

┌

└

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 year 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 of 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 “silver bullet” (a term made popular by Fred Brooks [Brooks 1987]). 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 Object Pascal (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 Object Pascal 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 inside a computer. 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 [Pullum 1991]) 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).

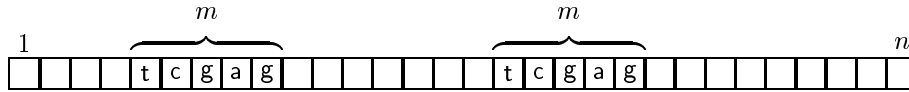
What is important is to not overgeneralize the conclusion we can draw from this simple observation. It is not that the Eskimo eye is in any significant respect different from my own, or that Eskimos can see things I cannot perceive. With time and training I could do just as well at differentiating types 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, 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 sorted this matrix by rows. If any pattern was repeated, then two adjacent rows in the sorted matrix had 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 [Pullum 1991].

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 idea 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 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 (FP or Haskell).

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

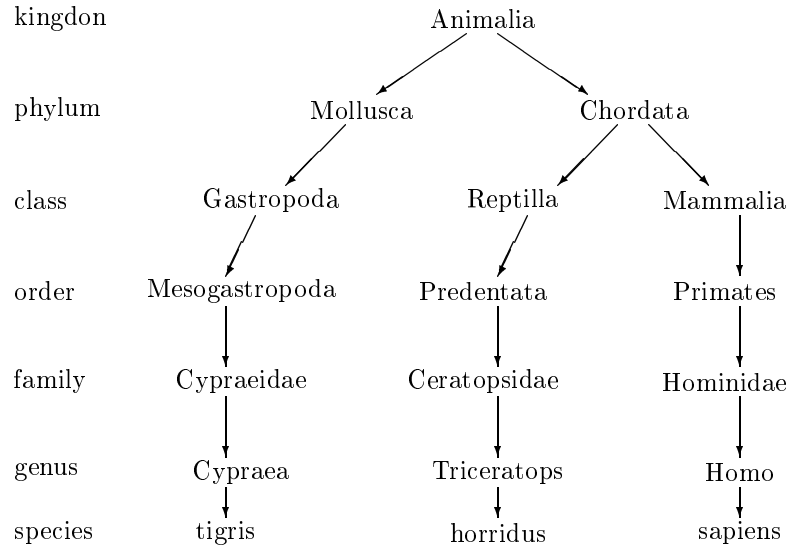


Figure 1.1 The Linnean Inheritance Hierarchy

it belongs. **2.** Any example or model. [Late Latin *paradigma*, 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 Linnæus (1707–1778) (Figure 1.1), if not further back to Plato. 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 usually 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 Robin wishes to send flowers to a friend named Chris, who lives in another city. Because of the distance, Robin cannot simply pick the flowers and take them to Chris in person. Nevertheless, it is a task that is easily solved. Robin simply walks to a nearby flower shop, run by a florist named Fred. Robin will tell Fred the kinds of flowers to send to Chris, and the address to which they should be delivered. Robin 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. Robin 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 Chris 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 Chris's city had obtained her flowers from a 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 inter-

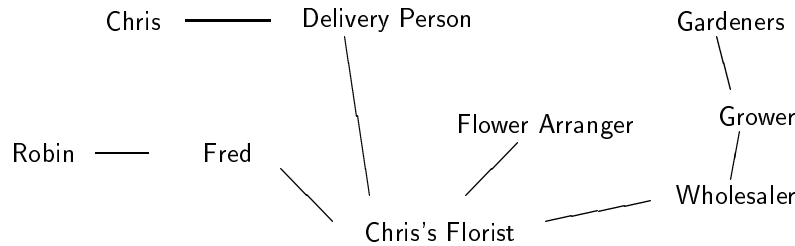


Figure 1.2 The community of agents helping delivery flowers

acting 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 Robin's problem began with a request given to the florist Fred. This request led to other requests, which led to still more requests, until the flowers ultimately reached Robin's friend Chris. 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 by which activities are initiated:

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 dependent on the receiver and can vary with different receivers. Robin 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 Chris). 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 Robin were to ask Kenneth, a dentist, to send flowers to Chris, 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*. Robin's request for action indicates only the desired outcome (flowers sent to Chris). Fred is free to pursue any technique that achieves the desired objective, and in doing so will not be hampered by interference from Robin.

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 Robin has only dealt with Fred a few times, Robin has a rough idea of the transaction that will occur inside Fred's flower shop. Robin is able to make certain assumptions based on previous experience with other florists, and hence Robin 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

Robin has more information about Fred—not necessarily because Fred is a florist but because he is a shopkeeper. Robin 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 Robin has of Shopkeepers is also true of Florists and hence of Fred.

One way to think about how Robin 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 Robin 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 Robin 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.

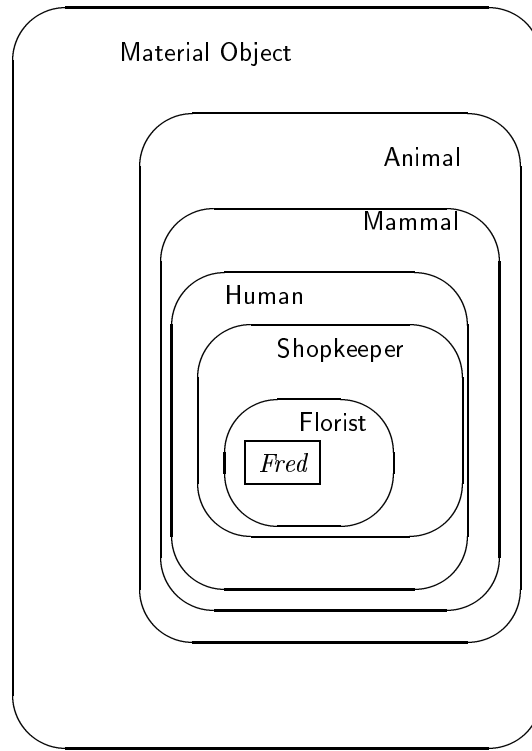


Figure 1.3 – The categories surrounding Fred.

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 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, Robin's dog Fido, Phyl the platypus who lives at the zoo, and the flowers the Robin is sending to Chris.

Information that Robin possess about Fred because Fred is an instance of class `Human` is also applicable to Elizabeth, for example. Information that Robin 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*

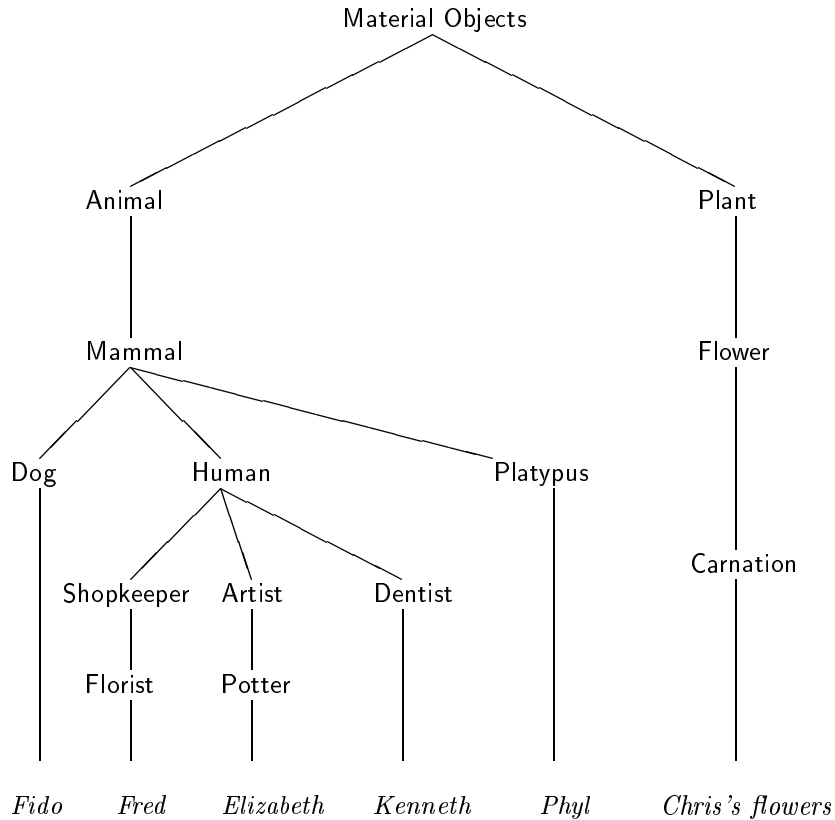


Figure 1.4 – A class hierarchy for various material objects.

higher in the tree. An *abstract parent class* is a class (such as `Mammal`) for which there are no direct instances; it is used only to create subclasses.

1.4.6 Method Binding, Overriding, and Exceptions

Phyl the platypus presents a problem for our simple organizing structure. Robin 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 Robin's messages, but use different methods to respond, is one form of *polymorphism*. We will discuss this important part of object-oriented programming in starting in Chapter 14. As explained, that Robin 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.

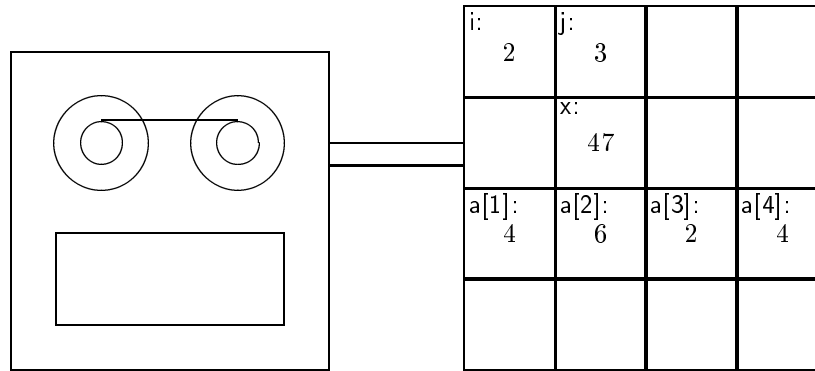


Figure 1.5 – Visualization of imperative programming.

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, we 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].



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

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 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 often easier to teach object-oriented programming concepts to computer novices than to computer professionals. Novice users quickly adapt the metaphors with which they are already comfortable from their everyday life, whereas seasoned computer professionals are 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 pure 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 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.

¹I 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.

- An object is an encapsulation of *state* (data values) and *behavior* (operations). Thus, an object is in many ways similar to a module or an abstract data type.
- 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.
- Objects and classes extend the concept of abstract data types by adding the notion of *inheritance*. Classes can be 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.

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 seem to 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].

It was the Swedish botanist Carolus Linnæus, you will recall, who developed the ideas of genus, species, and so forth. This is the prototypical hierarchical organization scheme exhibiting inheritance, since abstract classifications describe features that are largely common to all subclassifications. Most inheritance hierarchies closely follow the model of Linnæus.

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 article by Alan Kay in *Scientific American* [Kay 1977].)

┌

└

Chapter 2

Layers of 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.

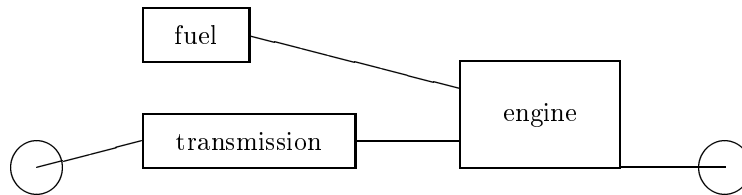
Fundamentally, people use only a few simple tools to create, understand, or manage complex systems. One of the most important techniques is termed *abstraction*.

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 or details.

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*.

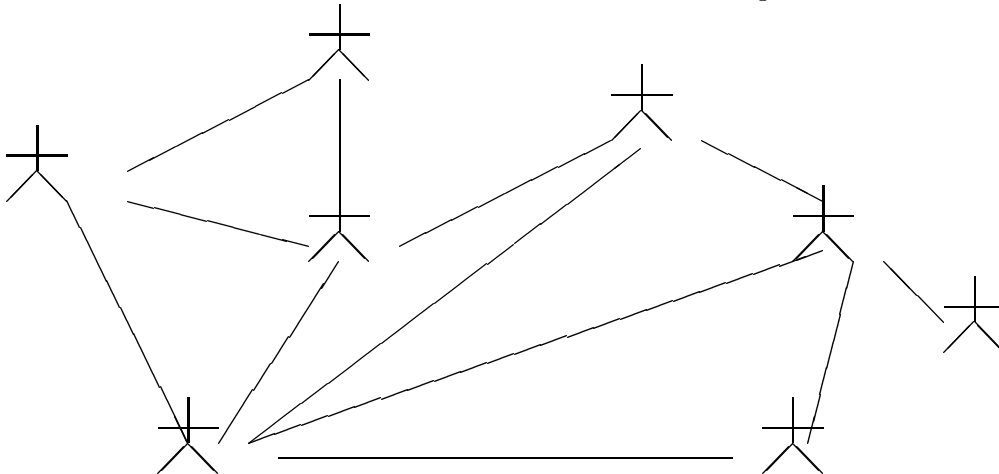
Information Hiding

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.

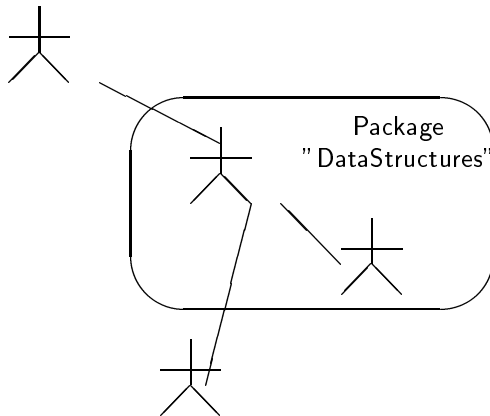
At the highest level a program is viewed as a “community” of objects that must interact with each other in order to achieve their common goal:



This notion of community is expressed in object-oriented development at two distinct levels. 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 write, which must interact with each other in the virtual universe in order to further their common goals. Key ideas such as information hiding and abstraction are applicable to both levels.

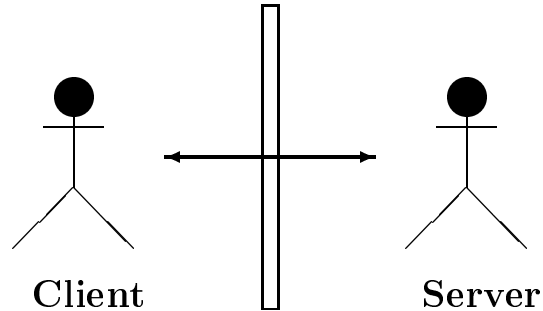
Each object in this community provides a service that is used by other members of the organization. At this highest level of abstraction the important features to emphasize are the lines of communication and cooperation, and the way in which the members must interact with each other.

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 Object Pascal. The unit allows certain names to be exposed to the world outside the unit, while other features remain hidden inside the unit.



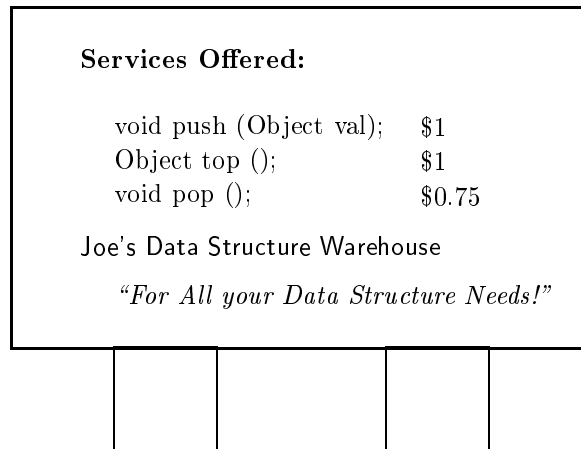
For readers familiar with concepts found in earlier languages, this notion of a unit is the heir to the idea of a *module* in languages such as C or Pascal. Later in this chapter we will present a short history of programming language abstractions, and note the debt that ideas of object-oriented programming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

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, ... {
    ...
}
```

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.

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 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 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,

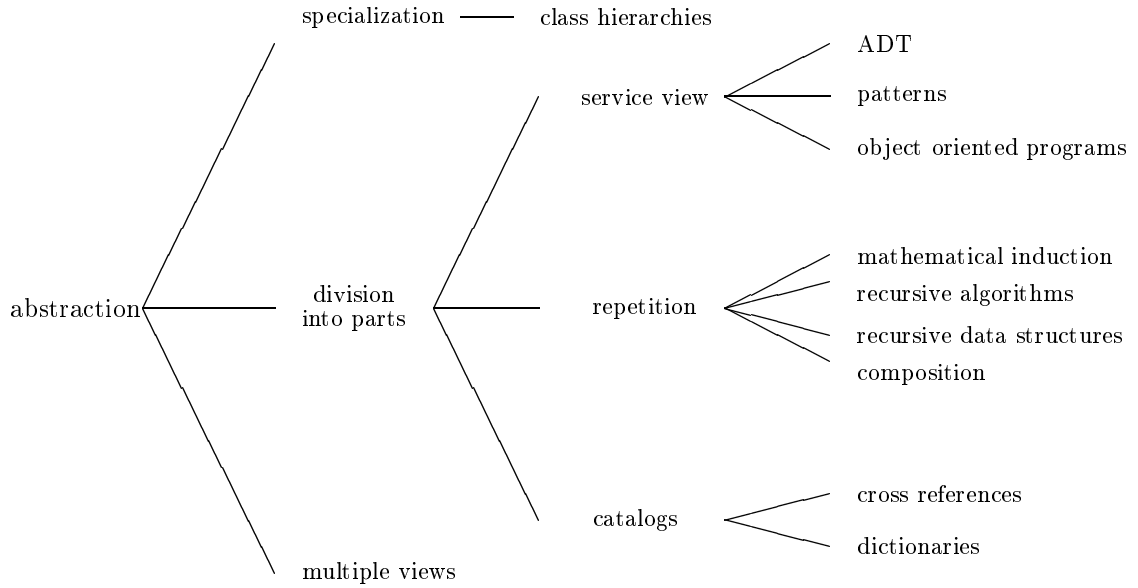


Figure 2.1 Some Techniques for Handling Complexity, with Examples

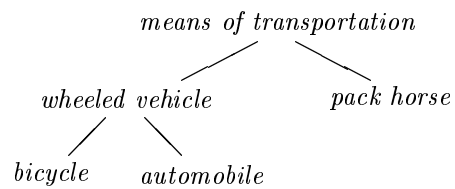
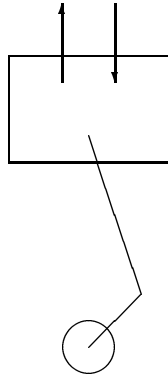


Figure 2.2 Layers of Specialization

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.

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.

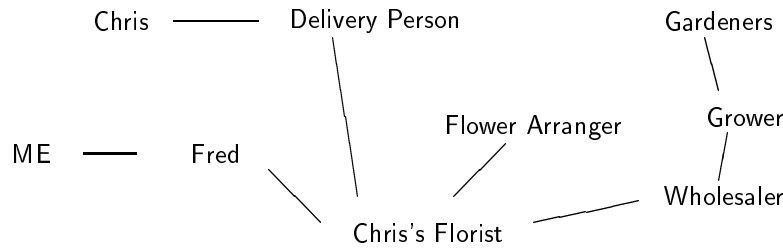
The division between interface and implementation not only makes it easier to understand a design at a high level (since the description of an interface is much simpler than the description of any specific implementation), but also make possible the interchangeability of software components (as I can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

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.



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 than composition, so that the following pattern represents the set of three letter values that begin with `ab`, and end with either an `a`, `c` or `d`:

$$\text{aba} \mid \text{abc} \mid \text{abd}$$

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

$$\text{ab}(\text{a} \mid \text{c} \mid \text{d})$$

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`.

$$(((\text{a} \mid \text{b})^* \text{c}) \mid \text{dd}) \text{a}$$

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 manager 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

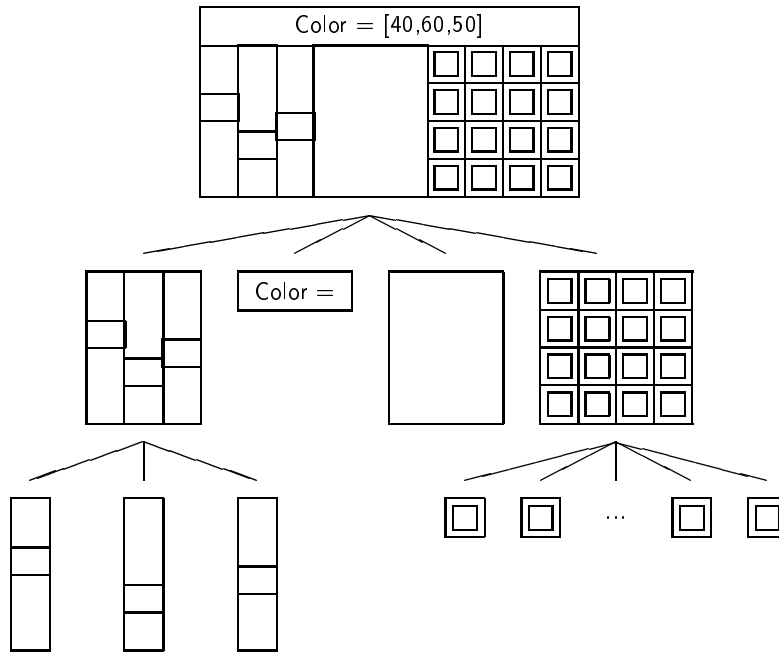


Figure 2.3 Composition in the Creation of User Interfaces

Handling Exceptions

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.

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 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

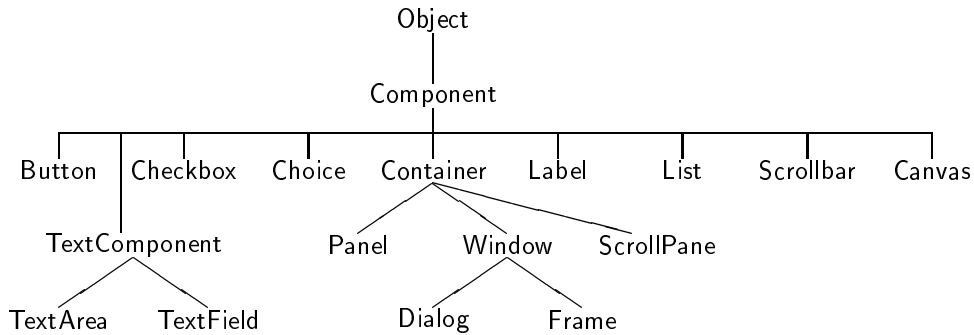


Figure 2.4 The AWT class hierarchy

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 a 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:

<code>setTitle(String)</code>	inherited from class <code>Frame</code>
<code>setSize(int, int)</code>	inherited from class <code>Component</code>
<code>show()</code>	inherited from class <code>Window</code>
<code>repaint()</code>	inherited from class <code>Component</code>
<code>paint()</code>	inherited from <code>Component</code> , 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. 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 sense, object-oriented techniques are not at all revolutionary, but are rather a natural outcome of a long progression from procedures, to modules, to abstract data types, and finally to objects.

2.3.1 Procedures

Procedures and functions were two of the first abstraction mechanisms to be widely used in programming languages. 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. The other programmers 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

```
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.

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.

2.3.2 Block Scoping

The block-scoping mechanism of Algol and its successors, such as Pascal, offers slightly more control over name visibility than does a simple distinction between local and global names. At first, we might hope that this ability solves the information-hiding problem. Unfortunately, it does not. Any scope that

permits access to the four named procedures must also permit access to their common data. To solve this problem, a different structuring mechanism had to be developed.

```
begin
  var
    datastack : array [ 1 .. 100 ] of integer;
    datatop : integer;

    procedure init; ...
    procedure push(val : integer); ...
    function pop : integer; ...
  ...
end;
```

2.3.3 Modules

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.

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?

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

An *abstract data type* is a programmer-defined data type that can be manipulated like the system-defined data types. As with system-defined types, an abstract data type corresponds to a set (perhaps infinite in size) of legal data values and a number of primitive operations that can be performed on those values. Users can create variables with values that range over the set of legal values and can operate on those values using the defined operations. For example, our intrepid programmer could define his stack as an abstract data type and the stack operations as the only legal operations allowed on instances of the stack.

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 Objects–Messages, Inheritance, and Polymorphism

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. In large part, this is merely a change of emphasis; the conventional view places primary emphasis on the operation, whereas the object-oriented view emphasizes the value itself. (Do you call the *push* routine with a stack and a data value, or do you ask a *stack* to *push* a value onto itself?) If this were all there is to object-oriented programming, the technique would not be considered a major innovation. But added to message passing are powerful mechanisms for overloading names and reusing software.

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 division into parts is *repetition*. With repetition we have several artifacts of the same type working together. Repetition therefore allows us to understand a system by considering it on two levels—as a single object in isolation, and as the entire collection of values. Examples of repetition include data structures such as linked lists, mathematical induction, or recursive procedures.

Yet another powerful tool for creating complex systems out of simple parts is *composition*. Composition begins with a small set of primitive values (such as the basic types in the Java programming language) and rules for creating new values out of existing values (such as the class mechanism). The composition mechanism then allows new artifacts to be created from the existing elements. But these new artifacts then become available for further use, and thus complex systems are constructed layer by layer.

Another form of layering 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).

Another tool we will use in the understanding of complex systems is the concept of *multiple views*. Here the idea is that we can consider the same artifact in different ways at different times, using abstraction to bring out certain details one time, and to bring out other details the next.

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 31 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 world 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 professor and a student. Describe the interface governing this relationship in terms of the services that each offers the other.
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.