(dx) < 0.05) then
        na := PI / 2
    else
        na := arctan (abs(dy / dx));
    if (dx < 0) then
        na := PI - na;
    if (dy < 0) then
        na := - na;
    hitAngle := na;
```

```
end;
```

7.3 The Main Program

The previous section described the static characteristics of the program. The dynamic characteristics are set in motion when a mouse press occurs, at which time the following function is invoked:

```
procedure TfrmGraphics.DoClick (Sender: TObject;
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
    bptr : TBall;
begin
    cueBall.energy := 20.0;
    cueBall.setDirection(hitAngle(cueBall.x - x, cueBall.y - y));
        { then loop as long as called for }
    ballMoved := true;
    while ballMoved do begin
        ballMoved := false;
        bptr := listOfBalls;
        while bptr <> nil do begin
            bptr.update;
            bptr := bptr.link;
        end;
    end;
end;
```

The remainder of the program is relatively straight forward and will not be presented here. The complete source is given in Appendix B. The majority of the code is concerned with the initialization of the new objects and with the event loop that waits for the user to perform an action. The programmer uses the Delphi development environment to match events, such as mouse presses, with procesures, such as DoClick.

To stress the point we made at the beginning of this chapter, the most important feature of this case study is the fashion in which control has been decentralized and the objects themselves have been given the power to control and direct the flow of execution. When a mouse press occurs, all that happens is that the cue ball is provided with a certain amount of energy. Thereafter, the interaction of the balls drives the simulation.

7.4 Using Inheritance

In Chapter 1 we informally introduced inheritance, and in Chapter 8 we will discuss how inheritance works in each of the languages we are considering. In

this section we will describe how inheritance can be used to simplify the billiards simulation, foreshadowing the discussion we will present in the next chapter. The reader may wish to return to this section after reading the general treatment of inheritance in the next chapter.

We have, in fact, been using inheritance throughout our development of the classes for our application. All of our classes inherit from the system class TObject. But as we did not use any behavior from this class, the issue was not very important. Now we will create parent classes that do embody useful behavior.

The first step in using inheritance in our billiards simulation is to define a general class for "graphical objects." This class includes all three items: balls, walls, and holes. The parent class is defined as follows:

```
type
  TBall = class; (* forward declaration *)

  TGraphicalObject = class(TObject)
  public
        constructor Create(ix, iy : Integer; il : TGraphicalObject);
        procedure draw (canvas : TCanvas); virtual; abstract;
        function hasIntersected (aBall : TBall): Boolean; virtual; abstract;
        procedure hitBy (aBall : TBall); virtual; abstract;
        procedure update; virtual;
  private
        x, y : Integer;
        link : TGraphicalObject;
  end;
```

Note the forward declaration for the class TBall. This allows the class TGraphicalObject to declare arguments of type TBall, even though the class definition has not yet been seen.

Every graphical object has a location and a link. The constructor sets these values. The methods draw, hasIntersected, and hitBy are declared as virtual and abstract. This means they are not defined in the parent class, but must be redefined in the child classes. The method update is declared as virtual, but not abstract. In the parent class it is defined to do nothing. This behavior will be overridden by class TBall, but not by the other two.

The classes Ball, Wall, and Hole are then declared as subclasses of the general class GraphicalObject and need not repeat the declarations for data areas or functions, unless they are being overridden.

```
THole = class(TGraphicalObject)
public
    constructor create
        (ix, iy : Integer; ilink : TGraphicalObject); overload;
    procedure draw (canvas : TCanvas);  override;
```

```
    function hasIntersected(aBall : TBall) : Boolean;  override;
    procedure hitBy (aBall : TBall); override;
end;
```

Compare this declaration to the one given earlier, and note how we have now eliminated the declaration for the data fields, since they have been moved to the parent class.

Constructors for the child classes must explicitly invoke the constructors for their parent classes, as in the following constructor for class TBall:

```
constructor TBall.Create (ix, iy : Integer; iLink : TGraphicalObject);
begin
    inherited Create(ix, iy, iLink);
    setDirection(0.0);
    energy := 0.0;
end;
```

By making CueBall a subclass of Ball, we can eliminate the conditional statement in the routine that draws the ball's image.

```
TCueBall = class(TBall)
public
        procedure draw (canvas : TCanvas); override;
end;

procedure TBall.draw(canvas : TCanvas);
begin
        with canvas do begin
                Brush.Style := bsSolid;
                Brush.Color := clBlack;
                Ellipse(x-5, y-5, x+5, y+5);
        end;
end;

procedure TCueBall.draw (canvas : TCanvas);
begin
        with canvas do begin
                Brush.Style := bsSolid;
                Brush.Color := clWhite;
                Ellipse(x-5, y-5, x+5, y+5);
        end;
end;
```

The greatest simplification comes from the fact that it is now possible to keep all graphical objects on a single linked list. Thus, the routine that draws the entire screen, for example, can be written as follows:

```
procedure TfrmGraphics.DrawExample(Sender: TObject);
var
    gptr : TGraphicalObject;
begin
    with imgGraph.Canvas do begin
        Brush.Color := clWhite;
        Brush.Style := bsSolid;
        FillRect(Rect(0, 0, 700, 700));
    end;
    gptr := listOfObjects;
    while (gptr <> nil) do begin
        gptr.draw(imgGraph.Canvas);
        gptr := gptr.link;
    end;
end;
```

The most important point in this code concerns the invocation of the function draw within the loop. Despite the fact that there is only one function call written here, sometimes the function invoked will be from class TBall; at other times it will be from class TWall, or class THole. The fact that one function call might result in many different function bodies being invoked is a form of *polymorphism*. We will discuss this important topic in more detail in Chapter 14.

The routine that tests to see if a moving ball has hit anything in the function Ball.update is similarly simplified. This can be seen in the complete source listing provided in Appendix B.

Chapter Summary

In our second case study we have examined a graphical program that simulates the behavior of a pool table. Once more our motivation for presenting the case study was not so much the problem being addressed, as it was the manner in which the problem was being solved. The balls, holes, and walls in the game are described as independently reacting agents. When the user interacts with the game by means of a mouse press, the effect is to impart some energy to the cue ball, thereby forcing it to move. Thereafter the objects interact among themselves, until all the balls run out of energy.

Further Information

In the first two editions of the book this case study was presented in Apple Object Pascal, instead of Delphi. Those versions can still be found in the web site, `ftp://ftp.cs.orst.edu/pub/budd/oopintro`.

As we noted at the beginning of this chapter, our concern here is with the programming language aspects of Delphi, which are only a small part of the entire Delphi system. Further information on Delphi can be found in [Lischner 2000, Kerman 2002]. Borland also provides a wealth of on-line material with the Delphi integrated program development system.

Self Study Questions

1. Give some examples of how the design makes holes, walls, and balls responsible for their own behavior.

2. By making each graphical object into a separate class, and making each responsible for a different aspect of behavior, the object-oriented design is able to support a great deal of information hiding. This, in turn, leads to programs which are considerably easier to modify than when conventional techniques are used. To illustrate this, explain what sections of code would need to be modified to produce each of the following changes:

   - Colored balls, rather than black and white.
   - Walls which absorb a bit of energy when they reflect a ball.
   - Holes which make a sound when they absorb a ball.
   - Balls which make a sound when they strike.

Exercises

1. Suppose you want to perform a certain action every time the billiards program executes the event loop task. Where is the best place to insert this code?

2. Suppose you want to make the balls colored. What portions of the program do you need to change?

3. Suppose you want to add pockets on the side walls, as on a conventional pool table. What portions of the program do you need to change?

4. The billiards program uses a "breadth-first" technique, cycling repeatedly over the list of balls, moving each a little as long as any ball has energy. An alternative, and in some ways more object-oriented, approach is to have each ball continue to update itself as long as it possesses any energy, and update any ball that it hits. With this technique, it is only necessary to

start the cue ball moving in order to put the simulation in motion. Revise the program to use this approach. Which do you think provides a more realistic simulation? Why?

5. A hole has the same graphical representation as a ball, namely a round black spot. Similarly the algorithms used to determine if a ball has intersected are the same for balls and holes. Given this, would it make sense to declare TBall as a child class of THole? What would be the advantages of doing so? What might be some problems introduced by this modification?

Chapter 8

Inheritance and Substitution

The first step in learning object-oriented programming is understanding the basic philosophy of organizing the performance of a task as the interaction of loosely coupled software components. This organizational approach was the central lesson in the case studies of Chapters 6 and 7.

The *next* step in learning object-oriented programming is organizing classes into a hierarchical structure based on the concept of inheritance. By *inheritance*, we mean the property that instances of a child class (or subclass) can access both data and behavior (methods) associated with a parent class (or superclass).

8.1 An Intuitive Description of Inheritance

Let us return to Chris and Fred, the customer and florist from the first chapter. There is a certain behavior we expect florists to exhibit, not because they are florists but simply because they are shopkeepers. For example, we expect Fred to request money for a transaction and in turn give back a receipt. These activities are not unique to florists, but are common to bakers, grocers, stationers, car dealers, and other merchants. It is as though we have associated certain behavior with the general category Shopkeeper, and as Florists are a specialized form of shopkeepers, the behavior is automatically identified with the subclass.

In programming languages, inheritance means that the behavior and data associated with child classes are always an *extension* (that is, a larger set) of the properties associated with parent classes. A subclass will have all the properties of the parent class, and other properties as well. On the other hand, since a child class is a more specialized (or restricted) form of the parent class, it is also, in a certain sense, a *contraction* of the parent type. This tension between inheritance as expansion and inheritance as contraction is a source for much of the power inherent in the technique, but at the same time it causes much confusion as to

its proper employment. We will see this when we examine a few of the uses of inheritance in a subsequent section.

Inheritance is always transitive, so that a class can inherit features from superclasses many levels away. That is, if class Dog is a subclass of class Mammal, and class Mammal is a subclass of class Animal, then Dog will inherit attributes both from Mammal and from Animal.

8.1.1 The Is-a Test

As we noted in Chapter 2, there is a rule-of-thumb that is commonly used to test whether two concepts should be linked by an inheritance relationship. This heuristic is termed the *is-a* test. The is-a test says that to tell if concept A should be linked by inheritance to concept B, try forming the English sentence "A(n) A is a(n) B." If the sentence "sounds right" to your ear, then inheritance is most likely appropriate in this situation. For example, the following all seem like reasonable assertions:

A Bird is an Animal
A Cat is a Mammal
An Apple Pie is a Pie
A TextWindow is a Window
A Ball is a GraphicalObject
An IntegerArray is an Array

On the other hand, the following assertions seem strange for one reason or another, and hence inheritance is likely not appropriate:

A Bird is a Mammal
An Apple Pie is an Apple
An Engine is a Car
A Ball is a Wall
An IntegerArray is an Integer

There are times when inheritance can reasonably be used even when the is-a test fails. Nevertheless, for the vast majority of situations, it gives a reliable indicator for the appropriate use of the technique.

8.1.2 Reasons to Use Inheritance

Although there are many uses for the mechanism of inheritance, two motivations far outweigh all other concerns:

- Inheritance as a means of code reuse. Because a child class can inherit behavior from a parent class, the code does not need to be rewritten for the child. This can greatly reduce the amount of code needed to develop a new idea.

- Inheritance as a means of concept reuse. This occurs when a child class overrides behavior defined in the parent. Although no code is shared between parent and child, the child and parent share the definition of the method.

An example of the latter was described in the previous chapter. The variable that was declared as holding a GraphicalObject could, in fact, be holding a Ball. When the message draw was given to the object, the code from class Ball, and not from GraphicalObject, was the method selected. Both code and concept reuse often appear in the same class hierarchies.

8.2 Inheritance in Various Languages

Object-oriented languages can be divided into those languages that require every class to inherit from an existing parent class, and those languages that do not. Java, Smalltalk, Objective-C, and Delphi Pascal are examples of the former, while C++ and Apple Pascal are examples of the latter. For the former group we have already seen the syntax used to indicate inheritance, for example in Figure 4.3 of Chapter 4. In Figure 8.1 we reiterate some of these, and also show the syntax used for some of the languages in the second group.

One advantage given to those languages that insist that all classes inherit from an existing class is that there is then a single root that is ancestor to all objects. This root class is termed Object in Smalltalk and Objective-C, and termed TObject in Delphi Pascal. Any behavior provided by this root class in inherited by all objects. Thus, every object is guaranteed to possess a common minimal level of functionality.

The disadvantage of a single large inheritance tree is that it combines all classes into a tightly coupled unit. By having several independent inheritance hierarchies, programs in C++ and other languages that do not make this restriction are not forced to carry a large library of classes, only a few of which may be used in any one program. Of course, that means there is no programmer-defined functionality that *all* objects are guaranteed to possess.

In part, the differing views of objects are one more distinction between languages that use dynamic typing and those that use static typing. In dynamic languages, objects are characterized chiefly by the messages they understand. If two objects understand the same set of messages and react in similar ways, they are, for all practical purposes, indistinguishable regardless of the relationships of their respective classes. Under these circumstances, it is useful to have all objects inherit a large portion of their behavior from a common base class.

8.3 Subclass, Subtype, and Substitution

Consider the relationship of a data type associated with a parent class to a data type associated with a derived, or child, class in a statically typed object-oriented

Public, Private and Protected

In earlier chapters we have seen the use of the terms public and private. A public feature is accessible to code outside the class definition, while a private feature is accessible only within the class definition. Inheritance introduces a third alternative. In C++ (also in C#, Delphi, Ruby and several other languages) a protected feature is accessible only within a class definition or within the definition of any child classes. Thus a protected feature is more accessible than a private one, and less accessible than a public feature. This is illustrated by the following example:

```
class Parent {
private:
    int three;
protected:
    int two;
public:
    int one;
    Parent () { one = two = three = 42; }
    void inParent ()
        { cout << one << two << three; /* all legal */ }
};

class Child : public Parent {
public:
    void inChild () {
        cout << one; // legal
        cout << two; // legal
        cout << three; // error - not legal
    }
};

void main () {
    Child c;
    cout << c.one; // legal
    cout << c.two; // error - not legal
    cout << c.three; // error - not legal
}
```

The lines marked as error will generate compiler errors. The private feature can be used only within the parent class. The protected feature only within the parent and child class. Only public features can be used outside the class definitions.

Java uses the same keyword, but there protected features are legal within the same package in which they are declared.

C++	`class Wall : public GraphicalObject {`
	`  ...`
	`}`

C#	`class Wall : GraphicalObject {`
	`  ...`
	`}`

CLOS	`(defclass Wall (GraphicalObject) () )`

Java	`class Wall extends GraphicalObject {`
	`  ...`
	`}`

Object Pascal	`type`
	`  Wall = object (GraphicalObject)`
	`  ...`
	`  end;`

Python	`class Wall(GraphicalObject):`
	`  def __init__(self):`
	`  ...`

Ruby	`class Wall < GraphicalObject`
	`  ...`
	`end`

Figure 8.1: Syntax Used to Indicate Inheritance in Several Languages

language. The following observations can be made:

- Instances of the child class must possess all data members associated with the parent class.
- Instances of the child class must implement, through inheritance at least (if not explicitly overridden) all functionality defined for the parent class. (They can also define new functionality, but that is unimportant for the present argument).
- Thus, an instance of a child class can mimic the behavior of the parent class and should be *indistinguishable* from an instance of the parent class if substituted in a similar situation.

We will see later in this chapter, when we examine the various ways in which inheritance can be used, that this is not always a valid argument. Nevertheless, it is a good description of our idealized view of inheritance. We will therefore formalize this ideal in what is called the *principle of substitution*.

The principle of substitution says that if we have two classes, A and B, such that class B is a subclass of class A (perhaps several times removed), it should be possible to substitute instances of class B for instances of class A in *any situation* with *no observable effect*.

The term *subtype* is used to refer to a subclass relationship in which the principle of substitution is maintained, to distinguish such forms from the general *subclass* relationship, which may or may not satisfy this principle. We saw a use of the principle of substitution in Chapter 7. Section 7.4 described the following procedure:

```
procedure drawBoard;
var
    gptr : GraphicalObject;
begin
        (* draw each graphical object *)
    gptr := listOfObjects;
    while gptr <> nil do begin
        gptr.draw;
        gptr := gptr.link;
    end;
end;
```

The global variable listOfObjects maintains a list of graphical objects, which can be any of three types. The variable gptr is declared to be simply a graphical object, yet during the course of executing the loop it takes on values that are, in fact, derived from each of the subclasses. Sometimes gptr holds a ball, sometimes a hole, and sometimes a wall. In each case, when the draw function is invoked, the correct method for the current value of gptr will be executed–not the method

in the declared class GraphicalObject. For this code to operate correctly it is imperative that the functionality of each of these subclasses match the expected functionality specified by the parent class; that is, the subclasses must also be subtypes.

All object oriented languages will support the principle of substitution, although some will require additional syntax when a method is overridden. Most support the concept in a very straight-forward fashion; the parent class simply holds a value from the child class. The one major exception to this is the language C++. In C++ only pointers and references truly support substitution; variables that are simply declared as value (and not as pointers) do not support substitution. We will see why this property is necessary in C++ in a later chapter.

8.3.1 Substitution and Strong Typing*

Statically typed languages (such as C++ and Object Pascal) place much more emphasis on the principle of substitution than do dynamically typed languages (such as Smalltalk and Objective-C). The reason for this is that statically typed languages tend to characterize objects by their class, whereas dynamically typed languages tend to characterize objects by their behavior. For example, a polymorphic function (a function that can take objects of various classes) in a statically typed language can ensure a certain level of functionality only by insisting that all arguments be subclasses of a given class. Since in a dynamically typed language arguments are not typed at all, the same requirement would be simply that an argument must be able to respond to a certain set of messages.

An example of this difference would be a function that requires an argument be an instance of a subclass of Measureable, as opposed to a function that requires an argument understand the messages lessThan and equal. The former is characterizing an object by its class, the latter is characterizing an object by its behavior. Both forms of type checking are found in object-oriented languages.

8.4 Overriding and Virtual Methods

In Chapter 1 we noted that child classes may sometimes find it necessary to *override* the behavior they would otherwise inherit from their parent classes. In syntactic terms, what this means is that a child class will define a method using the same name and type signature as one found in the parent class. When overriding is combined with substitution, we have the situation where a variable is declared as one class, but holds a value from a child class, and a method matching a given message is found in both classes. In almost all cases what we want to have happen in this situation is to execute the method found in the child class, ignoring the method from the parent class.

[0] Section headings followed by an asterisk indicate optional material.

C++	`class GraphicalObject {`
	`public:`
	`  virtual void draw();`
	`};`
	
	`class Ball : public Graphicalobject {`
	`public:`
	`  virtual void draw(); // virtual optional here`
	`};`

C#	`class GraphicalObject {`
	`  public viritual void draw () { ... }`
	`}`
	
	`class Ball : Graphical Object {`
	`  public override void draw () { ... }`
	`}`

Delphi	`type`
	`  GraphicalObject = class (TObject)`
	`  ...`
	`  procedure draw; virtual;`
	`  end;`
	
	`  Ball = class (GraphicalObject)`
	`  ...`
	`  procedure draw; override;`
	`  end;`

Object Pascal	`type`
	`  GraphicalObject = object`
	`  ...`
	`  procedure draw;`
	`  end;`
	
	`  Ball = object (GraphicalObject)`
	`  ...`
	`  procedure draw; override;`
	`  end;`

Figure 8.2: Overriding in Various Languages

In many object oriented languages (Smalltalk, Java) this desired behavior will occur naturally, as soon as a child class overrides a method in the parent class using the same type signature. Some languages, on the other hand, require the programmer to indicate that such a substitution is permitted. Many languages use the keyword virtual to indicate this. It may be necessary, as in C++, to place the keyword in the parent class[1] (indicating that overriding *may* take place; it does not indicate that it necessarily will take place) or, as in Object Pascal, in the child class (indicating that overridng *has* taken place). Or it may be required in both places, as in C# and Delphi. Figure 8.2 shows the syntax used for overriding in various langauges.

8.5 Interfaces and Abstract Classes

In Chapter 4 we briefly introduced the concept of an interface in Java and other languages. As with classes, interfaces are allowed to inherit from other interfaces and are even permitted to inherit from multiple parent interfaces. Although the specification that a new class inherits from a parent class and the specification that it implements an interface are not exactly the same, they are sufficiently similar that we will henceforth use the term *inheritance* to indicate both actions.

Several object-oriented languages support an idea, termed an *abstract method* that is mid-way between classes and interfaces. In Java and C#, for example, a class can define one or more methods using the keyword abstract. No body is then provided for the method. A child class *must* implement any abstract methods before an instance of the class can be created. Thus, abstract methods specify behavior in the parent class but the behavior itself must be provided by the child class:

```
abstract class Window {
    ...
    abstract public void paint ();  // draw contents of window
    ...
}
```

An entire class can be named as abstract, whether or not it includes any abstract methods. It is not legal to create an instance of an abstract class, it is only legal to use it as a parent class for purposes of inheritance.

In C++ the idea of an abstract method is termed a *pure virtual method*, and is indicated using the assignment operator:

```
class Window {
public:
    ...
```

[1] Virtual overriding in C++ is actually more complex, for reasons we will develop in the next several chapters.

```
    virtual void paint () = 0; // assignment makes it pure virtual
};
```

A class can have both abstract (or pure virtual) methods and non-abstract methods. A class in which all methods were declared as abstract (or pure virtual) would correspond to the Java idea of an interface.

Abstract methods can be simulated even when the language does not provide explicit support for the concept. In Smalltalk, for example, programmers frequently define a method so as to generate an error if it is invoked, with the expectation that it will be overwritten in child classes:

```
writeTo: stream
    ↑ self error: 'subclass must override writeTo'
```

This is not exactly the same as a true abstract method, since it does not preclude the creation of instances of the class. Nevertheless, if an instance is created and this method invoked the program will quickly fail, and hence such errors are easily detected.

8.6 Forms of Inheritance

Inheritance is used in a surprising variety of ways. In this section we will describe a few of its more common uses. Note that the following list represents general abstract categories and is not intended to be exhaustive. Furthermore, it sometimes happens that two or more descriptions are applicable to a single situation, because some methods in a single class use inheritance in one way while others use it in another.

8.6.1 Subclassing for Specialization (Subtyping)

Probably the most common use of inheritance and subclassing is for specialization. In subclassing for specialization, the new class is a specialized form of the parent class but satisfies the specifications of the parent in all relevant respects. Thus, in this form the principle of substitution is explicitly upheld. Along with the following category (subclassing for specification) this is the most ideal form of inheritance, and something that a good design should strive for.

Here is an example of subclassing for specialization. A class Window provides general windowing operations (moving, resizing, iconification, and so on). A specialized subclass TextEditWindow inherits the window operations and *in addition*, provides facilities that allow the window to display textual material and the user to edit the text values. Because the text edit window satisfies all the properties we expect of a window in general (thus, a TextEditWindow window is a subtype of Window in addition to being a subclass), we recognize this situation as an example of subclassing for specialization.

8.6.2 Subclassing for Specification

Another frequent use for inheritance is to guarantee that classes maintain a certain common interface–that is, they implement the same methods. The parent class can be a combination of implemented operations and operations that are deferred to the child classes. Often, there is no interface change of any sort between the parent class and the child class–the child merely implements behavior described, but not implemented, in the parent.

This is in essence a special case of subclassing for specialization, except that the subclasses are not refinements of an existing type but rather realizations of an incomplete abstract specification. In such cases the parent class is sometimes known as an *abstract specification class*.

A class that implements an interface is always fulfilling this form of inheritance. However, subclassing for specification can also arise in other ways. In the billiards simulation example presented in Chapter 7, for example, the class GraphicalObject was an abstract class since it described, but did not implement, the methods for drawing the object and responding to a hit by a ball. The subsequent classes Ball, Wall, and Hole then used subclassing for specification when they provided meanings for these methods.

In general, subclassing for specification can be recognized when the parent class does not implement actual behavior but merely defines the behavior that will be implemented in child classes.

8.6.3 Subclassing for Construction

A class can often inherit almost all of its desired functionality from a parent class perhaps changing only the names of the methods used to interface to the class, or modifying the arguments in a certain fashion. This may be true even if the new class and the parent class fail to share the *is-a* relationship.

For example, the Smalltalk class hierarchy implements a generalization of an array called Dictionary. A dictionary is a collection of key-value pairs, like an array, but the keys can be arbitrary values. A *symbol table*, such as might be used in a compiler, can be considered a dictionary indexed by symbol names in which the values have a fixed format (the symbol-table entry record). A class SymbolTable can therefore be made a subclass of the class Dictionary, with new methods defined that are specific to the use as a symbol table. Another example might be forming a *set* data abstraction on top of a base class which provides *list* methods. In both these cases, the child class is not a more specialized form of the parent class, because we would never think of substituting an instance of the child class in a situation where an instance of the parent class is being used.

A common use of subclassing for construction occurs when classes are created to write values to a binary file, for example, in a persistent storage system. A parent class may implement only the ability to write raw binary data. A subclass is constructed for every structure that is saved. The subclass implements a save procedure for the data type, which uses the behavior of the parent type to do

the actual storage.[2]

```
class Storable {
    void writeByte(unsigned char);
    };

class StoreMyStruct : public Storable {
    void writeStruct (MyStruct & aStruct);
    };
```

Subclassing for construction tends to be frowned upon in statically typed languages, since it often directly breaks the principle of substitution (forming subclasses that are not subtypes). On the other hand, because it is often a fast and easy route to developing new data abstractions, it is widely employed in dynamically typed languages. Many instances of subclassing for construction can be found in the Smalltalk standard library.

We will investigate an example of subclassing for construction in Chapter 9. We will also see that C++ provides an interesting mechanism, *private inheritance*, which permits subclassing for construction without breaking the principle of substitution.

8.6.4 Subclassing for Generalization

Using inheritance to subclass for generalization is, in a certain sense, the opposite of subclassing for specialization. Here, a subclass extends the behavior of the parent class to create a more general kind of object. Subclassing for generalization is often applicable when we build on a base of existing classes that we do not wish to modify, or cannot modify.

Consider a graphics display system in which a class Window has been defined for displaying on a simple black-and-white background. You could create a subtype ColoredWindow that lets the background color be something other than white by adding an additional field to store the color and overriding the inherited window display code that specifies the background be drawn in that color.

Subclassing for generalization frequently occurs when the overall design is based primarily on data values and only secondarily on behavior. This is shown in the colored window example, since a colored window contains data fields that are not necessary in the simple window case.

As a rule, subclassing for generalization should be avoided in favor of inverting the type hierarchy and using subclassing for specialization. However, this is not always possible.

[2] This example illustrates the blurred lines between categories. If the child class implements the storage using a different method name, we say it is subclassing for construction. If, on the other hand, the child class uses the same name as the parent class, we might say the result is subclassing for specification.

8.6.5 Subclassing for Extension

While subclassing for generalization modifies or expands on the existing functionality of an object, subclassing for extension adds totally new abilities. Subclassing for extension can be distinguished from subclassing for generalization in that the latter must override at least one method from the parent and the functionality is tied to that of the parent. Extension simply adds new methods to those of the parent, and the functionality is less strongly tied to the existing methods of the parent.

An example of subclassing for extension is a StringSet class that inherits from a generic Set class but is specialized for holding string values. Such a class might provide additional methods for string-related operations–for example, "search by prefix," which returns a subset of all the elements of the set that begin with a certain string value. These operations are meaningful for the subclass, but are not particularly relevant to the parent class.

As the functionality of the parent remains available and untouched, subclassing for extension does not contravene the principle of substitution and so such subclasses are always subtypes.

8.6.6 Subclassing for Limitation

Subclassing for limitation occurs when the behavior of the subclass is smaller or more restrictive than the behavior of the parent class. Like subclassing for generalization, subclassing for limitation occurs most frequently when a programmer is building on a base of existing classes that should not, or cannot, be modified.

For example, an existing class library provides a double-ended-queue, or *deque*, data structure. Elements can be added or removed from either end of the deque, but the programmer wishes to write a stack class, enforcing the property that elements can be added or removed from only one end of the stack.

In a manner similar to subclassing for construction, the programmer can make the Stack class a subclass of the existing Deque class, and can modify or override the undesired methods so that they produce an error message if used. These methods override existing methods and eliminate their functionality, which characterizes subclassing for limitation. (Overriding, by which a subclass changes the meaning of a method defined in a parent class, will be discussed in a subsequent chapter).

Because subclassing for limitation is an explicit contravention of the principle of substitution, and because it builds subclasses that are not subtypes, it should be avoided whenever possible.

8.6.7 Subclassing for Variance

Subclassing for variance is employed when two or more classes have similar implementations but do not seem to possess any hierarchical relationships between the abstract concepts represented by the classes. The code necessary to control a mouse, for example, may be nearly identical to the code required to control

a graphics tablet. Conceptually, however, there is no reason why class Mouse should be made a subclass of class Tablet, or the other way. One of the two classes is then arbitrarily selected to be the parent, with the common code being inherited by the other and device-specific code being overridden.

Usually, however, a better alternative is to factor out the common code into an abstract class, say PointingDevice, and to have both classes inherit from this common ancestor. As with subclassing for generalization, this choice may not be available if you are building on a base of existing classes.

8.6.8 Subclassing for Combination

A common situation is a subclass that represents a *combination* of features from two or more parent classes. A teaching assistant, for example, may have characteristics of both a teacher and a student, and can therefore logically behave as both. The ability of a class to inherit from two or more parent classes is known as *multiple inheritance*; it is sufficiently subtle and complex that we will devote an entire chapter to the concept.

8.6.9 Summary of the Forms of Inheritance

We can summarize the various forms of inheritance by the following table:

- Specialization. The child class is a special case of the parent class; in other words, the child class is a subtype of the parent class.
- Specification. The parent class defines behavior that is implemented in the child class but not in the parent class.
- Construction. The child class makes use of the behavior provided by the parent class, but is not a subtype of the parent class.
- Generalization. The child class modifies or overrides some of the methods of the parent class.
- Extension. The child class adds new functionality to the parent class, but does not change any inherited behavior.
- Limitation. The child class restricts the use of some of the behavior inherited from the parent class.
- Variance. The child class and parent class are variants of each other, and the class-subclass relationship is arbitrary.
- Combination. The child class inherits features from more than one parent class. This is multiple inheritance and will be the subject of a later chapter.

8.7 Variations on Inheritance*

In this section we will examine a number of mostly single-language specific variations on the themes of inheritance and overriding.

8.7.1 Anonymous Classes in Java

Occasionally a situation arises where a programmer needs to create a simple class, and knows there will never be more than one instance of the class. Such an object is often termed a *singleton*. The Java programming language provides a mechanism for creating such an object without even having to give a name to the class being used to define the object. Hence the name for this technique, *anonymous classes*.

In order to be able to create an anonymous class, several requirements must be met:

1. Only one instance of the anonymous class can be created.
2. The class must inherit from a parent class or interface, and not require a constructor for initialization.

These two conditions frequently arise in the context of user interfaces. For example, in Chapter 22 we will encounter a class named ButtonAdapter that is used to create graphical buttons. To give behavior to a button, the programmer must form a new class that inherits from ButtonAdapter, and overrides the method pressed. Since there is only one such object, this can be done with an anonymous class (also sometimes termed a *class definition expression*).

Graphical elements are added to a window using the method add. To place a new button in a window, all that is necessary is the following:

```
Window p = ...;

p.add (new ButtonAdapter("Quit"){
    public void pressed () { System.exit(0); }
    }
);
```

Study carefully the argument being passed to the add operator. It includes the creation of a new value, indicated by the new operator. But rather than ending the expression with the closing parenthesis on the argument list for new, a curly brace appears as if in a class definition. In fact, this is a new class definition. A subclass of ButtonAdapter is being formed, and a single instance of this class will be created. Any methods required by this new class are given immediately in-line. In this case, the new class overrides the method named pressed. The closing curly brace terminates the anonymous class expression.

[0] Section headings followed by an asterisk indicate optional material.

8.7.2 Inheritance and Constructors

A constructor, you will recall, is a procedure that is invoked implicitly during the creation of a new object value, and which guarantees that the newly created object is properly initialized. Inheritance complicates this process, since both the parent and the new child class may have initialization code to perform. Thus code from both classes must be executed.

In Java, C++ and other languages the constructor for both parent and child will automatically be executed as long as the parent constructor does not require additional parameters. When the parent does require parameters, the child must explicitly provide them. In Java this is done using the keyword `super`:

```
class Child extends Parent {
    public Child (int x) {
        super (x + 2); // invoke parent constructor
    ...
    }
}
```

In C++ the same task is accomplished by writing the parent class name in the form of an initializer:

```
class Child : public Parent {
public:
    Child (int x) : Parent(x+2) { ... }
};
```

In Delphi a constructor for a child class must always invoke the constructor for the parent class, even if the parent class constructor takes no arguments. The syntax is the same for executing the parent class behavior in any overridden method:

```
constructor TChildClass.Create;
begin
    inherited Create; // execute constructor in parent
end
```

Arguments to the parent constructor are added as part of the call:

```
constructor TChildClass.Create (x : Integer);
begin
    inherited Create(x + 2);
end
```

Similarly, an initialization method in Python does not automatically invoke the function in the parent, hence the programmer must not forget to do this task:

```
class Child(Parent):
    def __init__ (self):
        # first initialize parent
        Parent.__init__(self)
        # then do our initialization
        ...
```

8.7.3 Virtual Destructors

Recall from Chapter 5 that in C++ a destructor is a function that will be invoked just before the memory for a variable is recovered. Destructors are used to perform whatever tasks are necessary to ensure a value is properly deleted. For example, a destructor will frequently free any dynamically allocated memory the variable may hold.

If substitution and overriding are anticipated, then it is important that the destructor be declared as virtual. Failure to do so may result in destructors for child classes not being invoked. This following example shows this error:

```
class Parent {
public:
        // warning, destructor not declared virtual
    ~Parent () { cout << "in parent\n"; }
};

class Child : public Parent {
public:
    ~Child () { cout << "in child\n"; }
};
```

If an instance of the child class is held by a pointer to the parent class and subsequently released (say, by a delete statement), then only the parent destructor will be invoked.

```
    Parent * p = new Child();
    delete p;
in parent
```

If the parent destructor is declared as virtual, then both the parent and child destructors will be executed. In C++ it is a good idea to include a virtual

destructor, even if it performs no action, if there is any possibility that a class may later be subclassed.

8.8 The Benefits of Inheritance

In this section we will describe some of the many important benefits of the proper use of inheritance.

8.8.1 Software Reusability

When behavior is inherited from another class, the code that provides that behavior does not have to be rewritten. This may seem obvious, but the implications are important. Many programmers spend much of their time rewriting code they have written many times before–for example, to search for a pattern in a string or to insert a new element into a table. With object-oriented techniques, these functions can be written once and reused.

Other benefits of reusable code include increased reliability (the more situations in which code is used, the greater the opportunities for discovering errors) and the decreased maintenance cost because of sharing by all users of the code.

8.8.2 Code Sharing

Code sharing can occur on several levels with object-oriented techniques. On one level, many users or projects can use the same classes. (Brad Cox [Cox 1986] calls these software-ICs, in analogy to the integrated circuits used in hardware design). Another form of sharing occurs when two or more classes developed by a single programmer as part of a project inherit from a single parent class. For example, a Set and an Array may both be considered a form of Collection. When this happens, two or more types of objects will share the code that they inherit. This code needs to be written only once and will contribute only once to the size of the resulting program.

8.8.3 Consistency of Interface

When two or more classes inherit from the same superclass, we are assured that the behavior they inherit will be the same in all cases. Thus, it is easier to guarantee that interfaces to similar objects are in fact similar, and that the user is not presented with a confusing collection of objects that are almost the same but behave, and are interacted with, very differently.

8.8.4 Software Components

In Chapter 1, we noted that inheritance provides programmers with the ability to construct reusable software components. The goal is to permit the development of new and novel applications that nevertheless require little or no actual coding.

Already, several such libraries are commercially available, and we can expect many more specialized systems to appear in time.

8.8.5 Rapid Prototyping

When a software system is constructed largely out of reusable components, development time can be concentrated on understanding the new and unusual portion of the system. Thus, software systems can be generated more quickly and easily, leading to a style of programming known as *rapid prototyping* or *exploratory programming*. A prototype system is developed, users experiment with it, a second system is produced that is based on experience with the first, further experimentation takes place, and so on for several iterations. Such programming is particularly useful in situations where the goals and requirements of the system are only vaguely understood when the project begins.

8.8.6 Polymorphism and Frameworks

Software produced conventionally is generally written from the bottom up, although it may be *designed* from the top down. That is, the lower-level routines are written, and on top of these slightly higher abstractions are produced, and on top of these even more abstract elements are generated. This process is like building a wall, where every brick must be laid on top of an already laid brick.

Normally, code portability decreases as one moves up the levels of abstraction. That is, the lowest-level routines may be used in several different projects, and perhaps even the next level of abstraction may be reused, but the higher-level routines are intimately tied to a particular application. The lower-level pieces can be carried to a new system and generally make sense standing on their own; the higher-level components generally make sense (because of declarations or data dependencies) only when they are built on top of specific lower-level units.

Polymorphism in programming languages permits the programmer to generate high-level reusable components that can be tailored to fit different applications by changes in their low-level parts. We will have much more to say about this topic in subsequent chapters.

8.8.7 Information Hiding

A programmer who reuses a software component needs only to understand the nature of the component and its interface. It is not necessary for the programmer to have detailed information concerning matters such as the techniques used to implement the component. Thus, the interconnectedness between software systems is reduced. We earlier identified the interconnected nature of conventional software as being one of the principle causes of software complexity.

8.9 The Costs of Inheritance

Although the benefits of inheritance in object-oriented programming are great, almost nothing is without cost of one sort or another. For this reason, we must consider the cost of object-oriented programming techniques, and in particular the cost of inheritance.

8.9.1 Execution Speed

It is seldom possible for general-purpose software tools to be as fast as carefully hand-crafted systems. Thus, inherited methods, which must deal with arbitrary subclasses, are often slower than specialized code.

Yet, concern about efficiency is often misplaced.[3] First, the difference is often small. Second, the reduction in execution speed may be balanced by an increase in the speed of software development. Finally, most programmers actually have little idea of how execution time is being used in their programs. It is far better to develop a working system, monitor it to discover where execution time is being used, and improve those sections, than to spend an inordinate amount of time worrying about efficiency early in a project.

8.9.2 Program Size

The use of any software library frequently imposes a size penalty not imposed by systems constructed for a specific project. Although this expense may be substantial, as memory costs decrease the size of programs becomes less important. Containing development costs and producing high-quality and error-free code rapidly are now more important than limiting the size of programs.

8.9.3 Message-Passing Overhead

Much has been made of the fact that message passing is by nature a more costly operation than simple procedure invocation. As with overall execution speed, however, overconcern about the cost of message passing is frequently penny-wise and pound-foolish. For one thing, the increased cost is often marginal–perhaps two or three additional assembly-language instructions and a total time penalty of 10 percent. (Timing figures vary from language to language. The overhead of message passing will be much higher in dynamically bound languages, such as Smalltalk, and much lower in statically bound languages, such as C++.) This increased cost, like others, must be weighed against the many benefits of the object-oriented technique.

A few languages, notably C++, make a number of options available to the programmer that can reduce the message-passing overhead. These include elim-

[3] The following quote from an article by Bill Wulf offers some apt remarks on the importance of efficiency: "More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason–including blind stupidity" [Wulf 1972].

inating the polymorphism from message passing (qualifying invocations of member functions by a class name, in C++ terms) and expanding inline procedures. Similarly, the Delphi Pascal programmer can choose dynamic methods, which use a run-time lookup mechanism, or virtual methods, which use a slightly faster technique. Dynamic methods are inherently slower, but require less space.

8.9.4 Program Complexity

Although object-oriented programming is often touted as a solution to software complexity, in fact, overuse of inheritance can often simply replace one form of complexity with another. Understanding the control flow of a program that uses inheritance may require several multiple scans up and down the inheritance graph. This is what is known as the *yo-yo* problem, which we will discuss in more detail in a later chapter.

Chapter Summary

In this chapter we begin a detailed examination of inheritance and substitution, a topic that will be continued through the next several chapters. When a child class declares that it inherits from a parent class, code in the parent class does not have to be rewritten. Thus inheritance is a powerful mechanism of code reuse. But this is not the only reason to use inheritance. In the abstract, a child class is a representative of the category formed by the parent class, and hence it makes sense that an instance of the child class could be used in those situations where we expect an instance of the parent class. This is known as the principle of *substitution*. But this is only an idealization. Not all types of inheritance support this idea behavior.

We have described various forms of inheritance, noting when they seem to support substitution and when they may not.

The chapter concludes with descriptions of both the benefits of inheritance and the costs incurred through the use of the technique.

Further Reading

Many of the ideas introduced in this chapter will be developed and explored in more detail in subsequent chapters. Overriding is discussed in detail in Chapter 16. We will discuss static and dynamic typing more in Chapter 10, and polymorphism in more detail in Chapter 14.

In Section 8.1.2 we noted that inheritance is used both as a mechanism of code reuse and concept reuse. The fact that the same feature is serving two different purposes is a frequent criticism levied against object-oriented languages. Many writers have advocated separating these two tasks, for example using inheritance of classes only for code reuse, and using inheritance of interfaces (as, for example, in Java) for substitution (concept reuse). While this approach has a theoretical

appeal, from a practical standpoint it complicates the task of programming, and has not been widely adopted. See Exercise 5 for one way this could be accomplished.

The list describing the forms of inheritance is adopted from [Halbert 1987], although I have added some new categories of my own. The editable-window example is from [Meyer 1988a].

The principle of substitution is sometimes referred to as the *Liskov Substitution Principle*, since an early discussion of the idea was presented by Barbara Liskov and John Guttag [Liskov 1986].

Self Study Questions

1. In what ways is a child class an extension of its parent? In what ways is it a contraction?
2. What is the is-a test for inheritance?
3. What are the two major reasons for the use of inheritance?
4. What is the principle of substitution? What is the argument used to justify its application?
5. How is a class that contains abstract methods similar to an interface? If not all methods are abstract, how is it different?
6. What features characterize each of the following forms of inheritance?
   (a) Subclassing for Specialization
   (b) Subclassing for Specification
   (c) Subclassing for Construction
   (d) Subclassing for Generalization
   (e) Subclassing for Extension
   (f) Subclassing for Limitation
   (g) Subclassing for Variance
7. Why is subclassing for construction not normally considered to be a good idea?
8. Why is subclassing for limitation not a good idea?
9. How does inheritance facilitate software reuse?
10. How does it encourage consistency of interface?
11. How does it support the idea of rapid prototyping?
12. How does it encourage the principle of information hiding?

13. An anonymous class combines what two activities?

14. Why is the execution time cost incurred by the use of inheritance not usually important? What are some situations where it would be important?

Exercises

1. Suppose you were required to program a project in a non-object oriented language, such as Pascal or C. How would you simulate the notion of classes and methods? How would you simulate inheritance? Could you support multiple inheritance? Explain your answer.

2. We noted that the execution overhead associated with message passing is typically greater than the overhead associated with a conventional procedure call. How might you measure these overheads? For a language that supports both classes and procedures (such as C++ or Object Pascal), devise an experiment to determine the actual performance penalty of message passing.

3. Consider the three geometric concepts of a line (infinite in both directions), a ray (fixed at a point, infinite in one direction), and a segment (a portion of a line with fixed end points). How might you structure classes representing these three concepts in an inheritance hierarchy? Would your answer differ if you concentrated more on the data representation or more on the behavior? Characterize the type of inheritance you would use. Explain the reasoning behind your design.

4. The following appeared as an illustration of inheritance in a popular journal:

> "Perhaps the most powerful concept in object-oriented programming systems is inheritance. Objects can be created by inheriting the properties of other objects, thus removing the need to write any code whatsoever! Suppose, for example, a program is to process complex numbers consisting of real and imaginary parts. In a complex number, the real and imaginary parts behave like real numbers, so all of the operations (+, -, /, *, sqrt, sin, cos, etc.) can be inherited from the class of objects call REAL, instead of having to be written in code. This has a major impact on programmer productivity."

   (a) The quote seems to indicate that class Complex could be a child class of Real. Does the assertion that the child class Complex need not write any code seem plausible?

   (b) Does this organization make sense in terms of the data members each class must maintain? Why or why not?

(c) Does this organization make sense in terms of the methods each class must support? Why or why not?

(d) Can you describe a better approach for creating a class Complex using an existing class Real? What benefit does your new class derive from the existing class?

5. In Section 8.1.2 we noted how inheritance is used for two different purposes; as a vehicle for code reuse, and a vehicle for substitution. Among the major object-oriented langauges, Java comes closest to separating these two purposes, since the language supports both classes and interfaces. But it confuses the two topics by continuing to allow substitution for class values. Suppose we took the next step, and changed the Java language to eliminate substitution for class types. This could be accomplished by making the following two modifications to the language:

   - A variable declared as a class could hold values of the class, but not of child classes.
   - If a parent class indicates that it supports an interface, the child class would not automatically support the interface, but would have to explicitly indicate this fact in its class heading.

   We maintain inheritance and substitution of interfaces; a variable declared as an interface could hold a value from any class that implemented the interface.

   (a) Show that any class hierarchy, and any currently legal assignment, could be rewritten in this new framework. (You will need to introduce new interfaces).

   (b) Although the resulting system is much cleaner from a theoretical standpoint, what has been lost? Why did the designers of Java not follow this approach?

Chapter 9

Case study – A Card Game

In this third case study we will examine a simple card game, a version of solitaire. A slightly different rendition of this program was presented in C++ in the first edition of the present book, and rewritten to use the MFC library in another book [Budd 1999]. The program was translated into Java in the second edition, and revised once again in Java in yet another book [Budd 98b]. The program presented here is one more revision, this time translated into C#.

I have used this case study in so many different forms because the development of this program is a good illustration of the power of inheritance and overriding. We will get to those aspects, after first considering some of the basic elements of the game. The complete source for the program can be viewed in Appendix C.

9.1 The Class PlayingCard

Wherever possible, software development should strive for the creation of general purpose reusable classes; classes that make minimal demands on their environment and hence can be carried from one application to another. This idea is illustrated by the first class, which represents a playing card. The class defining the playing card abstraction is shown in Figure 9.1. We have examined aspects of this class in earlier chapters.

The methods isFaceUp, rank, suit and color have been writen as *properties*. Since they include only a get clause, and no set feature, they are properties that can be read and not modified. Two enumerated data types are used by the playing card class. The enumerated type Color is provided by the standard run-time system. The class Suits is specific to this project, and is defined as follows:

```
public enum Suits { Spade, Diamond, Club, Heart };
```

```
public class PlayingCard
{
    public PlayingCard (Suits sv, int rv)
        { s = sv; r = rv; faceUp = false; }

    public bool isFaceUp
    {
        get { return faceUp; }
    }

    public void flip ()
    {
        faceUp = ! faceUp;
    }

    public int rank
    {
        get { return r; }
    }

    public Suits suit
    {
        get { return s; }
    }

    public Color color
    {
        get
        {
            if ( suit == Suits.Heart || suit == Suits.Diamond )
                { return Color.Red; }
            return Color.Black;
        }
    }

    private bool faceUp;
    private int r;
    private Suits s;
}
```

Figure 9.1: The Definition of the Class PlayingCard

In C#, unlike C++, enuemrated constants must be prefixed by their type name. You can see this in the method color through the use of names such as Color.Black or Suits.Heart, instead of simply Black or Heart.

The class PlayingCard has no information about the application in which it is developed, and can easily be moved from this program to another program that uses the playing card abstraction.

9.2 Data and View Classes

Techniques used in the creation of visual interfaces have undergone frequent revisions, and this trend will likely continue for the foreseeable future. For this reason it is useful to separate classes that contain data values, such as the PlayingCard abstraction, from classes that are used to provide a graphical display of those values. By doing so the display classes can be modified or replaced, as necessary, leaving the original data classes untouched.

The display of the card abstraction will be provided by the class CardView. To isolate the library specific aspects of the card view, the actual display method is declared as *abstract* (see Section 8.5). This will later be subclassed and replaced with a function that will use the C# graphics facilities to generate the graphical interface:

```
public abstract class CardView
{
    public abstract void display (PlayingCard aCard, int x, int y);

    public static int Width = 50;
    public static int Height = 70;
}
```

By not only separating playing cards from card views, but also separating the concept of a card view from a specific implementation, we isolate any code that is specific to a single graphics library. The effect is that the majority of the code in the application has no knowledge of the graphics library being used. This facilitates any future modifications to the graphics aspects of the application, which are the features most likely to change.

The CardView class encapsulates a pair of static constants that represent the height and width of a card on the display. The PlayingCard class itself knows nothing about how it is displayed. One abstract method is prototyped. This method will display the face of a card at a given position on the display.

It can be argued that even including the height and width as values in this class is introducing some platform dependencies, however these are less likely to change than are the libraries used to perform the actual graphical display.

9.3 The Game

The version of solitaire we will describe is known as klondike. The countless variations on this game make it probably the most common version of solitaire; so much so that when you say "solitaire," most people think of klondike. The version we will use is that described in [Morehead 1949]; variations on the basic game are numerous.

The layout of the game is shown in Figure 9.2. A single standard pack of 52 cards is used. The *tableau*, or playing table, consists of 28 cards in 7 piles. the first pile has 1 card, the second 2, and so on up to 7. The top card of each pile is initially face up; all other cards are face down.

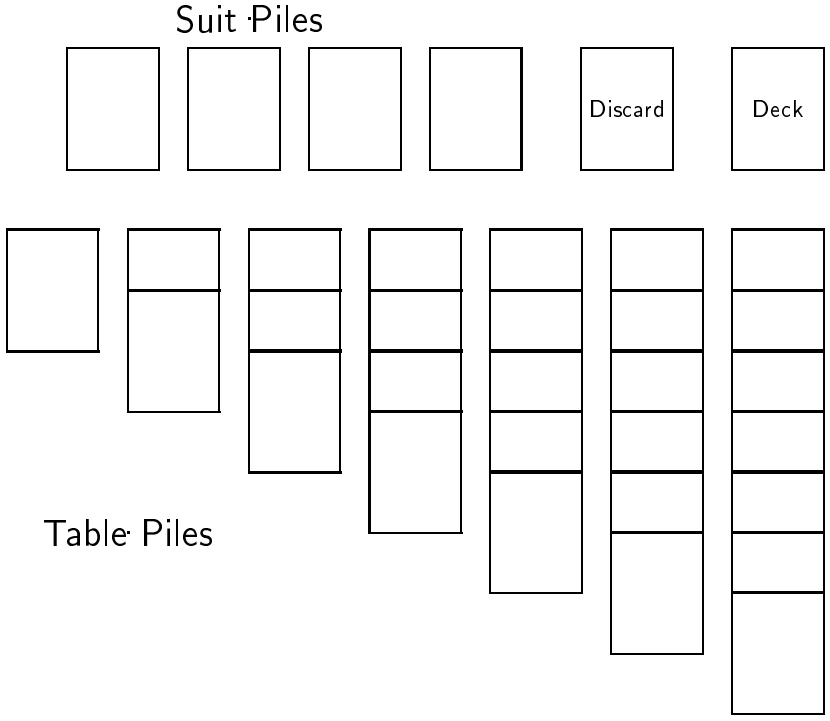


Figure 9.2: Layout for the solitaire game

The suit piles (sometimes called *foundations*) are built up from aces to kings in suits. They are constructed above the tableau as the cards become available. The object of the game is to build all 52 cards into the suit piles.

The cards that are not part of the tableau are initially all in the *deck*. Cards in the deck are face down, and are drawn one by one from the deck and placed, face up, on the *discard pile*. From there, they can be moved onto either a tableau pile or a foundation. Cards are drawn from the deck until the pile is empty; at

this point, the game is over if no further moves can be made.

Cards can be placed on a tableau pile only on a card of next-higher rank and opposite color. They can be placed on a foundation only if they are the same suit and next higher card or if the foundation is empty and the card is an ace. Spaces in the tableau that arise during play can be filled only by kings.

The topmost card of each tableau pile and the topmost card of the discard pile are always available for play. The only time more than one card is moved is when an entire collection of face-up cards from a tableau (called a *build*) is moved to another tableau pile. This can be done if the bottommost card of the build can be legally played on the topmost card of the destination. Our initial game will not support the transfer of a build, but we will discuss this as a possible extension. The topmost card of a tableau is always face up. If a card is moved from a tableau, leaving a face-down card on the top, the latter card can be turned face up.

From this short description, it is clear that the game of solitaire mostly involves manipulating piles of cards. Each type of pile has many features in common with the others and a few aspects unique to the particular type. In the next section, we will investigate in detail how inheritance can be used in such circumstances to simplify the implementation of the various card piles by providing a common base for the generic actions and permitting this base to be redefined when necessary.

9.4 Card Piles–Inheritance in Action

Much of the behavior we associate with a card pile is common to each variety of pile in the game. For example, each pile maintains a collection containing the cards in the pile, and the operations of inserting and deleting elements from this collection are common. Other operations are given default behavior in the class CardPile, but they are sometimes overridden in the various subclasses. The class CardPile is shown in Figure 9.3.

Each card pile maintains the coordinate location for the upper left corner of the pile, as well as a collection that contains the card in the pile. The Stack abstraction from the standard run-time library is used to hold the cards. All these values are set by the constructor for the class. The data fields, located near the end of the declaration, are declared as protected and thus accessible to member functions associated with this class and to member functions associated with subclasses.

The three functions top(), pop(), and isEmpty() manipulate the list of cards, using functions provided by the Stack class. The remaining five operations defined in class CardPile are common to the abstract notion of our card piles, but they differ in details in each case. For example, the function canTake(PlayingCard) asks whether it is legal to place a card on the given pile. A card can be added to a foundation pile, for instance, only if it is an ace and the foundation is empty, or if the card is of the same suit as the current topmost card in the pile and has

```
public class CardPile {
    public CardPile (int xl, int yl )
        { x = xl; y = yl; pile = new Stack(); }

    public PlayingCard top
        { get { return (PlayingCard) pile.Peek (); } }

    public bool isEmpty
        { get { return pile.Count == 0; }    }

    public PlayingCard pop
        { get { return (PlayingCard) pile.Pop (); } }

        // the following are sometimes overridden
    public virtual bool includes (int tx, int ty ) {
        return( ( x <= tx ) && ( tx <= x + CardView.Width ) &&
            ( y <= ty ) && ( ty <= y + CardView.Height ) );
    }

    public virtual void select (int tx, int ty ) {
        // do nothing–override
    }

    public virtual void addCard (PlayingCard aCard )
        { pile.Push(aCard); }

    public virtual void display (CardView cv) {
        if ( isEmpty ) {
            cv.display(null, x, y);
        } else {
            cv.display((PlayingCard) pile.Peek(), x, y );
        }
    }

    public virtual bool canTake (PlayingCard aCard)
        { return false; }

    protected int x, y; // coordinates of the card pile
    protected Stack pile; // card pile data
}
```

Figure 9.3: Description of the Class CardPile

the next-higher value. A card can be added to a tableau pile, on the other hand, only if the pile is empty and the card is a king, or if it is of the opposite color as the current topmost card in the pile and has the next lower value.

The actions of the five virtual functions defined in CardPile can be characterized as follows:

includes –Determines if the coordinates given as arguments are contained within the boundaries of the pile. The default action simply tests the topmost card; this is overridden in the tableau piles to test all card values.

canTake –Tells whether a pile can take a specific card. Only the tableau and suit piles can take cards, so the default action is simply to return no; this is overridden in the two classes mentioned.

addCard –Adds a card to the card list. It is redefined in the discard pile class to ensure that the card is face up.

display –Displays the card deck. The default method merely displays the topmost card of the pile, but is overridden in the tableau class to display a column of cards. The top half of each hidden card is displayed. So that the playing surface area is conserved, only the topmost and bottommost face-up cards are displayed (this permits us to give definite bounds to the playing surface).

select –Performs an action in response to a mouse click. It is invoked when the user selects a pile by clicking the mouse in the portion of the playing field covered by the pile. The default action does nothing, but is overridden by the table, deck, and discard piles to play the topmost card, if possible.

The following table illustrates the important benefits of inheritance. Given five operations and five classes, there are 25 potential methods we might have had to define. By making use of inheritance we need to implement only 13. Furthermore, we are guaranteed that each pile will respond in the same way to similar requests.

	CardPile	SuitPile	DeckPile	DiscardPile	TableauPile
includes	×				×
canTake	×	×			×
addCard	×			×	
display	×				×
select	×		×	×	×

9.4.1 The Default Card Pile

We will examine each of the subclasses of CardPile in detail, pointing out various uses of object-oriented features. Each of the five virtual methods is first defined in the class CardPile. These implementations will represent the default behavior, should they not be overridden. The implementation of these methods was shown in Figure 9.3.

9.4.2 The Suit Piles

The simplest subclass is the class SuitPile, which represents the pile of cards at the top of the playing surface. This is the pile being built up in suit from ace to king. The implementation of this class is as follows:

```
public class SuitPile : CardPile {
    public SuitPile (int x, int y) : base(x, y) {    }

    public override bool canTake (PlayingCard aCard ) {
        if( isEmpty )
            { return( aCard.rank == 0 ); }
        PlayingCard topCard = top;
        return( ( aCard.suit == topCard.suit ) &&
           ( aCard.rank == topCard.rank + 1 ) );
    }
}
```

The class SuitPile defines only two methods. The constructor for the class takes two integer arguments and does nothing more than invoke the constructor for the parent class CardPile.

The method canTake overrides the similarly named method in the parent class. Note the use of the keyword override that indicates this fact. This method determines whether or not a card can be placed on the pile. A card is legal if the pile is empty and the card is an ace (that is, has rank zero) or if the card is the same suit as the topmost card in the pile and of the next higher rank (for example, a three of spades can only be played on a two of spades). Since the methods rank and suit were declared as properties, they can be invoked without parenthesis.

All other behavior of the suit pile is the same as that of our generic card pile. When selected, a suit pile does nothing. When a card is added it is simply inserted into the stack. To display the pile only the topmost card is drawn.

9.4.3 The Deck Pile

The DeckPile maintains the deck from which new cards are drawn. It differs from the generic card pile in two ways. When constructed, rather than creating an empty pile of cards, it initializes itself by first creating an array containing the 52 cards in a conventional deck, then randomly selecting elements from this collection so as to generate a sorted deck. The method select is invoked when the mouse button is used to select the card deck. If the deck is empty, it does nothing. Otherwise, the topmost card is removed from the deck and added to the discard pile.

```
public class DeckPile : CardPile {
```

```
    public DeckPile (int x, int y) : base(x, y) {
        // create the new deck
        // first put cards into a local array
        ArrayList aList = new ArrayList ();
        for( int i = 0; i <= 12; i++) {
            aList.Add(new PlayingCard(Suits.Heart, i));
            aList.Add(new PlayingCard(Suits.Diamond, i));
            aList.Add(new PlayingCard(Suits.Spade, i));
            aList.Add(new PlayingCard(Suits.Club, i));
        }
            // then pull them out randomly
        Random myRandom = new Random( );
        for(int count = 0; count < 52; count++) {
            int index = myRandom.Next(aList.Count);
            addCard( (PlayingCard) aList [index] );
            aList.RemoveAt(index);
        }
    }

    public override void select (int tx, int ty) {
        if ( isEmpty ) { return; }
        Game.discardPile().addCard( pop );
    }
}
```

The implementation of the select method presents us with a new problem. When the mouse is pressed on the deck pile, the desired action is to move a card from the deck pile on to the discard pile, turning it face up in the process. The problem is that we now need to refer to a single unique card pile, namely the pile that represents the discard pile.

One approach would be to define the various card piles as global variables, which then could be universally accessed. In fact, this approach is used in the program described in my earlier C++ version of the game in the first edition of this book. But many languages, such as Java and C#, do not have global variables. There is good reason for this. Global variables tend to make it difficult to understand the flow of information through a program, since they can be accessed from any location (that's what makes them global).

A better and more object-oriented alternative to the use of global variables is a series of static values. This reduces the number of global values to one; the class name. Static methods in the class can then be used to access further state. In our program we will name this class Game. A discussion of the details of this class will be postponed until after the description of the various card piles.

9.4.4 The Discard Pile

The class DiscardPile redefines the addCard and select methods. The class is described as follows:

```
public class DiscardPile : CardPile {
    public DiscardPile (int x, int y ) : base(x, y) { }

    public override void addCard (PlayingCard aCard) {
        if( ! aCard.isFaceUp )
            { aCard.flip(); }
        base.addCard( aCard );
    }

    public override void select (int tx, int ty) {
        if( isEmpty ) { return; }
        PlayingCard topCard = pop;
        for( int i = 0; i < 4; i++ ) {
            if( Game.suitPile(i).canTake( topCard ) ) {
                Game.suitPile(i).addCard( topCard );
                return;
            }
        }

        for( int i = 0; i < 7; i++ ) {
            if( Game.tableau(i).canTake( topCard ) ) {
                Game.tableau(i).addCard( topCard );
                return;
            }
        }
        // nobody can use it, put it back on our stack
        addCard(topCard);
    }
}
```

The implementation of these methods is interesting in that they exhibit two very different forms of inheritance. The select method *overrides* or *replaces* the default behavior provided by class CardPile, replacing it with code that when invoked (when the mouse is pressed over the card pile) checks to see if the topmost card can be played on any suit pile or, alternatively, on any tableau pile. If the card cannot be played, it is kept in the discard pile.

The method addCard is a different sort of overriding. Here the behavior is a *refinement* of the default behavior in the parent class. That is, the behavior of the parent class is completely executed, and, in addition, new behavior is added. In this case, the new behavior ensures that when a card is placed on

the discard pile it is always face up. After satisfying this condition, the code in the parent class is invoked to add the card to the pile. The keyword base is necessary to avoid the confusion with the addCard method being defined. In Java the same problem would be addressed by sending a message to super (as in super.addCard(aCard)).

Another form of refinement occurs in the constructors for the various subclasses. Each must invoke the constructor for the parent class to guarantee that the parent is properly initialized before the constructor performs its own actions. The parent constructor is invoked by an initializer clause inside the constructor for the child class.

9.4.5 The Tableau Piles

The most complex of the subclasses of CardPile is that used to hold a tableau, or table pile. The implementation of this class redefines nearly all of the virtual methods defined in ClassPile. When initialized, by the constructor, the tableau pile removes a certain number of cards from the deck, placing them in its own pile. The number of cards so removed is determined by an additional argument to the constructor. The topmost card of this pile is then displayed face up.

```
public class TablePile : CardPile {
    public TablePile (int x, int y, int c) : base(x, y) {
        // initialize our pile of cards
        for(int i = 0; i < c; i++ ) {
            addCard(Game.deckPile().pop);
        }
        top.flip();
    }

    public override bool canTake (PlayingCard aCard ) {
        if( isEmpty ) { return(aCard.rank == 12); }
        PlayingCard topCard = top;
        return( ( aCard.color != topCard.color ) &&
            ( aCard.rank    == topCard.rank - 1 ) );
    }

    public override bool includes (int tx, int ty) {
        return( ( x <= tx ) && ( tx <= x + CardView.Width ) &&
                ( y <= ty ) );
    }

    public override void select (int tx, int ty) {
        if( isEmpty ) { return; }
        // if face down, then flip
        PlayingCard topCard = top;
```

```
        if( ! topCard.isFaceUp ) {
            topCard.flip();
            return;
        }
        // else see if any suit pile can take card
        topCard = pop;
        for(int i = 0; i < 4; i++ ) {
            if( Game.suitPile(i).canTake( topCard ) ) {
                Game.suitPile(i).addCard( topCard );
                return;
            }
        }
        // else see if any other table pile can take card
        for(int i = 0; i < 7; i++ ) {
            if( Game.tableau(i).canTake( topCard ) ) {
                Game.tableau(i).addCard( topCard );
                return;
            }
        }
        addCard( topCard );
    }

    public override void display (CardView cv) {
        Object [ ] cardArray = pile.ToArray();
        int size = pile.Count;
        int hs = CardView.Height / 2; // half size
        int ty = y;
        for (int i = pile.Count - 1; i >= 0; i--) {
            cv.display((PlayingCard) cardArray[i], x, ty);
            ty += hs;
        }
    }
}
```

A card can be added to the pile (method canTake) only if the pile is empty and the card is a king, or if the card is the opposite color from that of the current topmost card and one smaller in rank. When a mouse press is tested to determine if it covers the pile (method includes) the bottom bound is not tested since the pile may be of variable length. When the pile is selected, the topmost card is flipped if it is face down. If it is face up, an attempt is made to move the card first to any available suit pile, and then to any available table pile. Only if no pile can take the card is it left in place. Finally, to display a pile all the underlying cards are displayed. The stack must be converted into an array to do this, since we must access the cards top to bottom, which is the opposite of the order that stack elements would normally be enumerated.

9.5 Playing the Polymorphic Game

The need for the class Game was described earlier. This class holds the actual card piles used by the program, making them available through methods that are declared as static. Because these methods are static, they can be accessed using only the class name as a basis.

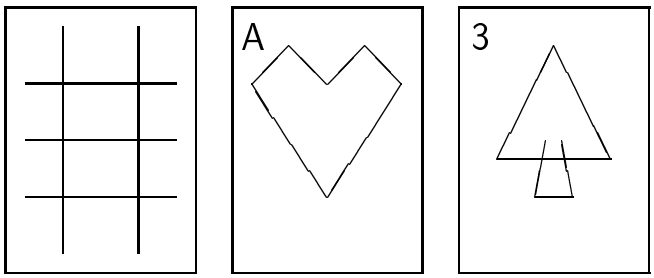
The definition of this class is shown in Figure 9.4. The game manager stores the various card piles in an array, one that is declared as CardPile, although in fact the values are polymorphic and hold a variety of different types of card piles. These values are initialized in the constructor, which is declared as static. A static constructor will be executed when the program begins execution.

By storing the card values in a polymorphic array the game manager need not distinguish the characteristics of the individual piles. For example, to repaint the display it is only necessary to tell each pile to repaint itself. The method display will be different, depending upon the actual type of card pile. Similarly, to respond to a mouse down, the manager simply cycles through the list of card piles.

9.6 The Graphical User Interface

We have taken pains in the development of this program to isolate the details of both the graphical user interface and of the high level program execution. This is because of all the elements of a program, the user interface is the most likely to require change as new graphical libraries are introduced or existing libraries are changed. Similarly the way that applications are initiated using C# introduces details that would have obscured the overall design of the application.

The card images are simple line drawings, as shown below. Diamonds and hearts are drawn in red, spades and clubs in black. The hash marks on the back are drawn in yellow.



We deal first with the user interface. Recall that the display of a card was provided by a method CardView, that was described as abstract. To produce actual output, we must create a subclass that implements the pure virtual methods. This class we will call WinFormsCardView:

```
public class Game {
    static Game () {
        allPiles = new CardPile[ 13 ];
        allPiles[0] = new DeckPile(335, 5 );
        allPiles[1] = new DiscardPile(268, 5 );
        for( int i = 0; i < 4; i++ ) {
            allPiles[2 + i] = new SuitPile(15 + 60 * i, 5);
        }
        for( int i = 0; i < 7; i++ ) {
            allPiles[6+i] = new TablePile(5+55*i, 80, i+1);
        }
    }

    public static void paint (CardView cv) {
        for( int i = 0; i < 13; i++ ) {
            allPiles[i].display(cv );
        }
    }

    public static void mouseDown (int x, int y) {
        for( int i = 0; i < 13; i++ ) {
            if( allPiles[i].includes(x, y) ) {
                allPiles [i].select(x, y);
            }
        }
    }

    public static CardPile deckPile ()
        { return allPiles[0]; }

    public static CardPile discardPile ()
        { return allPiles[1]; }

    public static CardPile tableau (int index)
        { return allPiles[6+index]; }

    public static CardPile suitPile (int index)
        { return allPiles[2+index]; }

    private static CardPile[] allPiles;
}
```

Figure 9.4: The Class Game

```
public class WinFormsCardView : CardView {
    public WinFormsCardView (Graphics aGraphicsObject) {
        g = aGraphicsObject;
    }

    public override void display (PlayingCard aCard,int x,int y) {
        if  (aCard == null) {
            Pen myPen = new Pen(Color.Black,2);
            Brush myBrush = new SolidBrush (Color.White);
            g.FillRectangle(myBrush,x,y,CardView.Width,CardView.Height);
            g.DrawRectangle(myPen,x,y,CardView.Width,CardView.Height);
        } else {
            paintCard (aCard,x,y);
        }
    }

    private void paintCard (PlayingCard aCard,int x,int y) {
        String [] names = { "A","2","3","4","5",
            "6","7","8","9","10","J","Q","K" };

        Pen myPen = new Pen (Color.Black,2);
        Brush myBrush = new SolidBrush (Color.White);

        g.FillRectangle (myBrush,x,y,CardView.Width,CardView.Height);
        g.DrawRectangle(myPen,x,y,CardView.Width,CardView.Height);
        myPen.Dispose();
        myBrush.Dispose();

        // draw body of card with a new pen-color
        if (aCard.isFaceUp) {
            if (aCard.color == Color.Red) {
                myPen = new Pen  (Color.Red,1);
                myBrush = new SolidBrush (Color.Red);
            } else {
                myPen = new Pen  (Color.Blue,1);
                myBrush = new SolidBrush (Color.Blue);
            }
            g.DrawString (names[ aCard.rank ],
                new Font("Times New Roman",10),myBrush,x+3,y+7);
            if (aCard.suit == Suits.Heart) {
                g.DrawLine(myPen,x+25,y+30,x+35,y+20);
                g.DrawLine(myPen,x+35,y+20,x+45,y+30);
                g.DrawLine(myPen,x+45,y+30,x+25,y+60);
                g.DrawLine(myPen,x+25,y+60,x+5,y+30);
```

```
                g.DrawLine(myPen,x+5,y+30,x+15,y+20);
                g.DrawLine(myPen,x+15,y+20,x+25,y+30);
            } else if (aCard.suit == Suits.Spade) {
                ... // see code in appendix
            } else if (aCard.suit == Suits.Diamond) {
                ...
            } else if (aCard.suit == Suits.Club) {
                ...
            }
        } else {   // face down
            myPen = new Pen (Color.Green,1);
            myBrush = new SolidBrush  (Color.Green);
            ...
        }
    }
    private Graphics g;
}
```

This is not a text on graphics, so the actual display will be rather simple. Basically, a card draws itself as a rectangle with a textual description. Empty piles are drawn in green, the backsides of cards in yellow, the faces in the appropriate color.

Graphical output in the C# library is based around a type of object from class Graphics. This object is passed as constructor to the class, and stored in the variable g. Details of the graphical output routines provided by the Windows library will not be discussed here, although many of the names are self-explanatory. The display for our game is rather primitive, consisting simply of line rectangles and the textual display of card information.

Applications in the C# framework are created by subclassing from a system provided class named System.WinForms.Form and overriding certain key methods. Much of the structure of the class is generated automatically if one uses a development environment, such as the Studio application. In the following we have marked the generated code with comments. The programmer than edits this code to fit the specific application. The final class is as follows:

```
public class Solitaire : System.WinForms.Form {
        // start of automatically generated code
        private System.ComponentModel.Container components;

        public Solitaire() {
            InitializeComponent();
        }

        public override void Dispose() {
            base.Dispose();
```

```
            components.Dispose();
        }

        private void InitializeComponent() {
        this.components = new System.ComponentModel.Container ();
        this.Text = "Solitaire";
        this.AutoScaleBaseSize = new System.Drawing.Size (5, 13);
        this.ClientSize = new System.Drawing.Size (392, 373);
    }
        // end of automatically generated code

    protected override void OnMouseDown (MouseEventArgs e ) {
        Game.mouseDown(e.X, e.Y);
        this.Invalidate(); // force screen redraw
        }

    protected override void OnPaint (PaintEventArgs pe ) {
        Graphics g = pe.Graphics;
        CardView cv = new WinFormsCardView(g);
        Game.paint(cv);
    }

    public static void Main(string[] args)
        { Application.Run(new Solitaire()); }
}
```

The window class is responsible for trapping the actual mouse presses and repainting the window. In our application these activities are simply passed on to the game manager. As with Java, execution begins with the method named Main. This method invokes a static method from a system class named Application, passing it an instance of the game controller class.

Chapter Summary

The solitaire game is a standard example program, found in many textbooks. We have here used the program as a case study to illustrate a number of important concepts. In the design of the PlayingCard and CardView classes, we have separated a model from a view. This is important, since aspects of the view are likely to change more rapidly than aspects of the model. Extending this further, we have defined the view as an abstract class, and thereby hidden all Windows-specific features in an implementation of this class. Moving to a different graphical library would therefore simply involve changing the implementation of this abstract class.

Probably the most notable feature of the game is the use of inheritance and overriding, exemplified by the classes CardPile and its various subclasses. Through the use of overriding we avoid having to write a large amount of code. Furthermore, the use of a polymorphic variable to reference the various classes simplifies the task of redrawing the screen or handling mouse operations.

Further Reading

Source for the various earlier versions of this program can be found on my website, `http://www.cs.orst.edu/~budd`.

We have in this simple application only scratched the surface of the functionality provided by the C# system. However, the details of how Windows programs are created are complicated, and beyond the issues being discussed here. A good introduction to the C# system is provided by Gunnerson [Gunnerson 2000].

Self Study Questions

1. Why should the class PlayingCard be written so as to have no knowledge of the application in which it is being used?

2. Why is it useful to separate the class PlayingCard from the class that will draw the image of the playing card in the current application?

3. Why is it further useful to define the interface for CardView as an abstract class, and then later supply an implementation of this class that uses the C# graphics facilities?

4. What are the different types of card piles in this solitare game?

5. What methods in CardPile are potentially overridden? What methods are not overridden? How can you tell from the class description which are which?

6. In what way does the variable allPiles exhibit polymorphism?

7. How does the polymorphism in allPiles simplify the design of the program?

Exercises

1. The solitaire game has been designed to be as simple as possible. A few features are somewhat annoying, but can be easily remedied with more coding. These include the following:

   (a) The topmost card of a tableau pile should not be moved to another tableau pile if there is another face-up card below it.

(b) An entire build should not be moved if the bottommost card is a king and there are no remaining face-down cards.

For each, describe what procedures need to be changed, and give the code for the updated routine.

2. The following are common variations of klondike. For each, describe which portions of the solitaire program need to be altered to incorporate the change.

   (a) If the user clicks on an empty deck pile, the discard pile is moved (perhaps with shuffling) back to the deck pile. Thus, the user can traverse the deck pile multiple times.

   (b) Cards can be moved from the suit pile back into the tableau pile.

   (c) Cards are drawn from the deck three at a time and placed on the discard pile in reverse order. As before, only the topmost card of the discard pile is available for playing. If fewer than three cards remain in the deck pile, all the remaining cards (as many as that may be) are moved to the discard pile. (In practice, this variation is often accompanied by variation 1, permitting multiple passes through the deck).

   (d) The same as variation 3, but any of the three selected cards can be played. (This requires a slight change to the layout as well as an extensive change to the discard pile class).

   (e) Any royalty card, not simply a king, can be moved onto an empty tableau pile.

3. The game "thumb and pouch" is similar to klondike except that a card may be built on any card of next-higher rank, of any suit but its own. Thus, a nine of spades can be played on a ten of clubs, but not on a ten of spades. This variation greatly improves the chances of winning. (According to Morehead [Morehead 1949], the chances of winning Klondike are 1 in 30, whereas the chances of winning thumb and pouch are 1 in 4.) Describe what portions of the program need to be changed to accommodate this variation.