Chapter 1

Thinking Object-Oriented

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

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

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

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

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

1.1 Why Is OOP Popular?

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

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

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

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

1.2 Language and Thought

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

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

1.2.1 Eskimos and Snow

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

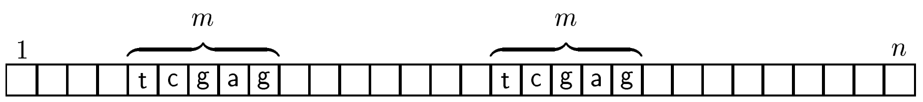
What is important is to not overgeneralize the conclusion we can draw from this simple observation. It is not that the Eskimo eye is in any significant respect different from my own, or that Eskimos can see things I cannot perceive. With time and training I could do just as well at differentiating types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

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

1.2.2 An Example from Computer Languages

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

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



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

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

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

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

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

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

$$\begin{array}{cccccc} & . & . & . & & \\ T & G & G & A & C & C \\ T & G & G & A & C & C \\ & . & . & . & & \end{array}$$

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

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

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

1.2.3 Church's Conjecture and the Whorf Hypothesis *

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

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

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

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

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

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

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

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

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

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

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

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

1.3 A New Paradigm

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

$$\begin{tikzcd}
\text{kingdom} & & & \text{Animalia} \arrow[dl] \arrow[drr] & & & \\
\text{phylum} & & \text{Mollusca} \arrow[dl] & & & \text{Chordata} \arrow[dl] \arrow[dr] & \\
\text{class} & \text{Gastropoda} \arrow[d] & & & \text{Reptilla} \arrow[d] & & \text{Mammalia} \arrow[d] \\
\text{order} & \text{Mesogastropoda} \arrow[d] & & & \text{Predentata} \arrow[d] & & \text{Primates} \arrow[d] \\
\text{family} & \text{Cypraeidae} \arrow[d] & & & \text{Ceratopsidae} \arrow[d] & & \text{Hominidae} \arrow[d] \\
\text{genus} & \text{Cypraea} \arrow[d] & & & \text{Triceratops} \arrow[d] & & \text{Homo} \arrow[d] \\
\text{species} & \text{tigris} & & & \text{horridus} & & \text{sapiens}
\end{tikzcd}$$

Figure 1.1: The Linnæn Inheritance Hierarchy

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

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

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

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

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

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

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

1.4 A Way of Viewing the World

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

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

1.4.1 Agents and Communities

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

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

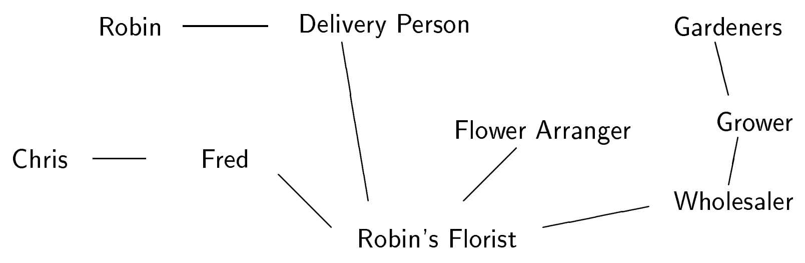


Figure 1.2: The community of agents helping delivery flowers

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

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

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

1.4.2 Messages and Methods

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

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

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

Messages versus Procedure Calls

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

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

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

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

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

1.4.3 Responsibilities

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

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

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

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

1.4.4 Classes and Instances

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

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

1.4.5 Class Hierarchies–Inheritance

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

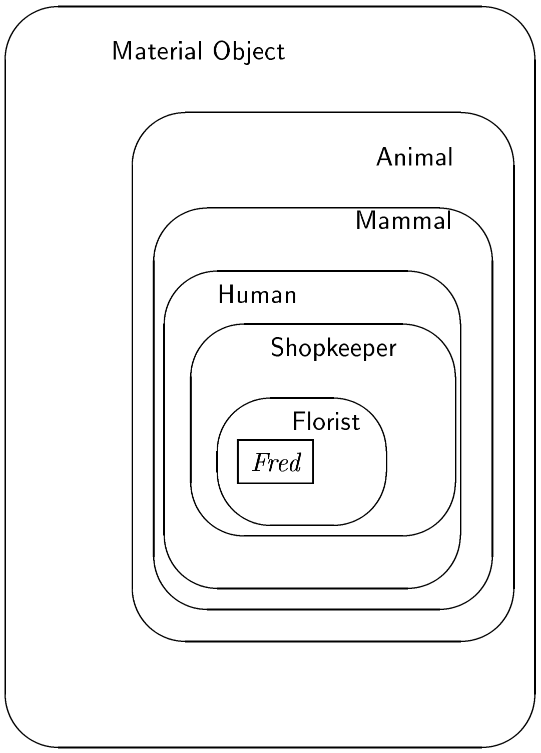


Figure 1.3: – The categories surrounding Fred.

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

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

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

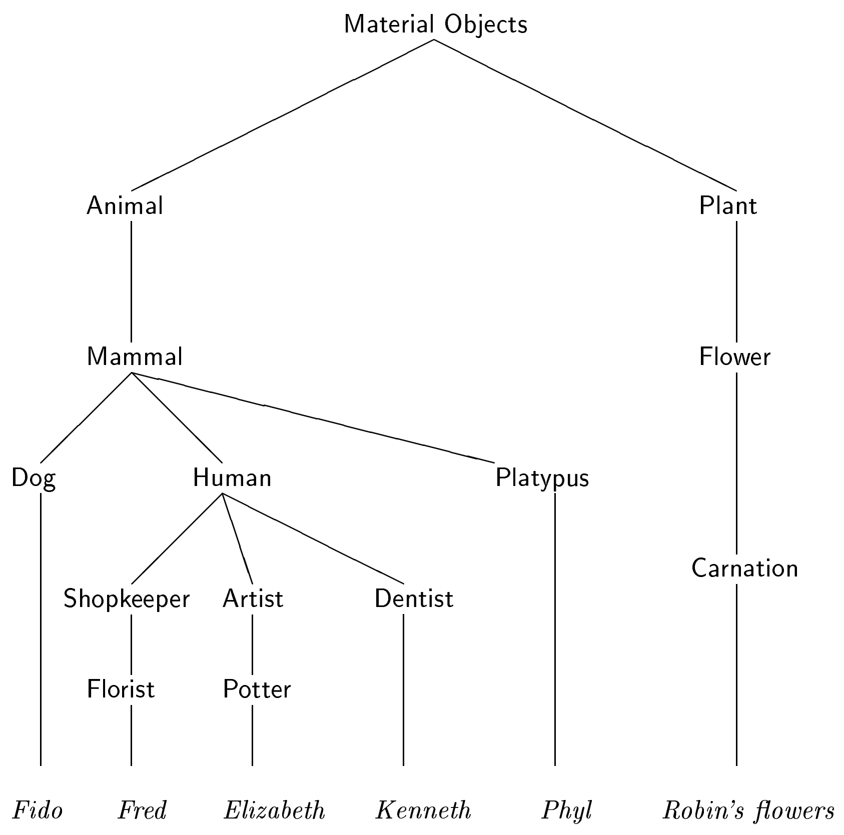


Figure 1.4: – A class hierarchy for various material objects.

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

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

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

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

1.4.6 Method Binding and Overriding

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

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

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

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

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

1.4.7 Summary of Object-Oriented Concepts

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

1. Everything is an *object*.
2. Computation is performed by objects communicating with each other, requesting that other objects perform actions. Objects communicate by sending and receiving *messages*. A message is a request for action bundled with whatever arguments may be necessary to complete the task.
3. Each object has its own *memory*, which consists of other objects.

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

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

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

1.5 Computation as Simulation *

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

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

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

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

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

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

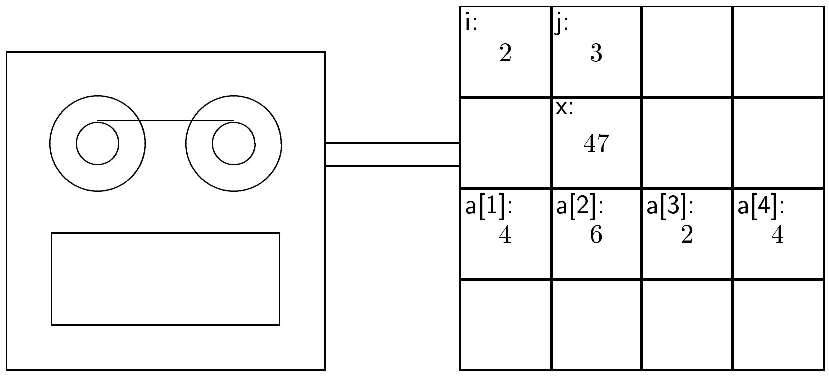


Figure 1.5: – Visualization of imperative programming.

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

1.5.1 The Power of Metaphor

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

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

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

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

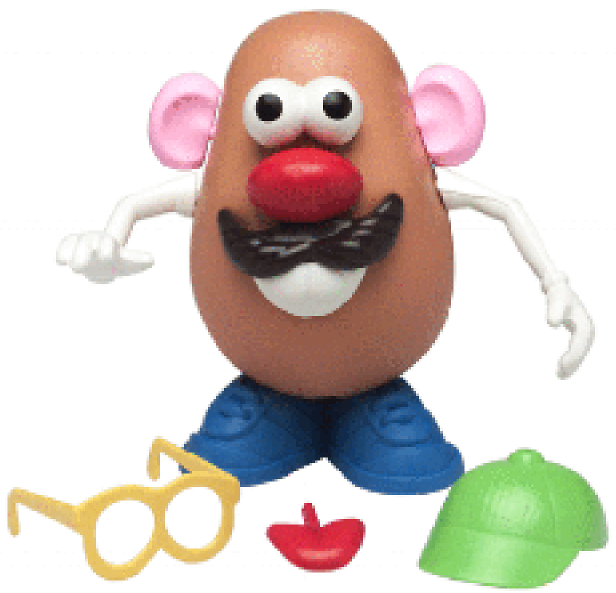


Figure 1.6: Mr. Potato Head, an Object-Oriented Toy

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

1.5.2 Avoiding Infinite Regression

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

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

1.6 A Brief History *

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

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

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

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

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

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

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

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

Chapter Summary

- Object-oriented programming is not simply a few new features added to programming languages. Rather, it is a new way of *thinking* about the process of decomposing problems and developing programming solutions.

- Object-oriented programming views a program as a collection of loosely connected agents, termed *objects*. Each object is responsible for specific tasks. It is by the interaction of objects that computation proceeds. In a certain sense, therefore, programming is nothing more or less than the simulation of a model universe.

- An object is an encapsulation of *state* (data values) and *behavior* (operations). Thus, an object is in many ways similar to special purpose computer.

- The behavior of objects is dictated by the object *class*. Every object is an instance of some class. All instances of the same class will behave in a similar fashion (that is, invoke the same method) in response to a similar request.

- An object will exhibit its behavior by invoking a method (similar to executing a procedure) in response to a message. The interpretation of the message (that is, the specific method used) is decided by the object and may differ from one class of objects to another.

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

Further Reading

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

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

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

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

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

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

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

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

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

Like most terms that have found their way into the popular jargon, *object-oriented* is used more often than it is defined. Thus, the question What is object-oriented programming? is surprisingly difficult to answer. Bjarne Stroustrup has quipped that many arguments appear to boil down to the following syllogism:

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

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

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

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

Self Study Questions

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

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

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

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

Exercises

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

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

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

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

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

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