Chapter 1

Thinking Object Oriented

Object-oriented programming 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 be able somehow to list "experience in object-oriented programming" on their resumes. To judge from this frantic activity, object-oriented programming is being greeted with an even greater popularity than we have seen in the past heralding earlier revolutionary ideas, such as "structured programming" or "expert systems".

My intent in this first chapter 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 Object-Oriented Programming Popular?

To judge from much of the popular press, the following represent a few of the possible reasons why Object-oriented programming has, in the past decade, become so popular:

- The hope that it will quickly and easily lead to increased productivity and improved reliability (help solve the software crises).
- The desire for an easy transition from existing languages.
- The resonant similarity to techniques of thinking about problems in other domains.

Object-Oriented programming is just the latest in a long series of solutions that have been proposed to help solve the "software crises". At heart, the software crises simply means that our imaginations, and the tasks we would like to solve with the help of computers, almost always nearly 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 87]). Programming a computer is still one of the most difficult tasks ever undertaken by humankind; 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 two characters to the programmer's 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 techniques.

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 56])

This quote emphasizes the fact that the languages we speak influence directly the way in which we view the world. This is true not only for natural languages, such as the kind studied by the early 20th 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 languages influencing thought, although also an erroneous one (see [Pullum 91]) is the "fact" that Eskimo (or Inuit) lan-

guages 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.

What is important is to not over generalize 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 that I can not 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. A different language (such as Inuktitut) thus 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 fashion. But simply using an object oriented language (such as 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 human languages is even more pronounced in the realm of artificial computer languages. That is, the language in which a programmer thinks a problem will be solved will color and alter, in a basic fundamental way, the fashion in which an algorithm is developed.

An example will help illustrate this relationship between computer language and problem solution. Several years ago a student working in genetic research was faced with a task involved in the analysis of DNA sequences. The problem could be reduced to a 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 vector of values.

```
ACTCGGATCTTGCATTTCGGCAATTGGACCCTGACTTGGCCA ...
```

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
```

The student was somewhat disappointed when trial runs indicated his program would need significantly many hours to complete. He discussed his problem with a second student, who happened to be proficient in the programming language APL. The second student 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, after all, and APL was only interpreted. Therefore it was with a certain amount of incredulity that he discovered 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:

A	C	T	C	G	G	positions 1 to M
C	T	C	G	G	A	positions 2 to $M+1$
T	C	G	G	A	T	positions 3 to $M+2$
C	G	G	A	T	T	positions 4 to $M+3$
G	G	A	T	T	C	positions 5 to $M+4$
G	A	T	T	C	T	positions 6 to $M+5$
	.	.	.			
T	G	G	A	C	C	
G	G	A	C	C	C	
	.	.	.			

The APL programmer then sorted this matrix by rows. If any pattern was repeated, then two adjacent rows in the sorted matrix would have identical values.

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

It was a trivial matter to check for this condition. The reason the APL program was faster had nothing to do with the speed of APL versus FORTRAN, but was simply because 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 to indicate that APL is in any way a "better" programming language than FORTRAN, but to ask why it was that the APL programmer was naturally led to discover a better 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 operation is so easy to perform, good APL programmers tend to look for novel applications for sorting. 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 one fashion or another.

1.2.3 Church's Conjecture and the Sapir-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. It should be noted, however, a stronger conjecture, known in linguistics as the Sapir-Whorf hypothesis, goes much further and remains controversial [Pullum 91].

The Sapir-Whorf hypothesis asserts that it may be possible for an individual working in one language to imagine thoughts or utter ideas which 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 1930's and on through the 40's and 50's 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 include the notations proposed by Church [Church 36], Post [Post 36], Markov [Markov 51], Turing [Turing 36], Kleene [Kleene 36] 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 others. Often, for a pair of systems, such arguments could be made in both directions; effectively showing that the systems were identical in computation power. The sheer number of such arguments lead 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 in order to be able to simulate such a device. In the 1960's, 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 66]. (This greatly misunderstood result was the major ammunition used to "prove" that the infamous `goto` statement was unnecessary.)

If we accept Church's conjecture, then 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," since 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 fashion 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, you will recall, claimed that it was possible for ideas to be expressed in one language that could 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 a snow and categorize it with respect to different types or uses, with time and training it certainly can be learned. Similarly, object-oriented techniques do not provide any new computational power which 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 it is the case that the language will *direct* thoughts, but cannot *proscribe* thoughts.

1.3 A New Paradigm

Object-oriented programming is often referred to as a new programming *paradigm*. Other programming paradigms sometimes mentioned 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 it belongs. **2.** Any example or model. [Late Latin paradīgma, from Greek paradeigma, model, from paradeiknunai, to compare, exhibit.]

At first blush, the conjugation or declension of Latin words would seem to have little to do with computer programming languages. To understand the connection, we must note that the word was brought back into the modern vocabulary through an influential book called *The Structure of Scientific Revolutions*, authored by the historian of science Thomas Kuhn [Kuhn 70]. 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 occurred when an older paradigm was 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 79], which was titled "The Paradigms of Programming". A programming paradigm is a way of conceptualizing what it means to perform computation, and how tasks that are 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 through the history of science in general at least as far as Linnæus (1707-1778), if not even further back to the Greek philosopher 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 those 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 77].

In trying to understand exactly what is meant by the term *object-oriented programming*, it is perhaps useful to examine the idea from several alternative perspectives. The next few sections outline three different aspects of object-oriented programming; each illustrates a particular reason why 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 I wish to send some flowers to my grandmother (who is named Elsie) for her birthday. She lives in a city many miles away, so the possibility of my picking the flowers and carrying them myself to her door is out of the question. Nevertheless, sending her the flowers is a task easy enough to do; I merely go down to my local florist (who happens to be named Flo), describe the kinds and numbers of flowers I want sent, and my grandmother's address, and I can be assured the flowers will be delivered expediently and automatically.

1.4.1 Agents, Responsibility, Messages and Methods

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

If I investigated, however, I might discover that Flo delivers a slightly different message to another florist in my grandmother's city. That florist, in turn, makes the arrangement

and passes it, along with yet another message, to a delivery person, and so on. In this manner my request is finally satisfied by a sequence of requests from one agent to another.

So our first 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 agent 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 can also observe is implicit in message passing. That principle is, if there is a task to perform, the first thought of the client is to find somebody else whom it can ask to do the work. It is interesting to note the degree to which this second reaction will often become atrophied in many programmers with extensive experience in using conventional techniques. Frequently, a difficult hurdle to overcome is the idea in the programmers mind that he or she must write everything themselves, 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 use an already developed component.

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.

There are two important distinctions. The first is that in a message there is a designated *receiver* for that message; the receiver is some agent to whom the message is sent. In a procedure call, there is no designated receiver. (Although we could adapt a convention of, for example, always calling the first argument to a procedure the receiver, something that is very close to how receivers are actually implemented.)

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. I can give exactly the same message to my wife Beth, for example, and she will understand the message and a satisfactory outcome will be produced (that is, flowers will be delivered to my grandmother). However, the method Beth uses to satisfy the request (in all likelihood, simply passing the request on to Flo), will be different from that performed by Flo in response to the same request. If I ask Ken, my dentist, to send flowers to my grandmother, I probably would be making an error, since Ken may not have a method for solving that problem. If he understood the request at all, he would 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 – that is, 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.

1.4.2 Responsibilities

A fundamental concept in object-oriented programming is to describe behavior in terms of *responsibilities*. My request for action indicates only the desired outcome (flowers for my grandmother). The florist is free to pursue any technique that achieves the desired objective, and is not hampered by interference on my part.

By discussing a problem in terms of responsibilities we increase the level of abstraction. This permits greater *independence* between agents, a critically important factor in solving complex problems. In Chapter 2 we will investigate in more detail how we can use an emphasis on responsibility to drive the software design process. The entire collection of responsibilities associated with an object is often described by the term *protocol*.

The difference between viewing software in traditional structured terms and viewing software in 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"

1.4.3 Classes and Instances

Although I have only dealt with Flo a few times in the past, I have a rough idea of the behavior I can expect when I go into her shop and present her with my request. I am able to make certain assumptions because I have some information about florists in general, and I expect that Flo, 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 second 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.

One current problem in the object-oriented community is the proliferation of different terms for similar ideas. Thus, a class is known in Object Pascal as an *object type*, and a

ancestor class (which we will describe shortly), is known as a *superclass* or *parent class*, and so on. The glossary at the end of this book should be of some help with unusual terms. We will use the convention, common in object-oriented languages, of always designating classes by a name beginning with an uppercase letter. Although commonly used, this convention is not enforced by most language systems.

1.4.4 Class Hierarchies – Inheritance

There is more information I have about Flo – not necessarily because she is a florist, but because she is a shopkeeper. I know, for example, that I probably will be asked for money as part of the transaction, and in return for payment I will be given a receipt. These actions are true of grocers, of stationers, and of other shopkeepers. Since the category **Florist** is a more specialized form of the category **Shopkeeper**, any knowledge I have of **Shopkeepers** is also true of **Florists**, and hence of Flo.

One way to think about how I have organized my knowledge of Flo is in terms of a hierarchy of categories (see Figure 1.1). Flo is a **Florist**, but **Florist** is a specialized form of **Shopkeeper**. Furthermore, a **Shopkeeper** is also a **Human**; so I know, for example, that Flo is probably bipedal. A **Human** is a **Mammal** (therefore they nurse their young), and a **Mammal** is an **Animal** (therefore it breathes oxygen), and an **Animal** is a **Material Object** (therefore it has mass and weight). Thus, there is quite a lot of knowledge I have that is applicable to Flo that is not directly associated with her, or even with her category **Florist**.

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

There is an alternative graphical technique that is 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, listed near the bottom. Figure 1.2 shows this class hierarchy for Flo. This hierarchy also includes Beth, my dog Flash, Phyl the platypus who lives at the zoo, and the flowers themselves that I am sending to my grandmother.

Information that I possess about Flo because she is an instance of class **Human** is also applicable to my wife Beth, for example. Information that I have about her because she is a **Mammal** is applicable to Flash as well. Information about all members of **Material Object** is equally applicable to Flo and to her flowers. We capture this in the idea of inheritance:

> Classes can be organized into a hierarchical *inheritance* structure. A *child class* (or *subclass*) will inherit attributes from a *parent class* higher in the tree. An *abstract parent class* is a class (such as **Mammal**) that is used only to create subclasses, for which there are no direct instances.

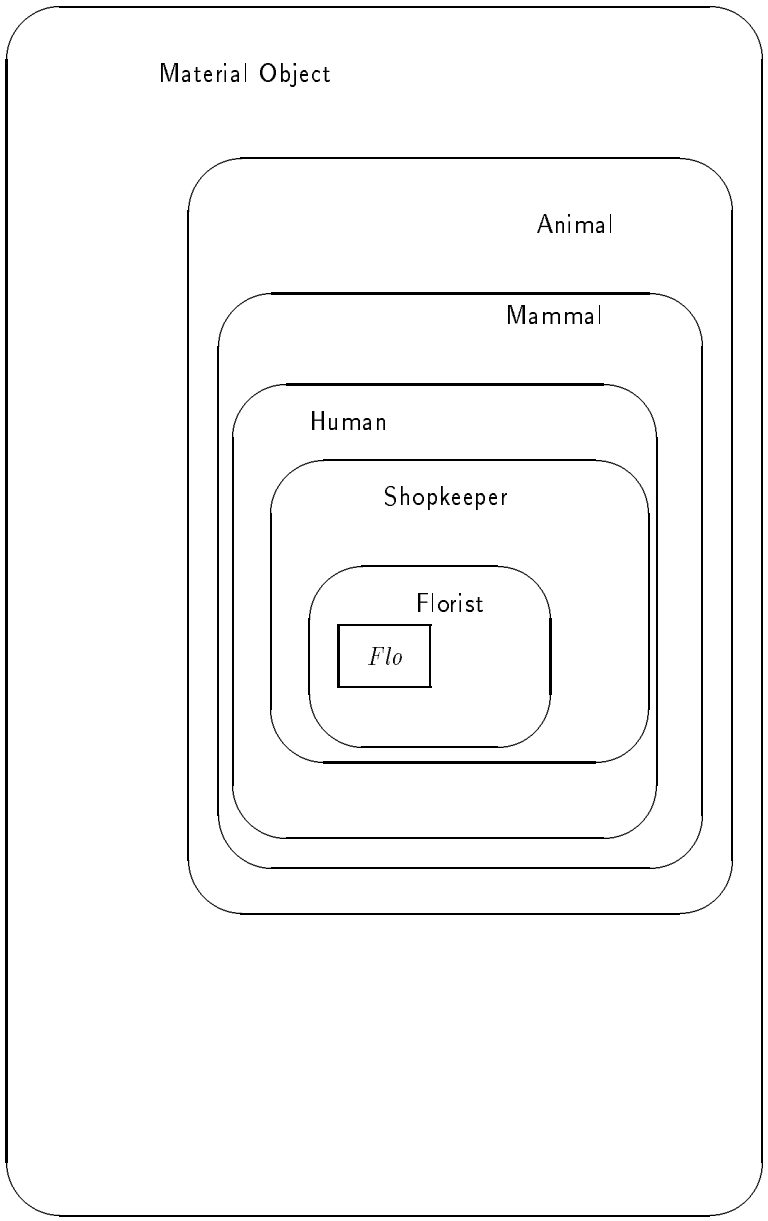

Figure 1.1: The categories surrounding Flo

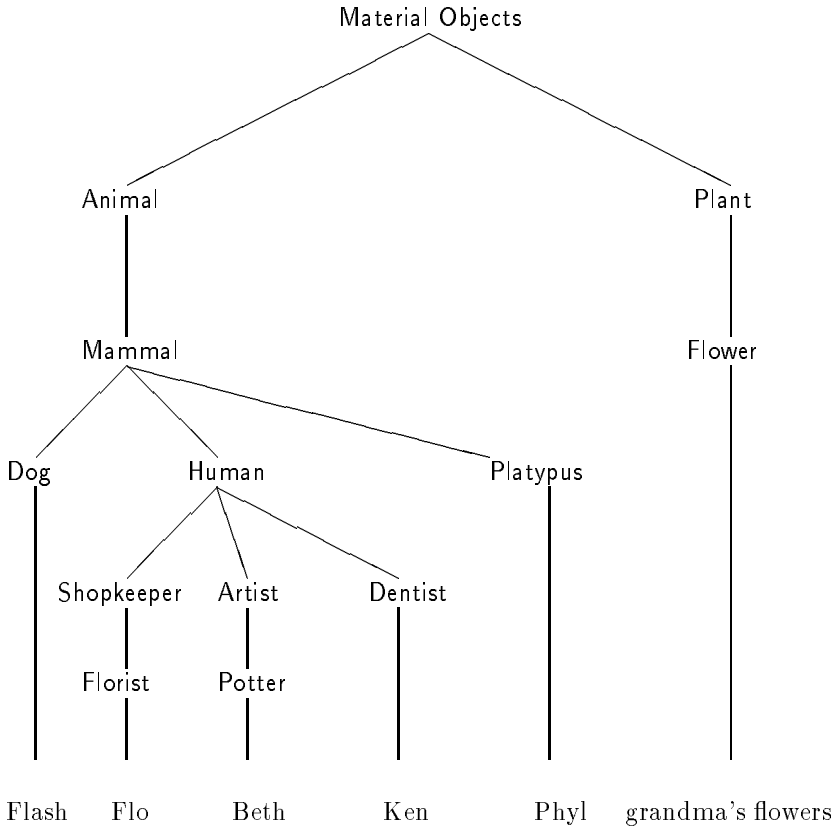

Figure 1.2: A class hierarchy for various material objects

1.4.5 Method Binding, Overriding and Exceptions

Phyl the platypus presents a problem for our simple organizing structure. I know that mammals give birth to live children, for example, and Phyl is certainly a Mammal, and yet Phyl (or rather his mate, Phyllis) continues to lay 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 to find 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 the compiler cannot determine which method will be invoked at run time, in many object-oriented languages 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. We will discuss the mechanism for overriding in various computer languages in Chapter 11.

That my wife Beth and my florist Flo will respond to my message by performing different methods is an example of one form of *polymorphism*. We will discuss this important part of object-oriented programming in Chapter 14. As we explained, that I do not, and need not, know exactly what method Flo will use to honor my message is an example of *information hiding*, which we will discuss in Chapter 17.

1.4.6 Summary of Object-Oriented Concepts

Alan Kay, considered by some to be the father of object-oriented programming, has identified the following characteristics as fundamental to OOP: [Kay 93]

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 descendent in this tree structure.

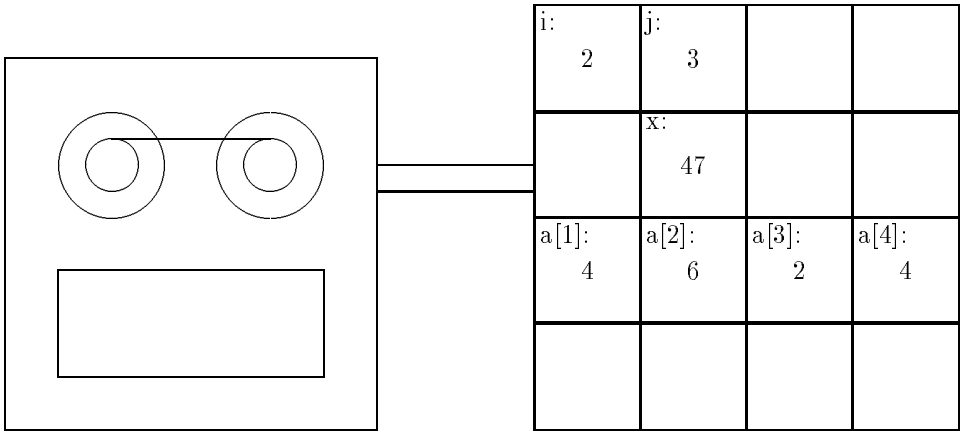

Figure 1.3: Visualization of Imperative Programming

1.5 Computation as Simulation

The view of programming that is represented by the example of sending flowers to my grandmother 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 1.3). 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 mentioned memory addresses or variables or assignments or any of the conventional programming terms. Instead, we spoke of objects, messages, and responsibility for some action. In Dan Ingalls memoriable 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 81].

Another author has described object-oriented programming as "animistic;" a process

of creating a host of helpers that forms a community and assists the programmer in the solution of a problem [Actor 87].

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 77].

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 computing is thought of in terms of pigeon-holes, mailboxes, or slots containing values, there is little in the average programmer's background to provide an intuitive insight into how problems should be structured.

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

It is possibly this feature, more than any other, that is responsible for the frequently observed phenomenon 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 Regress

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 other than 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 the introduction of "primitive" or "native" operations that are provided by the underlying system.

1.6 Coping with Complexity

When computing was in its infancy, most programs were written in assembly language, by a single individual, and would not be considered large by today's standards. Even so, as programs became more complex, programmers found that they had a difficult time remembering all the information they needed to know in order to develop or debug their software. Which values were contained in what registers? Did a new identifier name conflict with any other previously defined name? What variables needed to be initialized before control could be transferred to another section of code?

The introduction of higher-level languages, such as Fortran, Cobol and Algol, solved some difficulties (such as automatic management of local variables, and implicit matching of arguments to parameters), while simultaneously raising people's expectations of what a computer could do in a manner that only introduced yet new problems. As programmers attempted to solve ever more complex problems using a computer, tasks exceeding in size the grasp of even the best programmers became the norm. Thus, teams of programmers working together to undertake major programming efforts became commonplace.

1.6.1 The Nonlinear Behavior of Complexity

As programming projects became larger, an interesting phenomenon was observed. A task that would take one programmer 2 months to perform could not be accomplished by two programmers working for 1 month. In Fred Brooks's memorable phrase, "the bearing of a child takes nine months, no matter how many women are assigned to the task" [Brooks 75].

The reason for this nonlinear behavior was complexity – in particular, the interconnections between software components were complicated, and large amounts of information had to be communicated among various members of the programming team. Brooks further said:

> "Since software construction is inherently a systems effort – an exercise in complex interrelationships – communication effort is great, and it quickly dominates the decrease in individual task time brought about by partitioning. Adding more men then lengthens, not shortens, the schedule" [Brooks 75].

What brings about this complexity? It is not simply the sheer size of the tasks undertaken, because size by itself would not be a hindrance to partitioning each into several pieces. The unique feature of software systems developed using conventional techniques that

makes them among the most complex systems developed by people is their high degree of interconnectedness. Interconnectedness means the dependence of one portion of code on another section of code.

Consider that any portion of a software system must be performing an essential task, or it would not be there. Now, if this task is useful to the other parts of the program, there must be some communication of information either into or out of the component under consideration. Because of this, a complete understanding of what is going on requires a knowledge both of the portion of code we are considering and the code that uses it. In short, an individual section of code cannot be understood in isolation.

1.6.2 Abstraction Mechanisms

Programmers have had to deal with the problem of complexity for a long time in the history of computer science. To understand more fully the importance of object-oriented techniques, we should review the variety of mechanisms programmers have used to control complexity. Chief among these is *abstraction*, the ability to encapsulate and isolate design and execution information. In one sense, object-oriented techniques are not revolutionary at all, but can be seen to be a natural outcome of a long historical progression from procedures, to modules, to abstract data types and finally to objects.

Procedures

Procedures and functions were one of the first abstraction mechanisms to be widely used in programming languages. Procedures allowed tasks that were executed repeatedly, or were executed with only slight variations, to be collected in one place and reused, rather than the code 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 were 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 his or her work – say, a set of four routines: `init`, `push`, `pop`, and `top`. She then selects some suitable implementation technique. Here, there are many choices, 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 1.4.

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

void init()
{
    datatop = 0;
}

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

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

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

Figure 1.4: Failure of procedures in information hiding

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 FORTRAN, or in C prior to the introduction of the `static` modifier), then the stack data must be maintained in global variables. But if the variables are global, then there is no way to limit the accessibility or visibility of these names. For example, if the stack is represented in an array named `datastack`, this fact must be made known to all the other programmers, since they may want to create variables using the same name and should be discouraged from doing so. This is true even though these data values are important to only 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.

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 be tempted to hope that this ability would solve 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;
```

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, whereas 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 (in [Parnas 72]) 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 with *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 known as *information hiding*.

Modules solve some, but not all, of the problems of software development. For example, modules 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 she has developed a new type of numeric abstraction, called **Complex**. She has defined the arithmetic operations for complex numbers - addition, subtraction, multiplication, and so on. She has defined routines to convert from conventional numbers to complex. There is just one small problem: you can manipulate only one complex number.

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 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.

Abstract Data Types

An *abstract data type* is a programmer-defined data type that can be manipulated in a manner similar to 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 her stack as an abstract data type, and the stack operations as the only legal operations that are allowed to be performed 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 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:

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

Modules, as we have defined them, serve only as an information-hiding mechanism, and thus directly address only abilities 2 and 3, although the others can be accommodated using

appropriate programming techniques. *Packages*, found in languages such as CLU or 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. Although it is true that an object definition is an abstract data type, 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.

Objects – Messages, Inheritance and Polymorphism

The techniques of object-oriented programming add several important new ideas to the concept of the abstract data type. Foremost among these is the idea of *message passing*. Activity is initiated by a *request* being made to a specific object, not by a function using specific data being invoked. In large part, this is merely a change of emphasis; the conventional view places the primary importance on the operation, whereas the object-oriented view places primary importance on 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 on to 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 elicits will depend upon the object receiving the message. 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 each individual data type. 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.

1.7 Reusable Software

People have asked for decades why the construction of software could not mirror more closely the construction of other material objects. When we construct a building, a car, or an electronic device, for example, we typically piece together a number of off-the-shelf

components, rather than fabricating each new element from scratch. Could not software be constructed in the same fashion?

In the past, software reusability has been a much sought-after and seldom-achieved goal. A major reason for this is the tight interconnectedness of most software that has been constructed in a conventional manner. As we discussed in an earlier section, it is difficult to extract from one project elements of software that can be easily used in an unrelated project, because each portion of code typically has interdependencies with all other portions of code. These interdependencies may be a result of data definitions, or may be functional dependencies.

For example, organizing records into a table and performing indexed lookup operations on this table are perhaps some of the most common operations in programming. Yet table-lookup routines are almost always written over again for each new application. Why? Because, in conventional languages, the record format for the elements is tightly bound with the more general code for insertion and lookup. It is difficult to write code that can work for arbitrary data, for any record type.

Object-oriented techniques provide a mechanism for cleanly separating the essential information (insertion and retrieval) from the inconsequential information (the format for particular records). Thus, using object-oriented techniques, we can construct large reusable software components. Many such packages of commercial software components are now available, and the developing of such reusable components is a rapidly increasing part of the software industry.

1.8 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 largely autonomous agents, called *objects*. Each object is responsible for specific tasks. It is by the interaction of objects that computation proceeds. In a certain sense, therefore, programming is nothing more or less than the simulation of a model universe.

- An object is an encapsulation of *state* (data values) and *behavior* (operations). Thus, an object is in many ways similar to a module, or an abstract data type.

- The behavior of objects is dictated by the object's *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 performed) 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.

- 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 facts, this assertion is only somewhat supportable. Kay himself, in [Kay 93] traces much of the influence for his development of Smalltalk to the earlier computer programming Simula developed in Scandianavia in the early 1960's [Dahl 66, Kirkerud 89]. A more accurate history would say 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 has a whole had they not been rediscovered by Kay in the creation of the Smalltalk programming language. A widely read issue of *Byte* magazine in 1981 did much to popularize the concepts developed by Kay and his team at Xerox PARC.

The term "software crises" seems to have been coined by Doug McIlroy at a 1968 NATO conference on software engineering. It is curious to note 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 did in 1968. See, for example, the article "Software's Chronic Crisis" in the September 1994 issue of *Scientific American* [Gibbs 94].

To some extent, the software crises may be largely illusory. For example, tasks which were 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. This would seem to indicate that, indeed, the field of software development has 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", by Benjamin Lee Whorf. This article has been reprinted in the book *Language, Thought and Reality* [Whorf 56]. This book contains several interesting papers discussing the relationships between language and our habitual thinking processes. I urge any serious student of computer languages to read several of these essays, since some of them have surprising relevance to artificial languages.

Another interesting book along similar lines is *The Alphabet Effect*, by Robert Logan [Logan 86], which explains in terms of language why logic and science developed in the West, while for centuries China in the East has superior technology. In a more contemporary investigation of the effect of natural language on computer science, J. Marshall Unger [Unger 87] 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 has been debunked by Geoffrey Pullum in the title article in a book of essays in linguistics [Pullum 91]. In an article in the *Atlantic Monthly* ("In Praise of Snow", January 1995), Cullen Murphy points out that the vocabulary used to discuss snow among those 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.

In any case, the point is largely irrelevant to our discussion. It is certainly true that groups of individuals with common interests will tend to develop their own specialized vocabulary; and having done so, the vocabulary itself will tend 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 will help direct the programmers thought along lines that may not have been obvious without the OOP terminology.

My history is slightly imprecise with regards to church's conjecture and Turing machines. Church actually conjectured about partial functions [Church 36]. These were later shown to be equivalent to computations performed using Turing machines [Turing 36]. Kleene described the conjecture in the form we have here, also giving it the name by which it has become known. Rogers [Rogers 67] gives a good summary of the arguments for the equivalence of various computational models.

It was the Swedish botanist Carolus Linnæus, you will recall, who developed the idea of Genus, Species, etc. This system is the prototypical hierarchical organization scheme exhibiting inheritance, since abstract classifications describe features that are largely common to all subclassifications. Most inheritance hierarchies follow closely the model of Linnæus.

The criticism of procedures as an abstraction technique, because they fail to provide an adequate mechanism for information hiding, was first developed by William Wulf and Mary Shaw [Wulf 73] in an analysis of many of the problems surrounding the use of global variables. These arguments were later expanded upon by David Hanson [Hanson 81].

Like most terms that have found their way into the popular jargon, *object-oriented* is used with greater regularity than it is defined. Thus, the question "what is object-oriented programming?" is surprisingly difficult to answer. Bjarne Stroustrup has quipped [Stroustrup 88]

that many arguments appear to boil down to the following syllogism:

- **X** is good
- Object-Oriented is good
- *Ergo,* **X** is object-oriented

Roger King has argued (in [Kim 89]), 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 what properties a programming language must possess to be called *object-oriented.* See, for example, the analysis by Josephine Micallef [Micallef 88], or Peter Wegner [Wegner 86]. 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 90] - 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 divisively narrow, we should embrace any and all means that show promise of leading to a new software "Industrial Revolution." The book by Cox [Cox 86], although written early in the development of object-oriented programming and thus now somewhat dated in details, is nevertheless one of the most readable manifestos of the object-oriented movement.

Exercises

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

2. Look up the definition of the word *paradigm* in at least three different 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 the solution to the problem 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 computer language would direct the programmer to one type of solution, whereas a different language would encourage an alternative type of solution.

5. If you are familiar with two or more distinct natural languages, give an example of a situation that illustrates how one language would direct the speaker in a certain direction, whereas the other language would encourage a different line of thought.

6. Argue either for or against the position that computing is basically simulation. (You may want to examine the article by Alan Kay [Kay 77].)