Chapter 2

# The Object Model

Object-oriented technology is built on a sound engineering foundation, whose elements we collectively call the *object model of development* or simply the *object model*. The object model encompasses the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence. By themselves, none of these principles are new. What is important about the object model is that these elements are brought together in a synergistic way.

Let there be no doubt that object-oriented analysis and design is fundamentally different than traditional structured design approaches: It requires a different way of thinking about decomposition, and it produces software architectures that are largely outside the realm of the structured design culture.

## 2.1 The Evolution of the Object Model

Object-oriented development did not spontaneously generate itself from the ashes of the uncounted failed software projects that used earlier technologies. It is not a radical departure from earlier approaches. Indeed, it is founded in the best ideas from prior technologies. In this section we will examine the evolution of the tools of our profession to help us understand the foundation and emergence of object-oriented technology.

As we look back on the relatively brief yet colorful history of software engineering, we cannot help but notice two sweeping trends:

1. The shift in focus from programming-in-the-small to programming-in-the-large
2. The evolution of high-order programming languages

Most new industrial-strength software systems are larger and more complex than their predecessors were even just a few years ago. This growth in complexity has prompted a significant amount of useful applied research in software engineering, particularly with regard to decomposition, abstraction, and hierarchy. The development of more expressive programming languages has complemented these advances.

## The Generations of Programming Languages

Wegner has classified some of the more popular high-order programming languages in generations arranged according to the language features they first introduced [2]. (By no means is this an exhaustive list of all programming languages.)

- First-generation languages (1954–1958)

  | | |
  |---|---|
  | FORTRAN I | Mathematical expressions |
  | ALGOL 58 | Mathematical expressions |
  | Flowmatic | Mathematical expressions |
  | IPL V | Mathematical expressions |

- Second-generation languages (1959–1961)

  | | |
  |---|---|
  | FORTRAN II | Subroutines, separate compilation |
  | ALGOL 60 | Block structure, data types |
  | COBOL | Data description, file handling |
  | Lisp | List processing, pointers, garbage collection |

- Third-generation languages (1962–1970)

  | | |
  |---|---|
  | PL/1 | FORTRAN + ALGOL + COBOL |
  | ALGOL 68 | Rigorous successor to ALGOL 60 |
  | Pascal | Simple successor to ALGOL 60 |
  | Simula | Classes, data abstraction |

- The generation gap (1970–1980)

  Many different languages were invented, but few endured. However, the following are worth noting:

  | | |
  |---|---|
  | C | Efficient; small executables |
  | FORTRAN 77 | ANSI standardization |

Let's expand on Wegner's categories.

- Object-orientation boom (1980–1990, but few languages survive)

| | |
|---|---|
| Smalltalk 80 | Pure object-oriented language |
| C++ | Derived from C and Simula |
| Ada83 | Strong typing; heavy Pascal influence |
| Eiffel | Derived from Ada and Simula |

- Emergence of frameworks (1990–today)

  Much language activity, revisions, and standardization have occurred, leading to programming frameworks.

| | |
|---|---|
| Visual Basic | Eased development of the graphical user interface (GUI) for Windows applications |
| Java | Successor to Oak; designed for portability |
| Python | Object-oriented scripting language |
| J2EE | Java-based framework for enterprise computing |
| .NET | Microsoft's object-based framework |
| Visual C# | Java competitor for the Microsoft .NET Framework |
| Visual Basic .NET | Visual Basic for the Microsoft .NET Framework |

In successive generations, the kind of abstraction mechanism each language supported changed. First-generation languages were used primarily for scientific and engineering applications, and the vocabulary of this problem domain was almost entirely mathematics. Languages such as FORTRAN I were thus developed to allow the programmer to write mathematical formulas, thereby freeing the programmer from some of the intricacies of assembly or machine language. This first generation of high-order programming languages therefore represented a step closer to the problem space and a step further away from the underlying machine.

Among second-generation languages, the emphasis was on algorithmic abstractions. By this time, machines were becoming more and more powerful, and the economics of the computer industry meant that more kinds of problems could be automated, especially for business applications. Now, the focus was largely on telling the machine what to do: read these personnel records first, sort them next, and then print this report. Again, this new generation of high-order programming languages moved us a step closer to the problem space and further away from the underlying machine.

By the late 1960s, especially with the advent of transistors and then integrated circuit technology, the cost of computer hardware had dropped dramatically, yet processing capacity had grown almost exponentially. Larger problems could now be solved, but these demanded the manipulation of more kinds of data. Thus, third-generation languages such as ALGOL 60 and, later, Pascal evolved with support

for data abstraction. Now a programmer could describe the meaning of related kinds of data (their type) and let the programming language enforce these design decisions. This generation of high-order programming languages again moved our software a step closer to the problem domain and further away from the underlying machine.

The 1970s provided us with a frenzy of activity in programming language research, resulting in the creation of literally a couple of thousand different programming languages and dialects. To a large extent, the drive to write larger and larger programs highlighted the inadequacies of earlier languages; thus, many new language mechanisms were developed to address these limitations. Few of these languages survived (have you seen a recent textbook on the languages Fred, Chaos, or Tranquil?); however, many of the concepts that they introduced found their way into successors of earlier languages.

What is of the greatest interest to us is the class of languages we call *object-based* and *object-oriented*. Object-based and object-oriented programming languages best support the object-oriented decomposition of software. The number of these languages (and the number of "objectified" variants of existing languages) boomed in the 1980s and early 1990s. Since 1990 a few languages have emerged as mainstream OO languages with the backing of commercial programming tool vendors (e.g., Java, C++). The emergence of programming frameworks (e.g., J2EE, .NET), which provide a tremendous amount of support to the programmer by offering components and services that simplify the common and often mundane programming tasks, has greatly boosted productivity and demonstrated the elusive promise of component reuse.

## The Topology of First- and Early Second-Generation Programming Languages

Let's consider the structure of each generation of programming languages. In Figure 2–1, we see the topology of most first- and early second-generation programming languages. By *topology*, we mean the basic physical building blocks of the language and how those parts can be connected. In this figure, we see that for languages such as FORTRAN and COBOL, the basic physical building block of all applications is the subprogram (or the paragraph, for those who speak COBOL).

Applications written in these languages exhibit a relatively flat physical structure, consisting only of global data and subprograms. The arrows in this figure indicate dependencies of the subprograms on various data. During design, one can logically separate different kinds of data from one another, but there is little in these languages that can enforce these design decisions. An error in one part of a program can have a devastating ripple effect across the rest of the system because the global data structures are exposed for all subprograms to see.

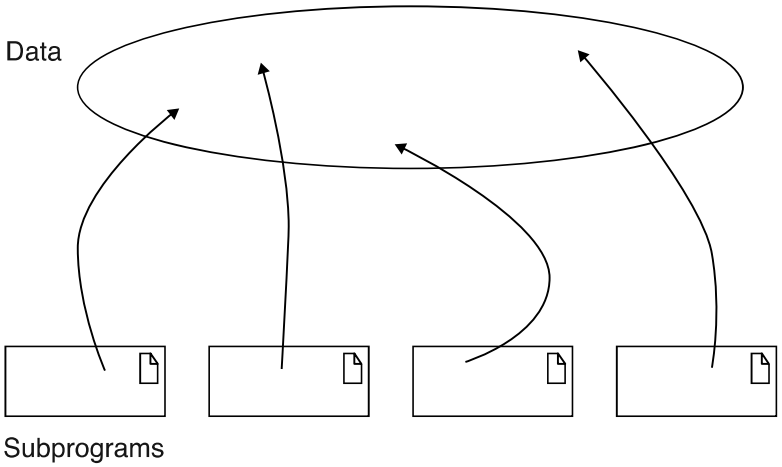



**Figure 2–1** The Topology of First- and Early Second-Generation Programming Languages

When modifications are made to a large system, it is difficult to maintain the integrity of the original design. Often, entropy sets in: After even a short period of maintenance, a program written in one of these languages usually contains a tremendous amount of cross-coupling among subprograms, implied meanings of data, and twisted flows of control, thus threatening the reliability of the entire system and certainly reducing the overall clarity of the solution.

## The Topology of Late Second- and Early Third-Generation Programming Languages

By the mid-1960s, programs were finally being recognized as important intermediate points between the problem and the computer [3]. "The first software abstraction, now called the 'procedural' abstraction, grew directly out of this pragmatic view of software. . . . Subprograms were invented prior to 1950, but were not fully appreciated as abstractions at the time. . . . Instead, they were originally seen as labor-saving devices. . . . Very quickly though, subprograms were appreciated as a way to abstract program functions" [4].

The realization that subprograms could serve as an abstraction mechanism had three important consequences. First, languages were invented that supported a variety of parameter-passing mechanisms. Second, the foundations of structured programming were laid, manifesting themselves in language support for the nesting of subprograms and the development of theories regarding control structures and the scope and visibility of declarations. Third, structured design methods emerged, offering guidance to designers trying to build large systems using subprograms as basic physical building blocks. Thus, it is not surprising, as Figure 2–2 shows, that the topology of late second- and early third-generation languages is largely a variation on the theme of earlier generations. This topology addresses

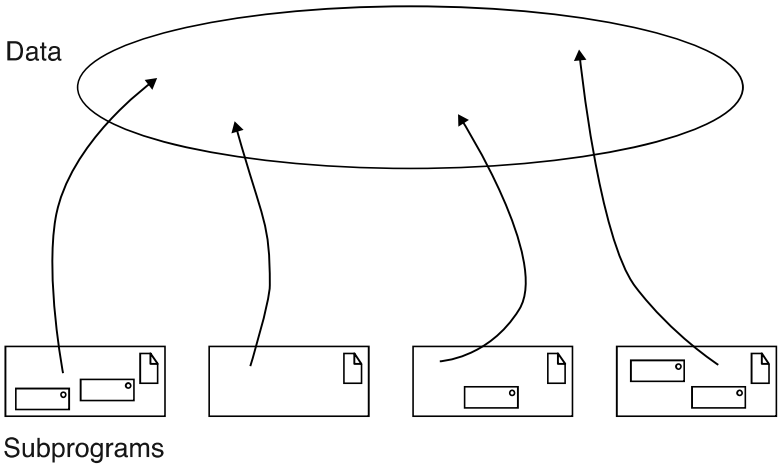



**Figure 2–2** The Topology of Late Second- and Early Third-Generation Programming Languages

some of the inadequacies of earlier languages, namely, the need to have greater control over algorithmic abstractions, but it still fails to address the problems of programming-in-the-large and data design.

## The Topology of Late Third-Generation Programming Languages

Starting with FORTRAN II, and appearing in most late third-generation program languages, another important structuring mechanism evolved to address the growing issues of programming-in-the-large. Larger programming projects meant larger development teams, and thus the need to develop different parts of the same program independently. The answer to this need was the separately compiled module, which in its early conception was little more than an arbitrary container for data and subprograms, as Figure 2–3 shows. Modules were rarely recognized as an important abstraction mechanism; in practice they were used simply to group subprograms that were most likely to change together.

Most languages of this generation, while supporting some sort of modular structure, had few rules that required semantic consistency among module interfaces. A developer writing a subprogram for one module might assume that it would be called with three different parameters: a floating-point number, an array of ten elements, and an integer representing a Boolean flag. In another module, a call to this subprogram might incorrectly use actual parameters that violated these assumptions: an integer, an array of five elements, and a negative number. Similarly, one module might use a block of common data that it assumed as its own, and another module might violate these assumptions by directly manipulating this

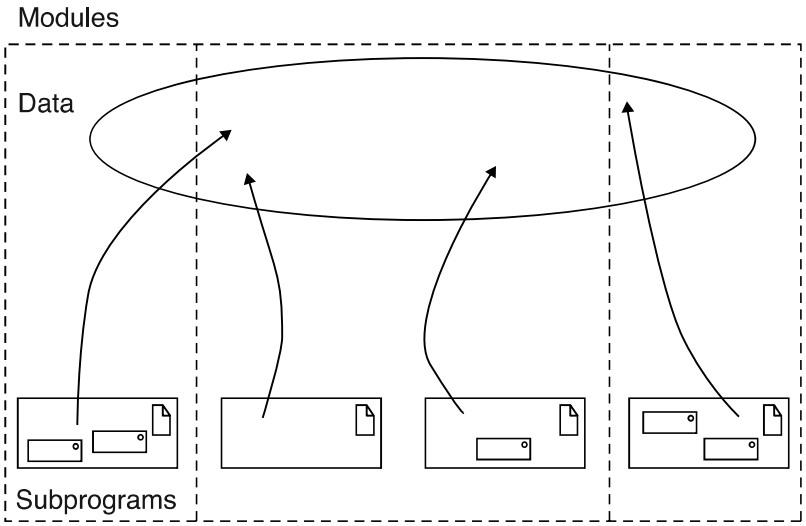



**Figure 2–3** The Topology of Late Third-Generation Programming Languages

data. Unfortunately, because most of these languages had dismal support for data abstraction and strong typing, such errors could be detected only during execution of the program.

## The Topology of Object-Based and Object-Oriented Programming Languages

Data abstraction is important to mastering complexity. "The nature of abstractions that may be achieved through the use of procedures is well suited to the description of abstract operations, but is not particularly well suited to the description of abstract objects. This is a serious drawback, for in many applications, the complexity of the data objects to be manipulated contributes substantially to the overall complexity of the problem" [5]. This realization had two important consequences. First, data-driven design methods emerged, which provided a disciplined approach to the problems of doing data abstraction in algorithmically oriented languages. Second, theories regarding the concept of a type appeared, which eventually found their realization in languages such as Pascal.

The natural conclusion of these ideas first appeared in the language Simula and was improved upon, resulting in the development of several languages such as Smalltalk, Object Pascal, C++, Ada, Eiffel, and Java. For reasons that we will explain shortly, these languages are called object-based or object-oriented. Figure 2–4 illustrates the topology of such languages for small to moderate-sized applications.

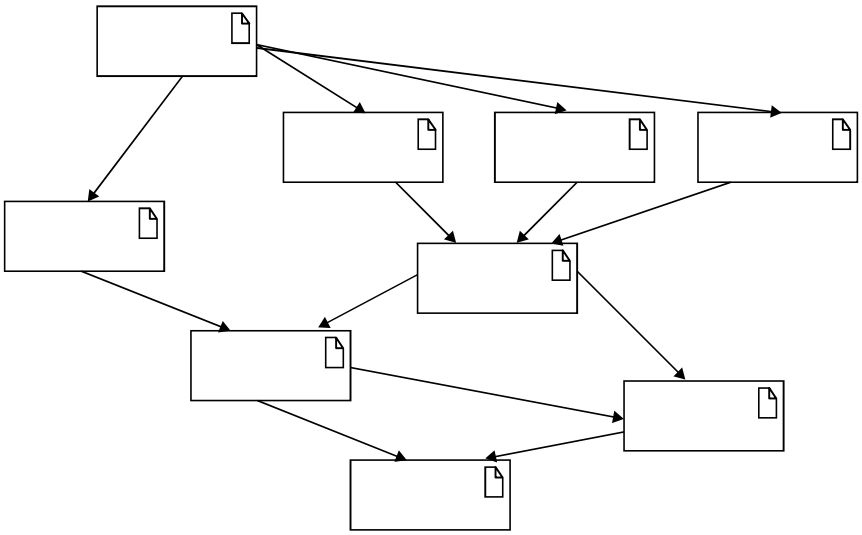


**Figure 2–4** The Topology of Small to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

The physical building block in such languages is the module, which represents a logical collection of classes and objects instead of subprograms, as in earlier languages. To state it another way, "If procedures and functions are verbs and pieces of data are nouns, a procedure-oriented program is organized around verbs while an object-oriented program is organized around nouns" [6]. For this reason, the physical structure of a small to moderate-sized object-oriented application appears as a graph, not as a tree, which is typical of algorithmically oriented languages. Additionally, there is little or no global data. Instead, data and operations are united in such a way that the fundamental logical building blocks of our systems are no longer algorithms, but instead are classes and objects.

By now we have progressed beyond programming-in-the-large and must cope with programming-in-the-colossal. For very complex systems, we find that classes, objects, and modules provide an essential yet insufficient means of abstraction. Fortunately, the object model scales up. In large systems, we find clusters of abstractions built in layers on top of one another. At any given level of abstraction, we find meaningful collections of objects that collaborate to achieve some higher-level behavior. If we look inside any given cluster to view its implementation, we unveil yet another set of cooperative abstractions. This is exactly the organization of complexity described in Chapter 1; this topology is shown in Figure 2–5.

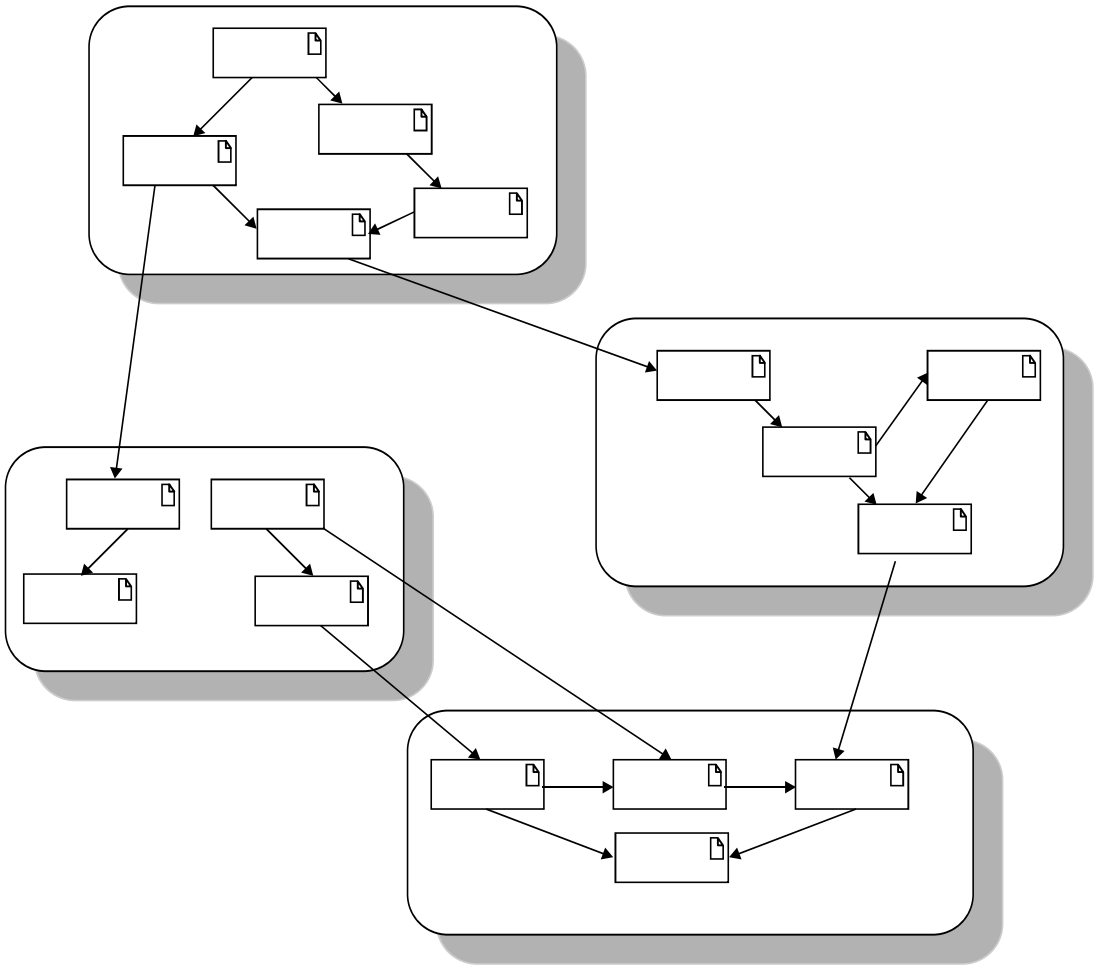


**Figure 2–5** The Topology of Large Applications Using Object-Based and Object-Oriented Programming Languages

## 2.2 Foundations of the Object Model

Structured design methods evolved to guide developers who were trying to build complex systems using algorithms as their fundamental building blocks. Similarly, object-oriented design methods have evolved to help developers exploit the expressive power of object-based and object-oriented programming languages, using the class and object as basic building blocks.

Actually, the object model has been influenced by a number of factors, not just object-oriented programming. Indeed, as further discussed in the sidebar, Foundations—The Object Model, the object model has proven to be a unifying concept in computer science, applicable not just to programming languages but also to the design of user interfaces, databases, and even computer architectures. The reason for this widespread appeal is simply that an object orientation helps us to cope with the complexity inherent in many different kinds of systems.

Object-oriented analysis and design thus represents an evolutionary development, not a revolutionary one; it does not break with advances from the past but builds on proven ones. Unfortunately, most programmers are not rigorously trained in OOAD. Certainly, many good engineers have developed and deployed countless useful software systems using structured design techniques. However, there are limits to the amount of complexity we can handle using only algorithmic decomposition; thus we must turn to object-oriented decomposition. Furthermore, if we try to use languages such as C++ and Java as if they were only traditional, algorithmically oriented languages, we not only miss the power available to us, but we usually end up worse off than if we had used an older language such as C or Pascal. Give a power drill to a carpenter who knows nothing about electricity, and he would use it as a hammer. He will end up bending quite a few nails and smashing several fingers, for a power drill makes a lousy hammer.

Because the object model derives from so many disparate sources, it has unfortunately been accompanied by a muddle of terminology. A Smalltalk programmer uses *methods*, a C++ programmer uses *virtual member functions*, and a CLOS programmer uses *generic functions*. An Object Pascal programmer talks of a *type coercion*; an Ada programmer calls the same thing a *type conversion*; a C# or Java programmer would use a *cast*. To minimize the confusion, let's define what is object-oriented and what is not.

The phrase *object-oriented* "has been bandied about with carefree abandon with much the same reverence accorded 'motherhood,' 'apple pie,' and 'structured programming'"[7]. What we can agree on is that the concept of an object is central to anything object-oriented. In the previous chapter, we informally defined an object as a tangible entity that exhibits some well-defined behavior. Stefik and Bobrow define objects as "entities that combine the properties of procedures and data since they perform computations and save local state" [8]. Defining objects as entities begs the question somewhat, but the basic concept here is that objects serve to unify the ideas of algorithmic and data abstraction. Jones further clarifies this term by noting that "in the object model, emphasis is placed on crisply characterizing the components of the physical or abstract system to be modeled by a programmed system. . . . Objects have a certain 'integrity' which should not—in fact, cannot—be violated. An object can only change state, behave, be manipulated, or stand in relation to other objects in ways appropriate to that object. Stated differently, there exist invariant properties that characterize an object and its behavior. An elevator, for example, is characterized by invariant properties including [that] it only travels up and down inside its shaft. . . . Any elevator simulation must incorporate these invariants, for they are integral to the notion of an elevator" [32].

## Foundations—The Object Model

As Yonezawa and Tokoro point out, "The term 'object' emerged almost independently in various fields in computer science, almost simultaneously in the early 1970s, to refer to notions that were different in their appearance, yet mutually related. All of these notions were invented to manage the complexity of software systems in such a way that objects represented components of a modularly decomposed system or modular units of knowledge representation" [9]. Levy adds that the following events have contributed to the evolution of object-oriented concepts:

- Advances in computer architecture, including capability systems and hardware support for operating systems concepts
- Advances in programming languages, as demonstrated in Simula, Smalltalk, CLU, and Ada
- Advances in programming methodology, including modularization and information hiding [10]

We would add to this list three more contributions to the foundation of the object model:

- Advances in database models
- Research in artificial intelligence
- Advances in philosophy and cognitive science

The concept of an object had its beginnings in hardware over twenty years ago, starting with the invention of descriptor-based architectures and, later, capability-based architectures [11]. These architectures represented a break from the classical von Neumann architectures and came about through attempts to close the gap between the high-level abstractions of programming languages and the low-level abstractions of the machine itself [12]. According to its proponents, the advantages of such architectures are many: better error detection, improved execution efficiency, fewer instruction types, simpler compilation, and reduced storage requirements. Computers can also have an object-oriented architecture.

Closely related to developments in object-oriented architectures are object-oriented operating systems. Dijkstra's work with the THE multiprogramming system first introduced the concept of building systems as layered state machines [18]. Other pioneering object-oriented operating systems include the Plessey/System 250 (for the Plessey 250 multiprocessor), Hydra (for CMU's C.mmp), CALTSS (for the CDC 6400), CAP (for the Cambridge CAP computer), UCLA Secure UNIX (for the PDP 11/45 and 11/70), StarOS (for CMU's Cm*), Medusa (also for CMU's Cm*), and iMAX (for the Intel 432) [19].

Perhaps the most important contribution to the object model derives from the class of programming languages we call object-based and object-oriented. The fundamental ideas of classes and objects first appeared in

the language Simula 67. The Flex system, followed by various dialects of Smalltalk, such as Smalltalk-72, -74, and -76, and finally the current version, Smalltalk-80, took Simula's object-oriented paradigm to its natural conclusion by making everything in the language an instance of a class. In the 1970s languages such as Alphard, CLU, Euclid, Gypsy, Mesa, and Modula were developed, which supported the then-emerging ideas of data abstraction. Language research led to the grafting of Simula and Smalltalk concepts onto traditional high-order programming languages. The unification of object-oriented concepts with C has lead to the languages C++ and Objective C. Then Java arrived to help programmers avoid common programming errors often seen when using C++. Adding object-oriented programming mechanisms to Pascal has led to the languages Object Pascal, Eiffel, and Ada. Additionally, many dialects of Lisp incorporate the object-oriented features of Simula and Smalltalk. Appendix A discusses some of these and other programming language developments in greater detail.

The first person to formally identify the importance of composing systems in layers of abstraction was Dijkstra. Parnas later introduced the idea of information hiding [20], and in the 1970s a number of researchers, most notably Liskov and Zilles [21], Guttag [22], and Shaw [23], pioneered the development of abstract data type mechanisms. Hoare contributed to these developments with his proposal for a theory of types and subclasses [24].

Although database technology has evolved somewhat independently of software engineering, it has also contributed to the object model [25], primarily through the ideas of the entity-relationship (ER) approach to data modeling [26]. In the ER model, first proposed by Chen [27], the world is modeled in terms of its entities, the attributes of these entities, and the relationships among these entities.

In the field of artificial intelligence, developments in knowledge representation have contributed to an understanding of object-oriented abstractions. In 1975, Minsky first proposed a theory of frames to represent real-world objects as perceived by image and natural language recognition systems [28]. Since then, frames have been used as the architectural foundation for a variety of intelligent systems.

Lastly, philosophy and cognitive science have contributed to the advancement of the object model. The idea that the world could be viewed in terms of either objects or processes was a Greek innovation, and in the seventeenth century, we find Descartes observing that humans naturally apply an object-oriented view of the world [29]. In the twentieth century, Rand expanded on these themes in her philosophy of objectivist epistemology [30]. More recently, Minsky has proposed a model of human intelligence in which he considers the mind to be organized as a society of otherwise mindless agents [31]. Minsky argues that only through the cooperative behavior of these agents do we find what we call *intelligence*.

# Object-Oriented Programming

What, then, is object-oriented programming (OOP)? We define it as follows:

> Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

There are three important parts to this definition: (1) Object-oriented programming uses objects, not algorithms, as its fundamental logical building blocks (the "part of" hierarchy we introduced in Chapter 1); (2) each object is an instance of some class; and (3) classes may be related to one another via inheritance relationships (the "is a" hierarchy we spoke of in Chapter 1). A program may appear to be object-oriented, but if any of these elements is missing, it is not an object-oriented program. Specifically, programming without inheritance is distinctly not object-oriented; that would merely be programming with abstract data types.

By this definition, some languages are object-oriented, and some are not. Stroustrup suggests that "if the term 'object-oriented language' means anything, it must mean a language that has mechanisms that support the object-oriented style of programming well. . . . A language supports a programming style well if it provides facilities that make it convenient to use that style. A language does not support a technique if it takes exceptional effort or skill to write such programs; in that case, the language merely enables programmers to use the techniques" [33]. From a theoretical perspective, one can fake object-oriented programming in non-object-oriented programming languages like Pascal and even COBOL or assembly language, but it is horribly ungainly to do so. Cardelli and Wegner thus say:

> [A] language is object-oriented if and only if it satisfies the following requirements:
>
> - It supports objects that are data abstractions with an interface of named operations and a hidden local state.
> - Objects have an associated type [class].
> - Types [classes] may inherit attributes from supertypes [superclasses]. [34]

For a language to support inheritance means that it is possible to express "is a" relationships among types, for example, a red rose is a kind of flower, and a flower is a kind of plant. If a language does not provide direct support for inheritance, then it is not object-oriented. Cardelli and Wegner distinguish such languages by calling them *object-based* rather than *object-oriented*. Under this definition, Smalltalk, Object Pascal, C++, Eiffel, CLOS, C#, and Java are all object-oriented, and Ada83 is object-based (support for object orientation was later added to Ada95). However, since objects and classes are elements of both

kinds of languages, it is both possible and highly desirable for us to use object-oriented design methods for both object-based and object-oriented programming languages.

# Object-Oriented Design

The emphasis in programming methods is primarily on the proper and effective use of particular language mechanisms. By contrast, design methods emphasize the proper and effective structuring of a complex system. What, then, is object-oriented design (OOD)? We suggest the following:

> Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design.

There are two important parts to this definition: object-oriented design (1) leads to an object-oriented decomposition and (2) uses different notations to express different models of the logical (class and object structure) and physical (module and process architecture) design of a system, in addition to the static and dynamic aspects of the system.

The support for object-oriented decomposition is what makes object-oriented design quite different from structured design: The former uses class and object abstractions to logically structure systems, and the latter uses algorithmic abstractions. We will use the term *object-oriented design* to refer to any method that leads to an object-oriented decomposition.

# Object-Oriented Analysis

The object model has influenced even earlier phases of the software development lifecycle. Traditional structured analysis techniques, best typified by the work of DeMarco [35], Yourdon [36], and Gane and Sarson [37], with real-time extensions by Ward and Mellor [38] and by Hatley and Pirbhai [39], focus on the flow of data within a system. Object-oriented analysis (OOA) emphasizes the building of real-world models, using an object-oriented view of the world:

> Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.

How are OOA, OOD, and OOP related? Basically, the products of object-oriented analysis serve as the models from which we may start an object-oriented design;

the products of object-oriented design can then be used as blueprints for completely implementing a system using object-oriented programming methods.

## 2.3 Elements of the Object Model

Jenkins and Glasgow observe that "most programmers work in one language and use only one programming style. They program in a paradigm enforced by the language they use. Frequently, they have not been exposed to alternate ways of thinking about a problem, and hence have difficulty in seeing the advantage of choosing a style more appropriate to the problem at hand" [40]. Bobrow and Stefik define a programming style as "a way of organizing programs on the basis of some conceptual model of programming and an appropriate language to make programs written in the style clear" [41]. They further suggest that there are five main kinds of programming styles, listed here with the kinds of abstractions they employ:

| | |
|---|---|
| 1. Procedure-oriented | Algorithms |
| 2. Object-oriented | Classes and objects |
| 3. Logic-oriented | Goals, often expressed in a predicate calculus |
| 4. Rule-oriented | If–then rules |
| 5. Constraint-oriented | Invariant relationships |

There is no single programming style that is best for all kinds of applications. For example, rule-oriented programming would be best suited for the design of a knowledge base, and procedure-oriented programming would be best for the design of computation-intense operations. From our experience, the object-oriented style is best suited to the broadest set of applications; indeed, this programming paradigm often serves as the architectural framework in which we employ other paradigms.

Each of these styles of programming is based on its own conceptual framework. Each requires a different mindset, a different way of thinking about the problem. For all things object-oriented, the conceptual framework is the object model. There are four major elements of this model:

1. Abstraction
2. Encapsulation
3. Modularity
4. Hierarchy

By *major*, we mean that a model without any one of these elements is not object-oriented.

There are three minor elements of the object model:

1. Typing
2. Concurrency
3. Persistence

By *minor*, we mean that each of these elements is a useful, but not essential, part of the object model.

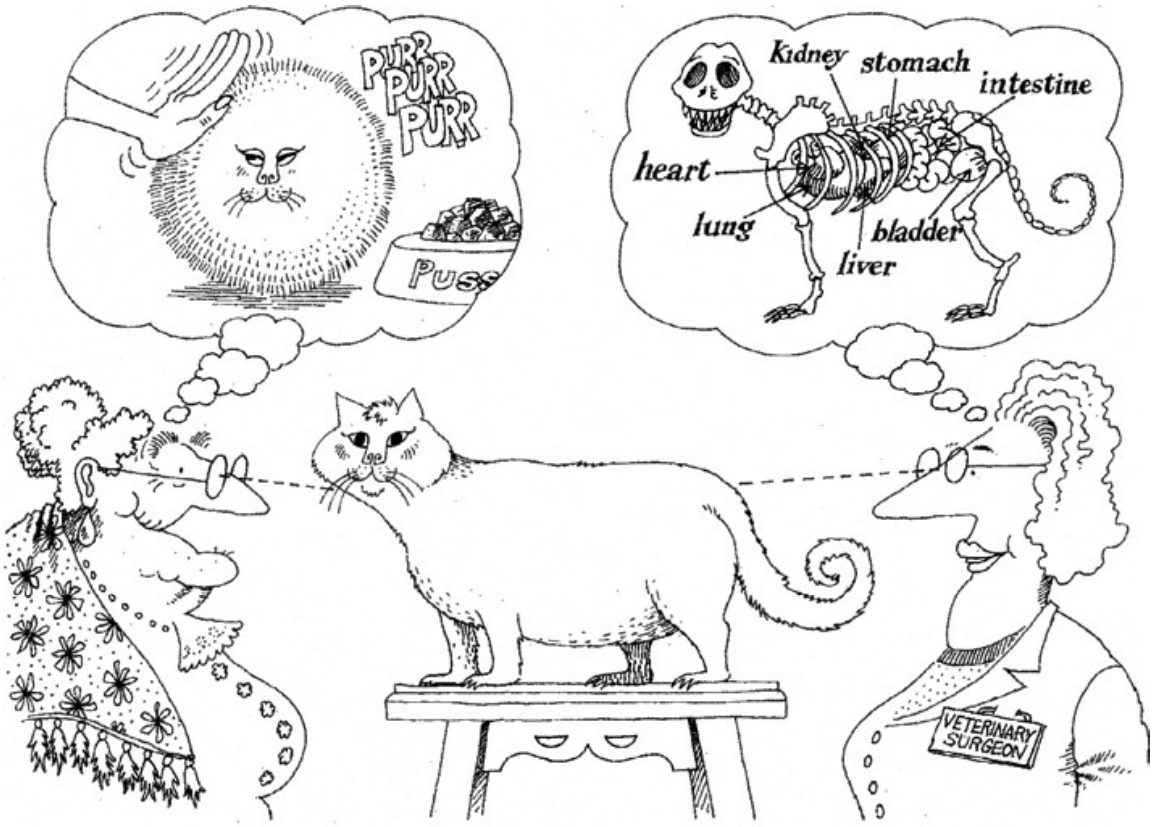
Without this conceptual framework, you may be programming in a language such as Smalltalk, Object Pascal, C++, Eiffel, or Ada, but your design is going to smell like a FORTRAN, Pascal, or C application. You will have missed out on or otherwise abused the expressive power of the object-oriented language you are using for implementation. More importantly, you are not likely to have mastered the complexity of the problem at hand.

# The Meaning of Abstraction

Abstraction is one of the fundamental ways that we as humans cope with complexity. Dahl, Dijkstra, and Hoare suggest that "abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon these similarities and to ignore for the time being the differences" [42]. Shaw defines an abstraction as "a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial or diversionary" [43]. Berzins, Gray, and Naumann recommend that "a concept qualifies as an abstraction only if it can be described, understood, and analyzed independently of the mechanism that will eventually be used to realize it" [44]. Combining these different viewpoints, we define an abstraction as follows:

> An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

An abstraction focuses on the outside view of an object and so serves to separate an object's essential behavior from its implementation. Abelson and Sussman call this behavior/implementation division an abstraction barrier [45] achieved by applying the *principle of least commitment*, through which the interface of an object provides its essential behavior, and nothing more [46]. We like to use an additional principle that we call the *principle of least astonishment*, through which an abstraction captures the entire behavior of some object, no more and no less, and offers no surprises or side effects that go beyond the scope of the abstraction.




Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.

Deciding on the right set of abstractions for a given domain is the central problem in object-oriented design. Because this topic is so important, the whole of Chapter 4 is devoted to it.

"There is a spectrum of abstraction, from objects which closely model problem domain entities to objects which really have no reason for existence" [47]. From the most to the least useful, these kinds of abstractions include the following:

| | |
|---|---|
| ■ Entity abstraction | An object that represents a useful model of a problem domain or solution domain entity |
| ■ Action abstraction | An object that provides a generalized set of operations, all of which perform the same kind of function |
| ■ Virtual machine abstraction | An object that groups operations that are all used by some superior level of control, or operations that all use some junior-level set of operations |
| ■ Coincidental abstraction | An object that packages a set of operations that have no relation to each other |

We strive to build entity abstractions because they directly parallel the vocabulary of a given problem domain.

A client is any object that uses the resources of another object (known as the server). We can characterize the behavior of an object by considering the services that it provides to other objects, as well as the operations that it may perform on other objects. This view forces us to concentrate on the outside view of an object and leads us to what Meyer calls the *contract model* of programming [48]: the outside view of each object defines a contract on which other objects may depend, and which in turn must be carried out by the inside view of the object itself (often in collaboration with other objects). This contract thus establishes all the assumptions a client object may make about the behavior of a server object. In other words, this contract encompasses the responsibilities of an object, namely, the behavior for which it is held accountable [49].

Individually, each operation that contributes to this contract has a unique signature comprising all of its formal arguments and return type. We call the entire set of operations that a client may perform on an object, together with the legal orderings in which they may be invoked, its *protocol*. A protocol denotes the ways in which an object may act and react and thus constitutes the entire static and dynamic outside view of the abstraction.

Central to the idea of an abstraction is the concept of invariance. An *invariant* is some Boolean (true or false) condition whose truth must be preserved. For each operation associated with an object, we may define *preconditions* (invariants assumed by the operation) as well as *postconditions* (invariants satisfied by the operation). Violating an invariant breaks the contract associated with an abstraction. If a precondition is violated, this means that a client has not satisfied its part of the bargain, and hence the server cannot proceed reliably. Similarly, if a postcondition is violated, this means that a server has not carried out its part of the contract, and so its clients can no longer trust the behavior of the server. An exception is an indication that some invariant has not been or cannot be satisfied. Certain languages permit objects to throw exceptions so as to abandon processing and alert some other object to the problem, which in turn may catch the exception and handle the problem.

As an aside, the terms *operation*, *method*, and *member function* evolved from three different programming cultures (Ada, Smalltalk, and C++, respectively). They all mean virtually the same thing, so we will use them interchangeably.

All abstractions have static as well as dynamic properties. For example, a file object takes up a certain amount of space on a particular memory device; it has a name, and it has contents. These are all static properties. The value of each of these properties is dynamic, relative to the lifetime of the object: A file object may grow or shrink in size, its name may change, its contents may change. In a

procedure-oriented style of programming, the activity that changes the dynamic value of objects is the central part of all programs; things happen when subprograms are called and statements are executed. In a rule-oriented style of programming, things happen when new events cause rules to fire, which in turn may trigger other rules, and so on. In an object-oriented style of programming, things happen whenever we operate on an object (i.e., when we send a message to an object). Thus, invoking an operation on an object elicits some reaction from the object. What operations we can meaningfully perform on an object and how that object reacts constitute the entire behavior of the object.

## *Examples of Abstraction*

Let's illustrate these concepts with some examples. We defer a complete treatment of how to find the right abstractions for a given problem to Chapter 4.

On a hydroponics farm, plants are grown in a nutrient solution, without sand, gravel, or other soils. Maintaining the proper greenhouse environment is a delicate job and depends on the kind of plant being grown and its age. One must control diverse factors such as temperature, humidity, light, pH, and nutrient concentrations. On a large farm, it is not unusual to have an automated system that constantly monitors and adjusts these elements. Simply stated, the purpose of an automated gardener is to efficiently carry out, with minimal human intervention, growing plans for the healthy production of multiple crops.

One of the key abstractions in this problem is that of a sensor. Actually, there are several different kinds of sensors. Anything that affects production must be measured, so we must have sensors for air and water temperature, humidity, light, pH, and nutrient concentrations, among other things. Viewed from the outside, a temperature sensor is simply an object that knows how to measure the temperature at some specific location. What is a temperature? It is some numeric value, within a limited range of values and with a certain precision, that represents degrees in the scale of Fahrenheit, Centigrade, or Kelvin, whichever is most appropriate for our problem. What is a location? It is some identifiable place on the farm at which we desire to measure the temperature; presumably, there are only a few such locations. What is important for a temperature sensor is not so much where it is located but the fact that it has a location and identity unique from all other temperature sensors. Now we are ready to ask: What are the responsibilities of a temperature sensor? Our design decision is that a sensor is responsible for knowing the temperature at a given location and reporting that temperature when asked. More concretely, what operations can a client perform on a temperature sensor? Our design decision is that a client can calibrate it, as well as ask what the current temperature is. (See Figure 2–6. Note that this representation is similar to the representation of a class in UML 2.0. You will learn the actual representation in Chapter 5.)

| **Abstraction:** Temperature Sensor |
| --- |
| **Important Characteristics:**<br>temperature<br>location |
| **Responsibilities:**<br>report current temperature<br>calibrate |

**Figure 2–6** Abstraction of a `Temperature Sensor`

The abstraction we have described thus far is passive; some client object must operate on an air `Temperature Sensor` object to determine its current temperature. However, there is another legitimate abstraction that may be more or less appropriate depending on the broader system design decisions we might make. Specifically, rather than the `Temperature Sensor` being passive, we might make it active, so that it is not acted on but rather acts on other objects whenever the temperature at its location changes a certain number of degrees from a given setpoint. This abstraction is almost the same as our first one, except that its responsibilities have changed slightly: A sensor is now responsible for reporting the current temperature when it changes, not just when asked. What new operations must this abstraction provide?

This abstraction is a bit more complicated than the first (see Figure 2–7). A client of this abstraction may invoke an operation to establish a critical range of temperatures. It is then the responsibility of the sensor to report whenever the temperature at its location drops below or rises above the given setpoint. When the function is invoked, the sensor provides its location and the current temperature, so that the client has sufficient information to respond to the condition.

| **Abstraction:** Active Temperature Sensor |
| --- |
| **Important Characteristics:**<br>temperature<br>location<br>setpoint |
| **Responsibilities:**<br>report current temperature<br>calibrate<br>establish setpoint |

**Figure 2–7** Abstraction of an `Active Temperature Sensor`

How the `Active Temperature Sensor` carries out its responsibilities is a function of its inside view and is of no concern to outside clients. These then are the secrets of the class, which are implemented by the class's private parts together with the definition of its member functions.

Let's consider a different abstraction. For each crop, there must be a growing plan that describes how temperature, light, nutrients, and other conditions should change over time to maximize the harvest. A growing plan is a legitimate entity abstraction because it forms part of the vocabulary of the problem domain. Each crop has its own growing plan, but the growing plans for all crops take the same form.

A growing plan is responsible for keeping track of all interesting actions associated with growing a crop, correlated with the times at which those actions should take place. For example, on day 15 in the lifetime of a certain crop, our growing plan might be to maintain a temperature of 78°F for 16 hours, turn on the lights for 14 of these hours, and then drop the temperature to 65°F for the rest of the day. We might also want to add certain extra nutrients in the middle of the day, while still maintaining a slightly acidic pH. From the perspective outside of each growing-plan object, a client must be able to establish the details of a plan, modify a plan, and inquire about a plan, as shown in Figure 2–8. (Note that abstractions are likely to evolve over the lifetime of a project. As details begin to be fleshed out, a responsibility such as "establish plan" could turn into multiple responsibilities, such as "set temperature," "set pH," and so forth. This is to be expected as more knowledge of client requirements is gained, designs mature, and implementation approaches are considered.)

Our decision is also that we will not require a growing plan to carry out its plan: We will leave this as the responsibility of a different abstraction (e.g., a `Plan Controller`). In this manner, we create a clear *separation of concerns* among the logically different parts of the system, so as to reduce the conceptual size of each individual abstraction. For example, there might be an object that sits at the

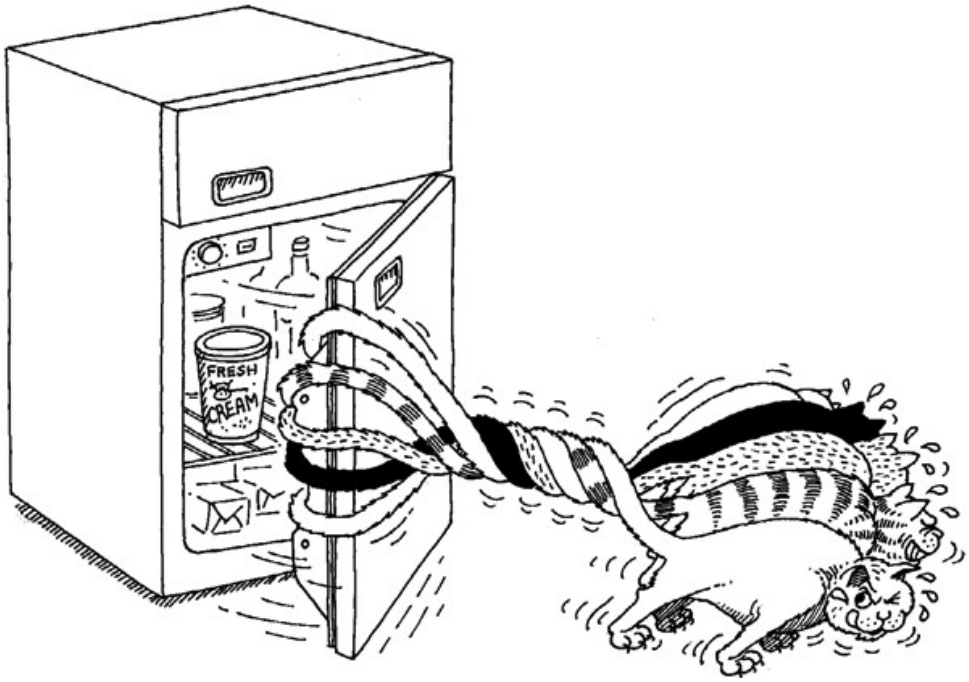
| **Abstraction:** Growing Plan |
|---|
| **Important Characteristics:**<br>name |
| **Responsibilities:**<br>establish plan<br>modify plan<br>clear plan |

Related Candidate Abstractions: Crop, Conditions, Plan Controller

**Figure 2–8** Abstraction of a `Growing Plan`

boundary of the human/machine interface and translates human input into plans. This is the object that establishes the details of a growing plan, so it must be able to change the state of a `Growing Plan` object. There must also be an object that carries out the growing plan, and it must be able to read the details of a plan for a particular time.

As this example points out, no object stands alone; every object collaborates with other objects to achieve some behavior.[1] Our design decisions about how these objects cooperate with one another define the boundaries of each abstraction and thus the responsibilities and protocol of each object.



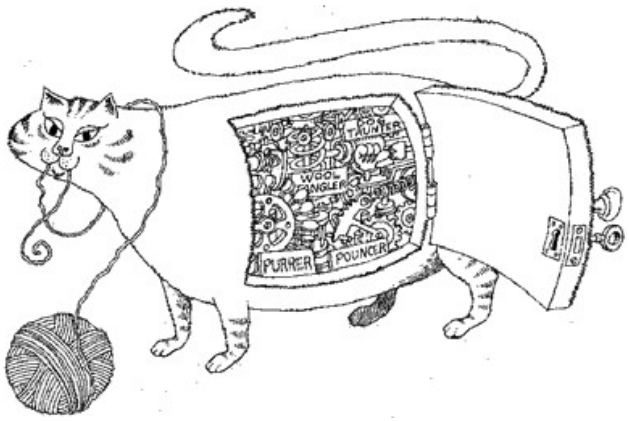

Objects collaborate with other objects to achieve some behavior.

# The Meaning of Encapsulation

Although we earlier described our abstraction of the `Growing Plan` as a time/action mapping, its implementation is not necessarily a literal table or map data structure. Indeed, whichever representation is chosen is immaterial to the client's contract with the `Growing Plan`, as long as that representation upholds the contract. Simply stated, the abstraction of an object should precede the decisions about its implementation. Once an implementation is selected, it should be treated as a secret of the abstraction and hidden from most clients.

---

1. Stated another way, with apologies to the poet John Donne, no object is an island (although an island may be abstracted as an object).




Encapsulation hides the details of the implementation of an object.

Abstraction and encapsulation are complementary concepts: Abstraction focuses on the observable behavior of an object, whereas encapsulation focuses on the implementation that gives rise to this behavior. Encapsulation is most often achieved through information hiding (not just data hiding), which is the process of hiding all the secrets of an object that do not contribute to its essential characteristics; typically, the structure of an object is hidden, as well as the implementation of its methods. "No part of a complex system should depend on the internal details of any other part" [50]. Whereas abstraction "helps people to think about what they are doing," encapsulation "allows program changes to be reliably made with limited effort" [51].

Encapsulation provides explicit barriers among different abstractions and thus leads to a clear separation of concerns. For example, consider again the structure of a plant. To understand how photosynthesis works at a high level of abstraction, we can ignore details such as the responsibilities of plant roots or the chemistry of cell walls. Similarly, in designing a database application, it is standard practice to write programs so that they don't care about the physical representation of data but depend only on a schema that denotes the data's logical view [52]. In both of these cases, objects at one level of abstraction are shielded from implementation details at lower levels of abstraction.

"For abstraction to work, implementations must be encapsulated" [53]. In practice, this means that each class must have two parts: an interface and an implementation. The interface of a class captures only its outside view, encompassing our abstraction of the behavior common to all instances of the class. The implementation of a class comprises the representation of the abstraction as well as the mechanisms that achieve the desired behavior. The interface of a class is the one place where we assert all of the assumptions that a client may make about any instances of the class; the implementation encapsulates details about which no client may make assumptions.

To summarize, we define encapsulation as follows:

> Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.

Britton and Parnas call these encapsulated elements the "secrets" of an abstraction [54].

## *Examples of Encapsulation*

To illustrate the principle of encapsulation, let's return to the problem of the Hydroponics Gardening System. Another key abstraction in this problem domain is that of a heater. A heater is at a fairly low level of abstraction, and thus we might decide that there are only three meaningful operations that we can perform on this object: turn it on, turn it off, and find out if it is running.

### Separation of Concerns

We do not make it a responsibility of the `Heater` abstraction to maintain a fixed temperature. Instead, we choose to give this responsibility to another object (e.g., the `Heater Controller`), which must collaborate with a temperature sensor and a heater to achieve this higher-level behavior. We call this behavior *higher-level* because it builds on the primitive semantics of temperature sensors and heaters and adds some new semantics, namely, *hysteresis*, which prevents the heater from being turned on and off too rapidly when the temperature is near boundary conditions. By deciding on this separation of responsibilities, we make each individual abstraction more cohesive.

All a client needs to know about the class `Heater` is its available interface (i.e., the responsibilities that it may execute at the client's request—see Figure 2–9).

Turning to the inside view of the `Heater`, we have an entirely different perspective. Suppose that our system engineers have decided to locate the computers that control each greenhouse away from the building (perhaps to avoid the harsh environment) and to connect each computer to its sensors and actuators via serial lines. One reasonable implementation for the `Heater` class might be to use an electromechanical relay that controls the power going to each physical heater, with the relays in turn commanded by messages sent along these serial lines. For example, to turn on a heater, we might transmit a special command string, followed by a number identifying the specific heater, followed by another number used to signal turning the heater on.

| **Abstraction:** Heater |
| --- |
| **Important Characteristics:**<br>location<br>status |
| **Responsibilities:**<br>turn on<br>turn off<br>provide status |

Related Candidate Abstractions: Heater Controller, Temperature Sensor

**Figure 2–9** Abstraction of a `Heater`

Suppose that for whatever reason our system engineers choose to use memory-mapped I/O instead of serial communication lines. We would not need to change the interface of the `Heater`, yet the implementation would be very different. The client would not see any change at all as the client sees only the `Heater` interface. This is the key point of encapsulation. In fact, the client should not care what the implementation is, as long as it receives the service it needs from the `Heater`.

Let's next consider the implementation of the class `GrowingPlan`. As we mentioned earlier, a growing plan is essentially a time/action mapping. Perhaps the most reasonable representation for this abstraction would be a dictionary of time/action pairs, using an open hash table. We need not store an action for every hour, because things don't change that quickly. Rather, we can store actions only for when they change, and have the implementation extrapolate between times.

In this manner, our implementation encapsulates two secrets: the use of an open hash table (which is distinctly a part of the vocabulary of the solution domain, not the problem domain) and the use of extrapolation to reduce our storage requirements (otherwise we would have to store many more time/action pairs over the duration of a growing season). No client of this abstraction need ever know about these implementation decisions because they do not materially affect the outwardly observable behavior of the class.

Intelligent encapsulation localizes design decisions that are likely to change. As a system evolves, its developers might discover that, in actual use, certain operations take longer than is acceptable or that some objects consume more space than is available. In such situations, the representation of an object is often changed so that more efficient algorithms can be applied or so that one can optimize for space by calculating rather than storing certain data. This ability to change the representation of an abstraction without disturbing any of its clients is the essential benefit of encapsulation.

Hiding is a relative concept: What is hidden at one level of abstraction may represent the outside view at another level of abstraction. The underlying representation of an object can be revealed, but in most cases only if the creator of the abstraction explicitly exposes the implementation, and then only if the client is willing to accept the resulting additional complexity. Thus, encapsulation cannot stop a developer from doing stupid things; as Stroustrup points out, "Hiding is for the prevention of accidents, not the prevention of fraud" [56]. Of course, no programming language prevents a human from literally seeing the implementation of a class, although an operating system might deny access to a particular file that contains the implementation of a class.

## The Meaning of Modularity

"The act of partitioning a program into individual components can reduce its complexity to some degree. . . . Although partitioning a program is helpful for this reason, a more powerful justification for partitioning a program is that it creates a number of well-defined, documented boundaries within the program. These boundaries, or interfaces, are invaluable in the comprehension of the program" [57]. In some languages, such as Smalltalk, there is no concept of a module, so the class forms the only physical unit of decomposition. Java has packages that contain classes. In many other languages, including Object Pascal, C++, and Ada, the module is a separate language construct and therefore warrants a separate set of design decisions. In these languages, classes and objects form the logical structure of a system; we place these abstractions in modules to produce the system's physical architecture. Especially for larger applications, in which we may have many hundreds of classes, the use of modules is essential to help manage complexity.


Modularity packages abstractions into discrete units.

"Modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules. We will use the definition of Parnas: 'The connections between modules are the assumptions which the modules make about each other'" [58]. Most languages that support the module as a separate concept also distinguish between the interface of a module and its implementation. Thus, it is fair to say that modularity and encapsulation go hand in hand.

Deciding on the right set of modules for a given problem is almost as hard a problem as deciding on the right set of abstractions. Zelkowitz is absolutely right when he states that "because the solution may not be known when the design stage starts, decomposition into smaller modules may be quite difficult. For older applications (such as compiler writing), this process may become standard, but for new ones (such as defense systems or spacecraft control), it may be quite difficult" [59].

Modules serve as the physical containers in which we declare the classes and objects of our logical design. This is no different than the situation faced by the electrical engineer designing a computer motherboard. NAND, NOR, and NOT gates might be used to construct the necessary logic, but these gates must be physically packaged in standard integrated circuits. Lacking any such standard software parts, the software engineer has considerably more degrees of freedom—as if the electrical engineer had a silicon foundry at his or her disposal.

For tiny problems, the developer might decide to declare every class and object in the same package. For anything but the most trivial software, a better solution is to group logically related classes and objects in the same module and to expose only those elements that other modules absolutely must see. This kind of modularization is a good thing, but it can be taken to extremes. For example, consider an application that runs on a distributed set of processors and uses a message-passing mechanism to coordinate the activities of different programs. In a large system, such as a command and control system, it is common to have several hundred or even a few thousand kinds of messages. A naive strategy might be to define each message class in its own module. As it turns out, this is a singularly poor design decision. Not only does it create a documentation nightmare, but it makes it terribly difficult for any users to find the classes they need. Furthermore, when decisions change, hundreds of modules must be modified or recompiled. This example shows how information hiding can backfire [60]. Arbitrary modularization is sometimes worse than no modularization at all.

In traditional structured design, modularization is primarily concerned with the meaningful grouping of subprograms, using the criteria of coupling and cohesion. In object-oriented design, the problem is subtly different: The task is to decide where to physically package the classes and objects, which are distinctly different from subprograms.

Our experience indicates that there are several useful technical as well as nontechnical guidelines that can help us achieve an intelligent modularization of classes and objects. As Britton and Parnas have observed, "The overall goal of the decomposition into modules is the reduction of software cost by allowing modules to be designed and revised independently. . . . Each module's structure should be simple enough that it can be understood fully; it should be possible to change the implementation of other modules without knowledge of the implementation of other modules and without affecting the behavior of other modules; [and] the ease of making a change in the design should bear a reasonable relationship to the likelihood of the change being needed" [61]. There is a pragmatic edge to these guidelines. In practice, the cost of recompiling the body of a module is relatively small: Only that unit need be recompiled and the application relinked. However, the cost of recompiling the interface of a module is relatively high. Especially with strongly typed languages, one must recompile the module interface, its body, all other modules that depend on this interface, the modules that depend on these modules, and so on. Thus, for very large programs (assuming that our development environment does not support incremental compilation), a change in a single module interface might result in much longer compilation time. Obviously, a development manager cannot often afford to allow a massive "big bang" recompilation to happen too frequently. For this reason, a module's interface should be as narrow as possible, yet still satisfy the needs of the other modules that use it. Our style is to hide as much as we can in the implementation of a module. Incrementally shifting declarations from a module's implementation to its interface is far less painful and destabilizing than ripping out extraneous interface code.

The developer must therefore balance two competing technical concerns: the desire to encapsulate abstractions and the need to make certain abstractions visible to other modules. "System details that are likely to change independently should be the secrets of separate modules; the only assumptions that should appear between modules are those that are considered unlikely to change. Every data structure is private to one module; it may be directly accessed by one or more programs within the module but not by programs outside the module. Any other program that requires information stored in a module's data structures must obtain it by calling module programs" [62]. In other words, strive to build modules that are cohesive (by grouping logically related abstractions) and loosely coupled (by minimizing the dependencies among modules). From this perspective, we may define modularity as follows:

> Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

Thus, the principles of abstraction, encapsulation, and modularity are synergistic. An object provides a crisp boundary around a single abstraction, and both encapsulation and modularity provide barriers around this abstraction.

Two additional technical issues can affect modularization decisions. First, since modules usually serve as the elementary and indivisible units of software that can be reused across applications, a developer might choose to package classes and objects into modules in a way that makes their reuse convenient. Second, many compilers generate object code in segments, one for each module. Therefore, there may be practical limits on the size of individual modules. With regard to the dynamics of subprogram calls, the placement of declarations within modules can greatly affect the locality of reference and thus the paging behavior of a virtual memory system. Poor locality happens when subprogram calls occur across segments and lead to cache misses and page thrashing that ultimately slow down the whole system.

Several competing nontechnical needs may also affect modularization decisions. Typically, work assignments in a development team are given on a module-by-module basis, so the boundaries of modules may be established to minimize the interfaces among different parts of the development organization. Senior designers are usually given responsibility for module interfaces, and more junior developers complete their implementation. On a larger scale, the same situation applies with subcontractor relationships. Abstractions may be packaged so as to quickly stabilize the module interfaces as agreed upon among the various companies. Changing such interfaces usually involves much wailing and gnashing of teeth—not to mention a vast amount of paperwork—so this factor often leads to conservatively designed interfaces. Speaking of paperwork, modules also usually serve as the unit of documentation and configuration management. Having ten modules where one would do sometimes means ten times the paperwork, and so, unfortunately, sometimes the documentation requirements drive the module design decisions (usually in the most negative way). Security may also be an issue. Most code may be considered unclassified, but other code that might be classified secret or higher is best placed in separate modules.

Juggling these different requirements is difficult, but don't lose sight of the most important point: Finding the right classes and objects and then organizing them into separate modules are largely independent design decisions. The identification of classes and objects is part of the logical design of the system, but the identification of modules is part of the system's physical design. One cannot make all the logical design decisions before making all the physical ones, or vice versa; rather, these design decisions happen iteratively.

## *Examples of Modularity*

Let's look at modularity in the Hydroponics Gardening System. Suppose we decide to use a commercially available workstation where the user can control the system's operation. At this workstation, an operator could create new growing plans, modify old ones, and follow the progress of currently active ones. Since one

of our key abstractions here is that of a growing plan, we might therefore create a module whose purpose is to collect all of the classes associated with individual growing plans (e.g., `FruitGrowingPlan`, `GrainGrowingPlan`). The implementations of these `GrowingPlan` classes would appear in the implementation of this module. We might also define a module whose purpose is to collect all of the code associated with all user interface functions.

Our design will probably include many other modules. Ultimately, we must define some main program from which we can invoke this application. In object-oriented design, defining this main program is often the least important decision, whereas in traditional structured design, the main program serves as the root, the keystone that holds everything else together. We suggest that the object-oriented view is more natural, for, as Meyer observes, "Practical software systems are more appropriately described as offering a number of services. Defining these systems by single functions is usually possible, but yields rather artificial answers. . . . Real systems have no top" [63].

# The Meaning of Hierarchy

Abstraction is a good thing, but in all except the most trivial applications, we may find many more different abstractions than we can comprehend at one time. Encapsulation helps manage this complexity by hiding the inside view of our abstractions. Modularity helps also, by giving us a way to cluster logically related abstractions. Still, this is not enough. A set of abstractions often forms a hierarchy, and by identifying these hierarchies in our design, we greatly simplify our understanding of the problem.

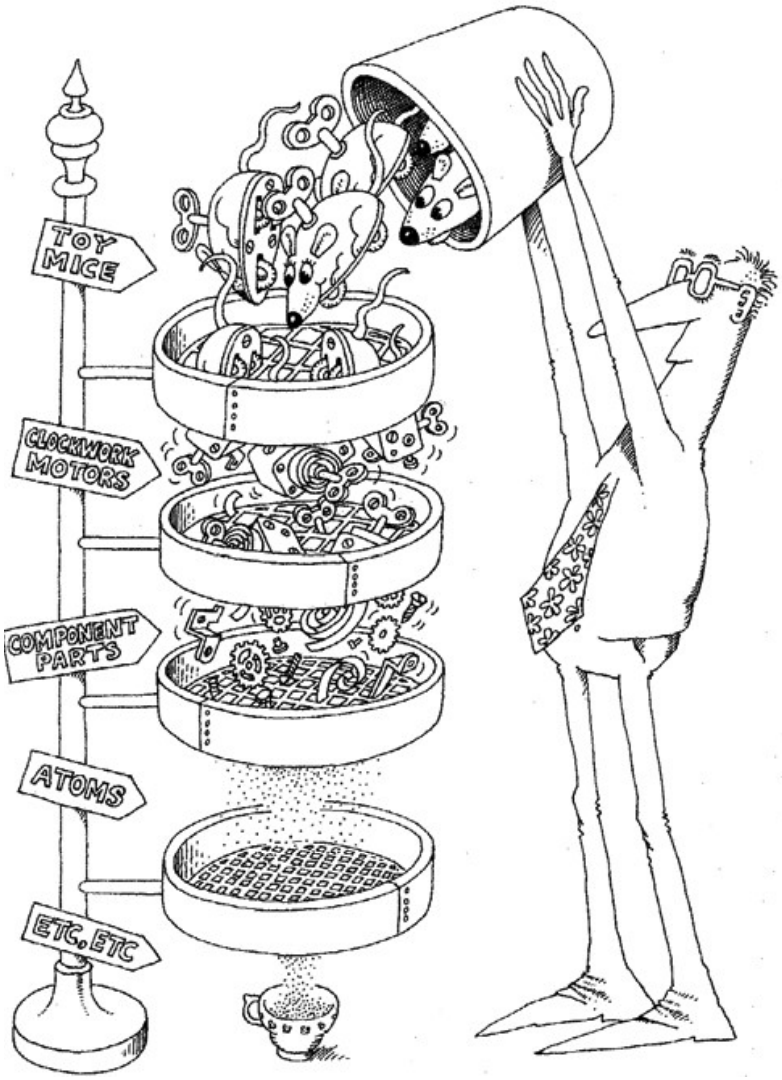
We define hierarchy as follows:

> Hierarchy is a ranking or ordering of abstractions.

The two most important hierarchies in a complex system are its class structure (the "is a" hierarchy) and its object structure (the "part of" hierarchy).

## *Examples of Hierarchy: Single Inheritance*

Inheritance is the most important "is a" hierarchy, and as we noted earlier, it is an essential element of object-oriented systems. Basically, inheritance defines a relationship among classes, wherein one class shares the structure or behavior defined in one or more classes (denoting single inheritance and multiple inheritance, respectively). Inheritance thus represents a hierarchy of abstractions, in which a subclass inherits from one or more superclasses. Typically, a subclass augments or redefines the existing structure and behavior of its superclasses.




Abstractions form a hierarchy.

Semantically, inheritance denotes an "is a" relationship. For example, a bear "is a" kind of mammal, a house "is a" kind of tangible asset, and a quick sort "is a" particular kind of sorting algorithm. Inheritance thus implies a generalization/specialization hierarchy, wherein a subclass specializes the more general structure or behavior of its superclasses. Indeed, this is the litmus test for inheritance: If `B` is not a kind of `A`, then `B` should not inherit from `A`.

Consider the different kinds of growing plans we might use in the Hydroponics Gardening System. An earlier section described our abstraction of a very generalized growing plan. Different kinds of crops, however, demand specialized growing plans. For example, the growing plan for all fruits is generally the same but is quite different from the plan for all vegetables, or for all floral crops. Because of this clustering of abstractions, it is reasonable to define a standard fruit-growing plan that encapsulates the behavior common to all fruits, such as the knowledge of when to pollinate or when to harvest the fruit. We can assert that `FruitGrowingPlan` "is a" kind of `GrowingPlan`.

In this case, `FruitGrowingPlan` is more specialized, and `GrowingPlan` is more general. The same could be said for `GrainGrowingPlan` or `VegetableGrowingPlan`, that is, `GrainGrowingPlan` "is a" kind of `GrowingPlan`, and `VegetableGrowingPlan` "is a" kind of `GrowingPlan`. Here, `GrowingPlan` is the more general superclass, and the others are specialized subclasses.

As we evolve our inheritance hierarchy, the structure and behavior that are common for different classes will tend to migrate to common superclasses. This is why we often speak of inheritance as being a generalization/specialization hierarchy. Superclasses represent generalized abstractions, and subclasses represent specializations in which fields and methods from the superclass are added, modified, or even hidden. In this manner, inheritance lets us state our abstractions with an economy of expression. Indeed, neglecting the "is a" hierarchies that exist can lead to bloated, inelegant designs. "Without inheritance, every class would be a free-standing unit, each developed from the ground up. Different classes would bear no relationship with one another, since the developer of each provides methods in whatever manner he chooses. Any consistency across classes is the result of discipline on the part of the programmers. Inheritance makes it possible to define new software in the same way we introduce any concept to a newcomer, by comparing it with something that is already familiar" [64].

There is a healthy tension among the principles of abstraction, encapsulation, and hierarchy. "Data abstraction attempts to provide an opaque barrier behind which methods and state are hidden; inheritance requires opening this interface to some extent and may allow state as well as methods to be accessed without abstraction" [65]. For a given class, there are usually two kinds of clients: objects that invoke operations on instances of the class and subclasses that inherit from the class. Liskov therefore notes that, with inheritance, encapsulation can be violated in one of three ways: "The subclass might access an instance variable of its superclass, call a private operation of its superclass, or refer directly to superclasses of its superclass" [66]. Different programming languages trade off support for encapsulation and inheritance in different ways. C++ and Java offer great flexibility. Specifically, the interface of a class may have three parts: private parts, which declare members that are accessible only to the class itself; protected parts, which declare members that are accessible only to the class and its subclasses; and public parts, which are accessible to all clients.

## *Examples of Hierarchy: Multiple Inheritance*

The previous example illustrated the use of single inheritance: the subclass `FruitGrowingPlan` had exactly one superclass, the class `GrowingPlan`. For certain abstractions, it is useful to provide inheritance from multiple superclasses.

For example, suppose that we choose to define a class representing a kind of plant. Our analysis of the problem domain might suggest that flowering plants and fruits and vegetables have specialized properties that are relevant to our application. For example, given a flowering plant, its expected time to flower and time to seed might be important to us. Similarly, the time to harvest might be an important part of our abstraction of all fruits and vegetables. One way we could capture our design decisions would be to make two new classes, a `Flower` class and a `FruitVegetable` class, both subclasses of the class `Plant`. However, what if we need to model a plant that both flowered and produced fruit? For example, florists commonly use blossoms from apple, cherry, and plum trees. For this abstraction, we would need to invent a third class, `FlowerFruitVegetable`, that duplicated information from the `Flower` and `FruitVegetable` classes.

A better way to express our abstractions and thereby avoid this redundancy is to use multiple inheritance. First, we invent classes that independently capture the properties unique to flowering plants and to fruits and vegetables. These two classes have no superclass; they stand alone. These are called *mixin classes* because they are meant to be mixed together with other classes to produce new subclasses.

For example, we can define a `Rose` class (see Figure 2–10) that inherits from both `Plant` and `FlowerMixin`. Instances of the subclass `Rose` thus include the structure and behavior from the class `Plant` together with the structure and behavior from the class `FlowerMixin`.

Similarly, a `Carrot` class could be as shown in Figure 2–11. In both cases, we form the subclass by inheriting from two superclasses.

Now, suppose we want to declare a class for a plant such as the cherry tree that has both flowers and fruit. This would be conceptualized as shown in Figure 2–12.

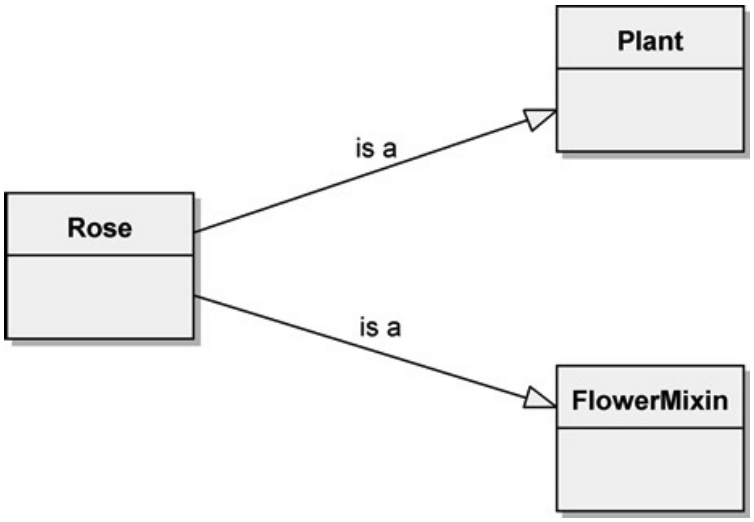



**Figure 2–10** The `Rose` Class, Which Inherits from Multiple Superclasses

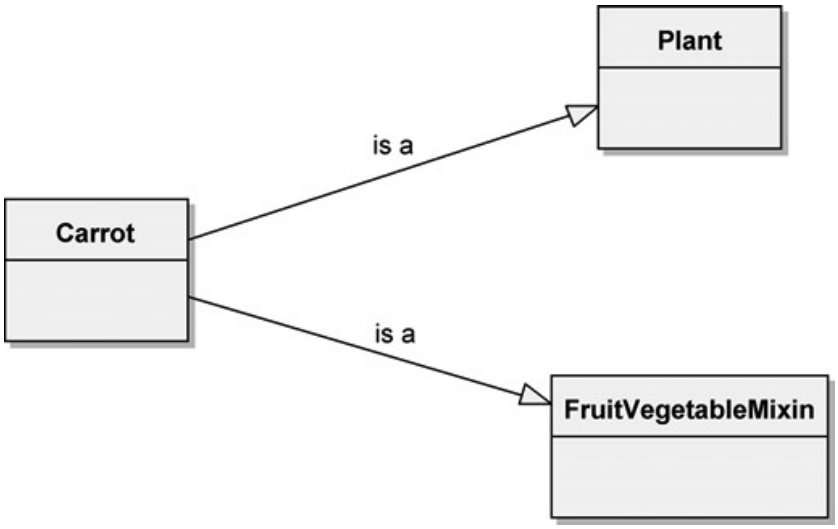



**Figure 2–11** The `Carrot` Class, Which Inherits from Multiple Superclasses

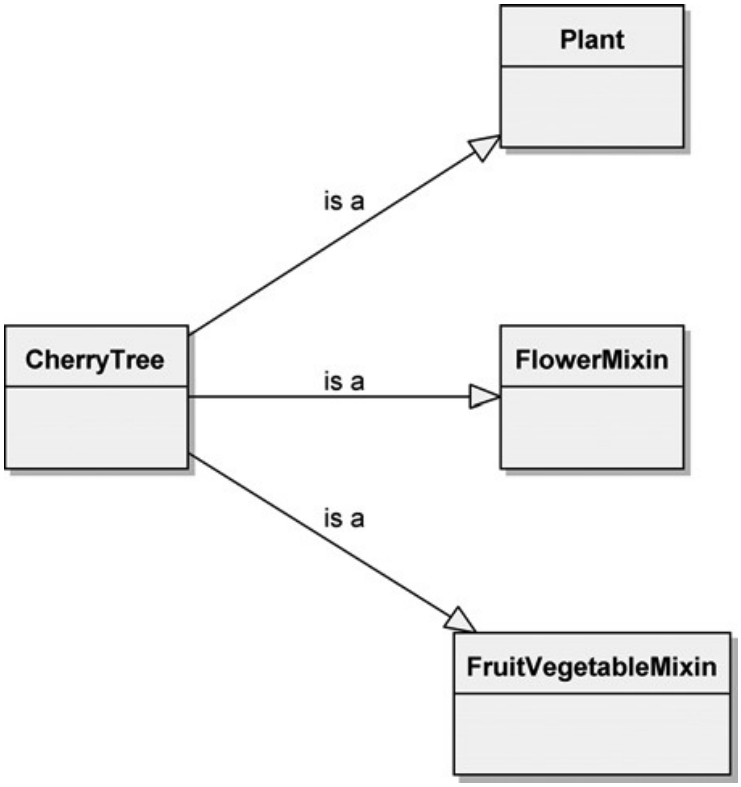



**Figure 2–12** The `CherryTree` Class, Which Inherits from Multiple Superclasses

Multiple inheritance is conceptually straightforward, but it does introduce some practical complexities for programming languages. Languages must address two issues: clashes among names from different superclasses and repeated inheritance. Clashes will occur when two or more superclasses provide a field or operation with the same name or signature as a peer superclass.

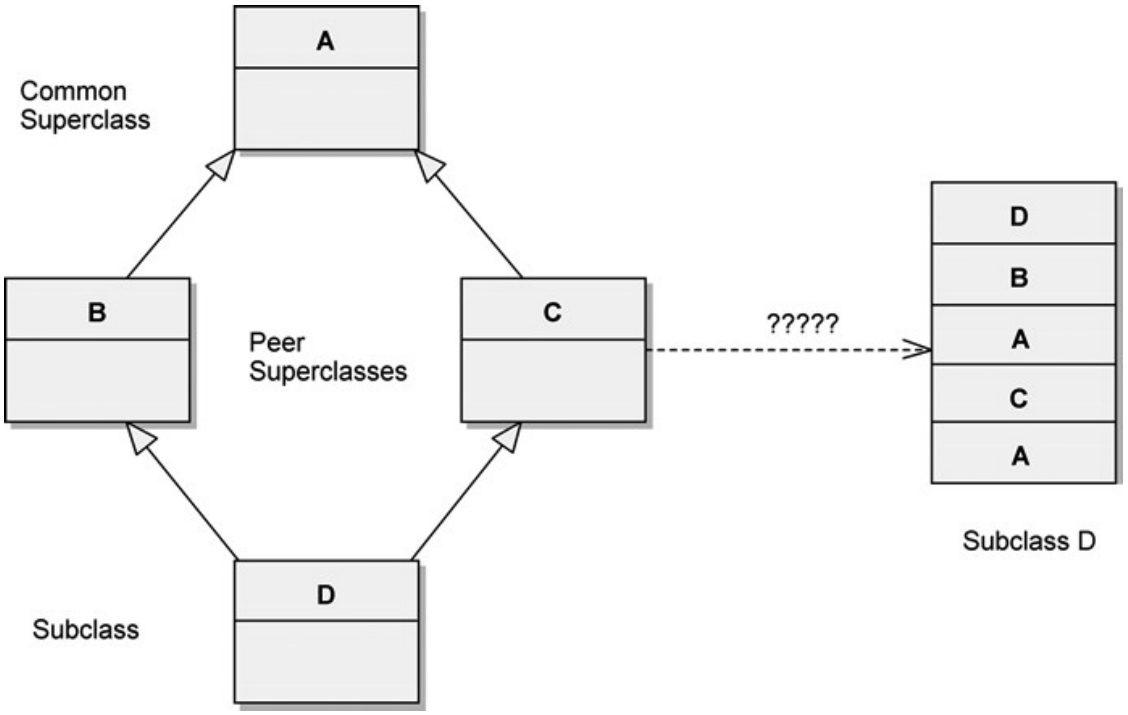



**Figure 2–13** The Repeated Inheritance Problem

Repeated inheritance occurs when two or more peer superclasses share a common superclass. In such a situation, the inheritance lattice will be diamond-shaped, so the question arises, does the leaf class (i.e., subclass) have one copy or multiple copies of the structure of the shared superclass? (See Figure 2–13.) Some languages prohibit repeated inheritance, some unilaterally choose one approach, and others, such as C++, permit the programmer to decide. In C++, virtual base classes are used to denote a sharing of repeated structures, whereas nonvirtual base classes result in duplicate copies appearing in the subclass (with explicit qualification required to distinguish among the copies).

Multiple inheritance is often overused. For example, cotton candy is a kind of candy, but it is distinctly not a kind of cotton. Again, the litmus test for inheritance applies: If `B` is not a kind of `A`, then `B` should not inherit from `A`. Ill-formed multiple inheritance lattices should be reduced to a single superclass plus aggregation of the other classes by the subclass, where possible.

## *Examples of Hierarchy: Aggregation*

Whereas these "is a" hierarchies denote generalization/specialization relationships, "part of" hierarchies describe aggregation relationships. For example, consider the abstraction of a garden. We can contend that a garden consists of a collection of plants together with a growing plan. In other words, plants are "part of" the garden, and the growing plan is "part of" the garden. This "part of" relationship is known as *aggregation*.

Aggregation is not a concept unique to object-oriented development or object-oriented programming languages. Indeed, any language that supports record-like structures supports aggregation. However, the combination of inheritance with aggregation is powerful: Aggregation permits the physical grouping of logically related structures, and inheritance allows these common groups to be easily reused among different abstractions.

When dealing with hierarchies such as these, we often speak of levels of abstraction, a concept first described by Dijkstra [67]. In terms of its "is a" hierarchy, a high-level abstraction is generalized, and a low-level abstraction is specialized. Therefore, we say that a `Flower` class is at a higher level of abstraction than a `Plant` class. In terms of its "part of" hierarchy, a class is at a higher level of abstraction than any of the classes that make up its implementation. Thus, the class `Garden` is at a higher level of abstraction than the type `Plant`, on which it builds.

Aggregation raises the issue of ownership. Our abstraction of a garden permits different plants to be raised in a garden over time, but replacing a plant does not change the identity of the garden as a whole, nor does removing a garden necessarily destroy all of its plants (they are likely just transplanted). In other words, the lifetime of a garden and its plants are independent. In contrast, we have decided that a `GrowingPlan` object is intrinsically associated with a `Garden` object and does not exist independently. Therefore, when we create an instance of `Garden`, we also create an instance of `GrowingPlan`; when we destroy the `Garden` object, we in turn destroy the `GrowingPlan` instance.

# The Meaning of Typing

The concept of a type derives primarily from the theories of abstract data types. As Deutsch suggests, "A type is a precise characterization of structural or behavioral properties which a collection of entities all share" [68]. For our purposes, we will use the terms *type* and *class* interchangeably.[2] Although the concepts of a type and a class are similar, we include typing as a separate element of the object

2. A type and a class are not exactly the same thing; some languages distinguish these two concepts. For example, early versions of the language Trellis/Owl permitted an object to have both a class and a type. In Smalltalk, objects of the classes `SmallInteger`, `LargeNegativeInteger`, and `LargePositiveInteger` are all of the same type, `Integer`, although not of the same class [69]. For most mortals, however, separating the concepts of type and class is utterly confusing and adds very little value. It is sufficient to say that a class implements a type.

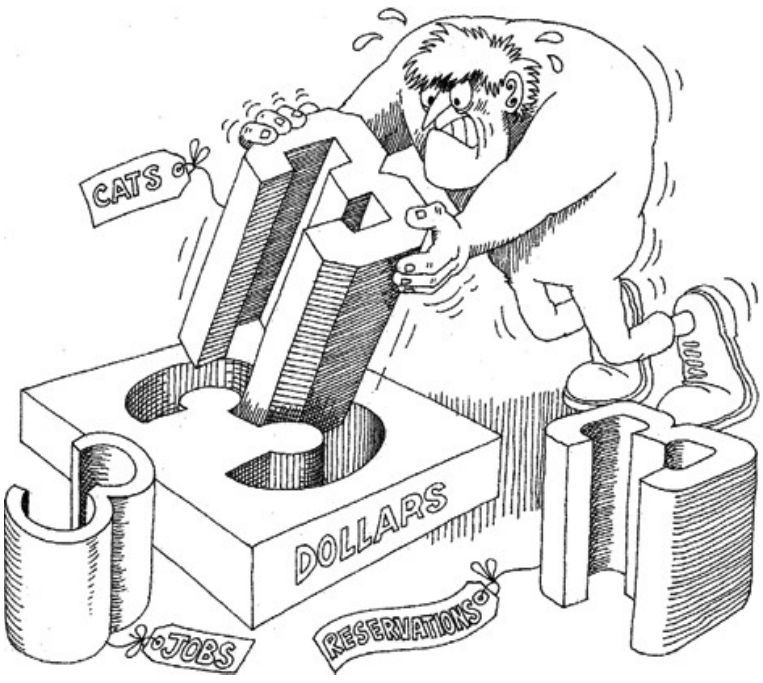
model because the concept of a type places a very different emphasis on the meaning of abstraction. Specifically, we state the following:

> Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways.

Typing lets us express our abstractions so that the programming language in which we implement them can be made to enforce design decisions.

A given programming language may be strongly typed, weakly typed, or even untyped, yet still be called object-oriented. For example, Eiffel is strongly typed, meaning that type conformance is strictly enforced: Operations cannot be called on an object unless the exact signature of that operation is defined in the object's class or superclasses.

The idea of conformance is central to the notion of typing. For example, consider units of measurement in physics [71]. When we divide distance by time, we expect some value denoting speed, not weight. Similarly, dividing a unit of force by temperature doesn't make sense, but dividing force by mass does. These are both examples of strong typing, wherein the rules of our domain prescribe and enforce certain legal combinations of abstractions.




Strong typing prevents mixing of abstractions.

Strong typing lets us use our programming language to enforce certain design decisions and so is particularly relevant as the complexity of our system grows. However, there is a dark side to strong typing. Practically, strong typing introduces semantic dependencies such that even small changes in the interface of a base class require recompilation of all subclasses.

There are two general solutions to these problems. First, we could use a type-safe container class that manipulates only objects of a specific class. This approach addresses the first problem, wherein objects of different types are incorrectly mingled. Second, we could use some form of runtime type identification; this addresses the second problem of knowing what kind of object you happen to be examining at the moment. In general, however, runtime type identification should be used only when there is a compelling reason because it can represent a weakening of encapsulation. As we will discuss in the next section, the use of polymorphic operations can often (but not always) mitigate the need for runtime type identification.

As Tesler points out, there are a number of important benefits to be derived from using strongly typed languages:

- Without type checking, a program in most languages can 'crash' in mysterious ways at runtime.
- In most systems, the edit-compile-debug cycle is so tedious that early error detection is indispensable.
- Type declarations help to document programs.
- Most compilers can generate more efficient object code if types are declared. [72]

Untyped languages offer greater flexibility, but even with untyped languages, as Borning and Ingalls observe, "In almost all cases, the programmer in fact knows what sorts of objects are expected as the arguments of a message, and what sort of object will be returned" [73]. In practice, the safety offered by strongly typed languages usually more than compensates for the flexibility lost by not using an untyped language, especially for programming-in-the-large.

## Examples of Typing: Static and Dynamic Typing

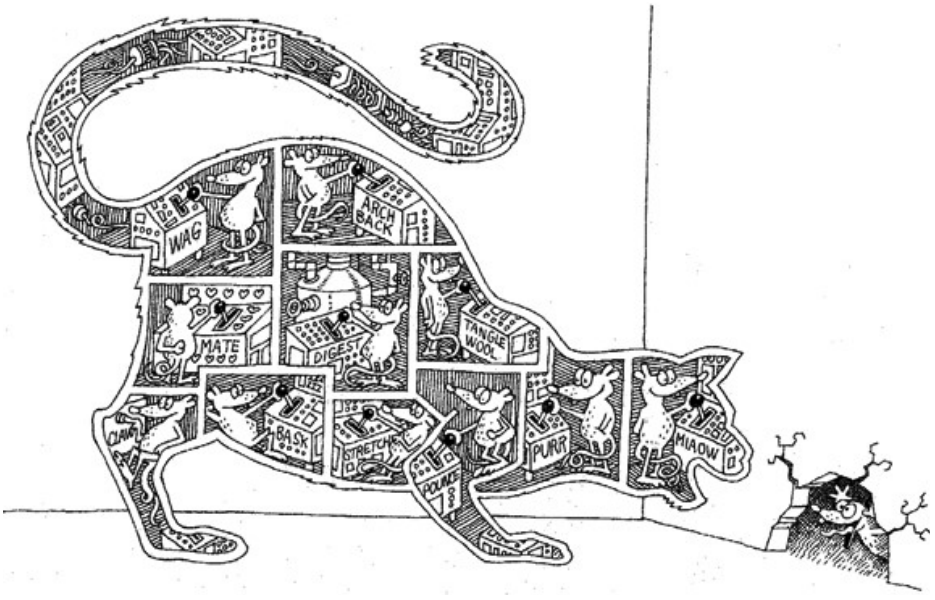
The concepts of *strong and weak* typing and *static and dynamic* typing are entirely different. Strong and weak typing refers to *type consistency*, whereas static and dynamic typing refers to the *time* when names are bound to types. Static typing (also known as *static binding* or *early binding*) means that the types of all variables and expressions are fixed at the time of compilation; dynamic typing (also known as *late binding*) means that the types of all variables and expressions are not known until runtime. A language may be both strongly and statically typed (Ada), strongly typed yet supportive of dynamic typing (C++, Java), or untyped yet supportive of dynamic typing (Smalltalk).

*Polymorphism* is a condition that exists when the features of dynamic typing and inheritance interact. Polymorphism represents a concept in type theory in which a single name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass. Any object denoted by this name is therefore able to respond to some common set of operations [74]. The opposite of polymorphism is *monomorphism*, which is found in all languages that are both strongly and statically typed.

Polymorphism is perhaps the most powerful feature of object-oriented programming languages next to their support for abstraction, and it is what distinguishes object-oriented programming from more traditional programming with abstract data types. As we will see in the following chapters, polymorphism is also a central concept in object-oriented design.

# The Meaning of Concurrency

For certain kinds of problems, an automated system may have to handle many different events simultaneously. Other problems may involve so much computation that they exceed the capacity of any single processor. In each of these cases, it is natural to consider using a distributed set of computers for the target implementation or to use multitasking. A single process is the root from which independent dynamic action occurs within a system. Every program has at least one thread of control, but a system involving concurrency may have many such threads: some that are transitory and others that last the entire lifetime of the system's execution. Systems executing across multiple CPUs allow for truly concurrent threads of




Concurrency allows different objects to act at the same time.

control, whereas systems running on a single CPU can only achieve the illusion of concurrent threads of control, usually by means of some time-slicing algorithm.

We also distinguish between *heavyweight* and *lightweight* concurrency. A heavyweight process is one that is typically independently managed by the target operating system and so encompasses its own address space. A lightweight process usually lives within a single operating system process along with other lightweight processes, which share the same address space. Communication among heavyweight processes is generally expensive, involving some form of interprocess communication; communication among lightweight processes is less expensive and often involves shared data.

Building a large piece of software is hard enough; designing one that encompasses multiple threads of control is much harder because one must worry about such issues as deadlock, livelock, starvation, mutual exclusion, and race conditions. "At the highest levels of abstraction, OOP can alleviate the concurrency problem for the majority of programmers by hiding the concurrency inside reusable abstractions" [76]. Black et al. therefore suggest that "an object model is appropriate for a distributed system because it implicitly defines (1) the units of distribution and movement and (2) the entities that communicate" [77].

Whereas object-oriented programming focuses on data abstraction, encapsulation, and inheritance, concurrency focuses on process abstraction and synchronization [78]. The object is a concept that unifies these two different viewpoints: Each object (drawn from an abstraction of the real world) may represent a separate thread of control (a process abstraction). Such objects are called *active*. In a system based on an object-oriented design, we can conceptualize the world as consisting of a set of cooperative objects, some of which are active and thus serve as centers of independent activity. Given this conception, we define concurrency as follows:

> Concurrency is the property that distinguishes an active object from one that is not active.

## Examples of Concurrency

Let's consider a sensor named `ActiveTemperatureSensor`, whose behavior requires periodically sensing the current temperature and then notifying the client whenever the temperature changes a certain number of degrees from a given setpoint. We do not explain how the class implements this behavior. That fact is a secret of the implementation, but it is clear that some form of concurrency is required.

In general, there are three approaches to concurrency in object-oriented design. First, concurrency is an intrinsic feature of certain programming languages, which provide mechanisms for concurrency and synchronization. In this case, we

may create an active object that runs some process concurrently with all other active objects.

Second, we may use a class library that implements some form of lightweight processes. Naturally, the implementation of this kind is highly platform-dependent, although the interface to the library may be relatively portable. In this approach, concurrency is not an intrinsic part of the language (and so does not place any burdens on nonconcurrent systems) but appears as if it were intrinsic, through the presence of these standard classes.

Third, we may use interrupts to give us the illusion of concurrency. Of course, this requires that we have knowledge of certain low-level hardware details. For example, in our implementation of the class `ActiveTemperatureSensor`, we might have a hardware timer that periodically interrupts the application, during which time all such sensors read the current temperature and then invoke their callback function as necessary.

No matter which approach to concurrency we take, one of the realities about concurrency is that once you introduce it into a system, you must consider how active objects synchronize their activities with one another as well as with objects that are purely sequential. For example, if two active objects try to send messages to a third object, we must be certain to use some means of mutual exclusion, so that the state of the object being acted on is not corrupted when both active objects try to update its state simultaneously. This is the point where the ideas of abstraction, encapsulation, and concurrency interact. In the presence of concurrency, it is not enough simply to define the methods of an object; we must also make certain that the semantics of these methods are preserved in the presence of multiple threads of control.

## The Meaning of Persistence

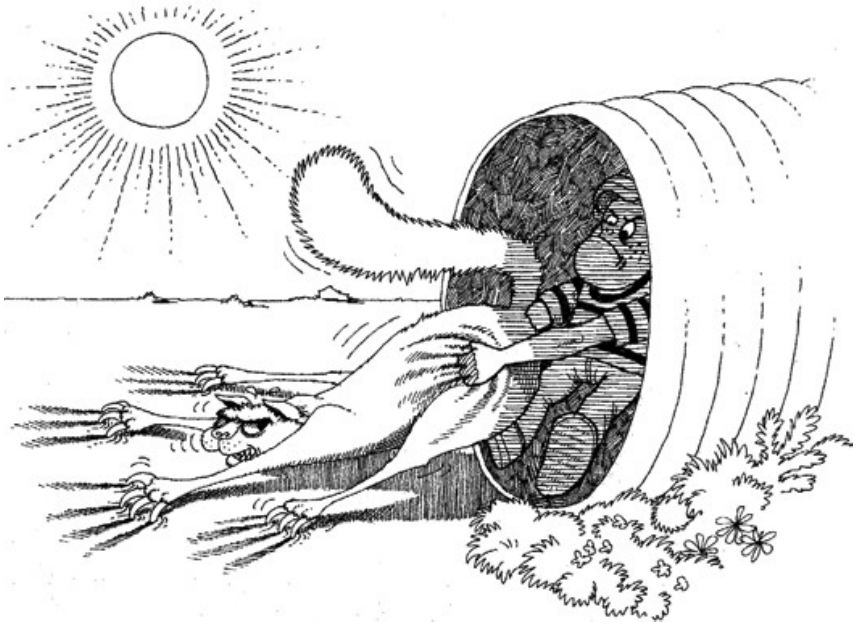
An object in software takes up some amount of space and exists for a particular amount of time. Atkinson et al. suggest that there is a continuum of object existence, ranging from transitory objects that arise within the evaluation of an expression to objects in a database that outlive the execution of a single program. This spectrum of object persistence encompasses the following:

- Transient results in expression evaluation
- Local variables in procedure activations
- Own variables [as in ALGOL 60], global variables, and heap items whose extent is different from their scope
- Data that exists between executions of a program
- Data that exists between various versions of a program
- Data that outlives the program [79]

Traditional programming languages usually address only the first three kinds of object persistence; persistence of the last three kinds is typically the domain of database technology. This leads to a clash of cultures that sometimes results in very strange architectures: Programmers end up crafting ad hoc schemes for storing objects whose state must be preserved between program executions, and database designers misapply their technology to cope with transient objects [80].

An interesting variant of Atkinson et al.'s "Data that outlives the program" is the case of Web applications where the application may not be connected to the data it is using through the entire transaction execution. What changes may happen to data provided to a client application or Web service while disconnected to the data source, and how should resolution of the two be handled? Frameworks like Microsoft's ActiveX Data Object for .NET (ADO.NET) have arisen to help address such distributed, disconnected scenarios.

Unifying the concepts of concurrency and objects gives rise to concurrent object-oriented programming languages. In a similar fashion, introducing the concept of persistence to the object model gives rise to object-oriented databases. In practice, such databases build on proven technology, such as sequential, indexed, hierarchical, network, or relational database models, but then offer to the programmer the abstraction of an object-oriented interface, through which database queries and other operations are completed in terms of objects whose lifetimes transcend the lifetime of an individual program. This unification vastly simplifies the development of certain kinds of applications. In particular, it allows us to apply the same design methods to the database and nondatabase segments of an application.



Persistence saves the state and class of an object across time or space.

Some object-oriented programming languages provide direct support for persistence. Java provides Enterprise Java Beans (EJBs) and Java Data Objects. Smalltalk has protocols for streaming objects to and from storage (which must be redefined by subclasses). However, streaming objects to flat files is a naive solution to persistence that does not scale well. Persistence may be achieved through a modest number of commercially available object-oriented databases [81]. A more typical approach to persistence is to provide an object-oriented skin over a relational database. Customized object-relational mappings can be created by the individual developer. However, that is a very challenging task to do well. Frameworks are available to ease this task, such as the open source framework Hibernate [85]. Commercial object-relational mapping software is available. This approach is most appealing when there is a large capital investment in relational database technology that would be risky or too expensive to replace.

Persistence deals with more than just the lifetime of data. In object-oriented databases, not only does the state of an object persist, but its class must also transcend any individual program, so that every program interprets this saved state in the same way. This clearly makes it challenging to maintain the integrity of a database as it grows, particularly if we must change the class of an object.

Our discussion thus far pertains to persistence in time. In most systems, an object, once created, consumes the same physical memory until it ceases to exist. However, for systems that execute on a distributed set of processors, we must sometimes be concerned with persistence across space. In such systems, it is useful to think of objects that can move from machine to machine and that may even have different representations on different machines.

To summarize, we define persistence as follows:

> Persistence is the property of an object through which its existence transcends time (i.e., the object continues to exist after its creator ceases to exist) and/or space (i.e., the object's location moves from the address space in which it was created).

## 2.4 Applying the Object Model

As we have shown, the object model is fundamentally different from the models embraced by the more traditional methods of structured analysis, structured design, and structured programming. This does not mean that the object model abandons all of the sound principles and experiences of these older methods. Rather, it introduces several novel elements that build on these earlier models. Thus, the object model offers a number of significant benefits that other models simply do not provide. Most importantly, the use of the object model leads us to

construct systems that embody the five attributes of well-structured complex systems noted in Chapter 1: hierarchy, relative primitives (i.e., multiple levels of abstraction), separation of concerns, patterns, and stable intermediate forms. In our experience, there are five other practical benefits to be derived from the application of the object model.

## Benefits of the Object Model

First, the use of the object model helps us to exploit the expressive power of object-based and object-oriented programming languages. As Stroustrup points out, "It is not always clear how best to take advantage of a language such as C++. Significant improvements in productivity and code quality have consistently been achieved using C++ as 'a better C' with a bit of data abstraction thrown in where it is clearly useful. However, further and noticeably larger improvements have been achieved by taking advantage of class hierarchies in the design process. This is often called object-oriented design and this is where the greatest benefits of using C++ have been found" [82]. Our experience has been that, without the application of the elements of the object model, the more powerful features of languages such as Smalltalk, C++, Java, and so forth are either ignored or greatly misused.

Second, the use of the object model encourages the reuse not only of software but of entire designs, leading to the creation of reusable application frameworks [83]. We have found that object-oriented systems are often smaller than equivalent non-object-oriented implementations. Not only does this mean less code to write and maintain, but greater reuse of software also translates into cost and schedule benefits. However, reuse does not just happen. If reuse is not a primary goal of your project, it is unlikely that it will be achieved. Plus, designing for reuse may cost you more when initially implementing the reusable component. The good news is that the initial cost will be recovered in the subsequent uses of that component.

Third, the use of the object model produces systems that are built on stable intermediate forms, which are more resilient to change. This also means that such systems can be allowed to evolve over time, rather than be abandoned or completely redesigned in response to the first major change in requirements.

Chapter 7, Pragmatics, explains further how the object model reduces the risks inherent in developing complex systems. This fourth benefit accrues primarily because integration is spread out across the lifecycle rather than occurring as one major event. The object model's guidance in designing an intelligent separation of concerns also reduces development risk and increases our confidence in the correctness of our design.

Finally, the object model appeals to the workings of human cognition. As Robson suggests, "Many people who have no idea how a computer works find the idea of object-oriented systems quite natural" [84].

## Open Issues

To effectively apply the elements of the object model, we must next address several open issues.

- What exactly are classes and objects?
- How does one properly identify the classes and objects that are relevant to a particular application?
- What is a suitable notation for expressing the design of an object-oriented system?
- What process can lead us to a well-structured object-oriented system?
- What are the management implications of using object-oriented design?

These issues are the themes of the next five chapters.

## Summary

- The maturation of software engineering has led to the development of object-oriented analysis, design, and programming methods, all of which address the issues of programming-in-the-large.
- There are several different programming paradigms: procedure-oriented, object-oriented, logic-oriented, rule-oriented, and constraint-oriented.
- An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.
- Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.
- Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.
- Hierarchy is a ranking or ordering of abstractions.
- Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged or, at the most, may be interchanged only in very restricted ways.

- Concurrency is the property that distinguishes an active object from one that is not active.
- Persistence is the property of an object through which its existence transcends time and/or space.

Chapter 3

# Classes and Objects

Both the engineer and the artist must be intimately familiar with the materials of their trade. Oils versus watercolors, steel versus aluminum, bolts versus nails, object versus classes—each of these materials serves similar functions (e.g., bolts and nails are both fasteners), yet each has its own specific properties and uses. The architect may not know the most efficient way to drive a nail (that is a specific skill of the carpenter), but the architect must understand when it is appropriate to use nails or bolts or glue or welds. To ignore such fundamental considerations can yield disastrous results.

When we use object-oriented methods to analyze or design a complex software system, our basic building blocks are classes and objects. Since we have thus far provided only informal definitions of these two elements, in this chapter we turn to a detailed study of the nature of classes, objects, and their relationships, and along the way we provide several rules of thumb for crafting quality abstractions and mechanisms.

## 3.1 The Nature of an Object

The ability to recognize physical objects is a skill that humans learn at a very early age. A brightly colored ball will attract an infant's attention, but typically, if you hide the ball, the child will not try to look for it; when the object leaves her field of vision, as far as she can determine, it ceases to exist. It is not until near the age of one that a child normally develops what is called the *object concept*, a skill that is of critical importance to future cognitive development. Show a ball to a one-year-old and then hide it, and she will usually search for it even though it is

not visible. Through the object concept, a child comes to realize that objects have a permanence and identity apart from any operations on them [1].

## What Is and What Isn't an Object

In Chapter 1, we informally defined an object as a tangible entity that exhibits some well-defined behavior. From the perspective of human cognition, an object is any of the following:

- A tangible and/or visible thing
- Something that may be comprehended intellectually
- Something toward which thought or action is directed

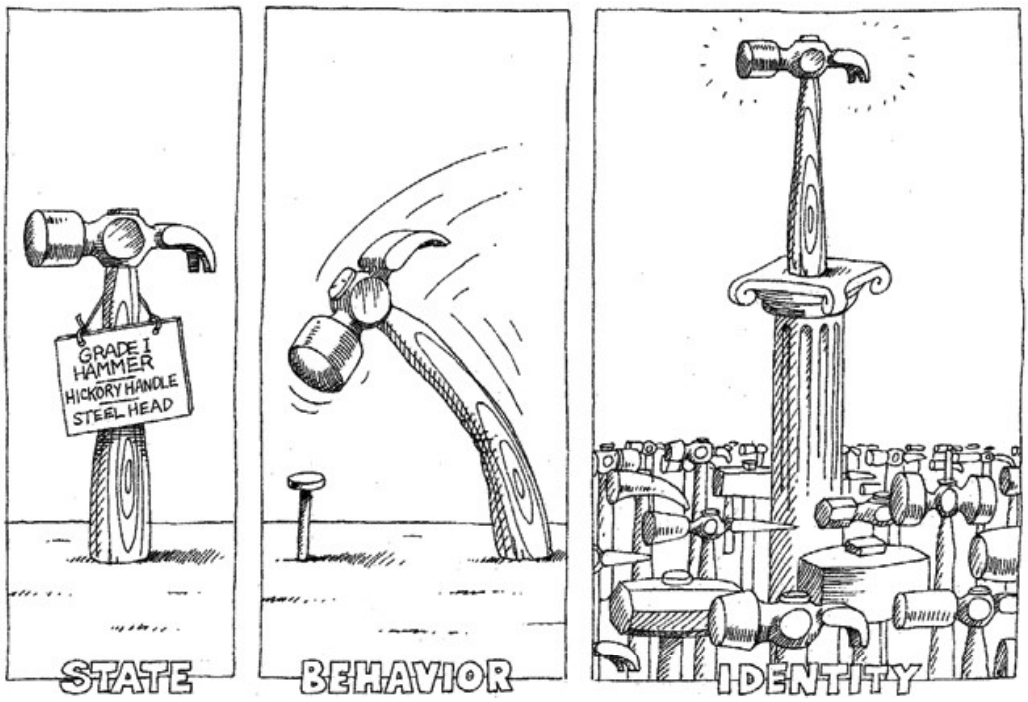
We add to our informal definition the idea that an object models some part of reality and is therefore something that exists in time and space. In software, the term *object* was first formally applied in the Simula language; objects typically existed in Simula programs to simulate some aspect of reality [2].

Real-world objects are not the only kinds of objects that are of interest to us during software development. Other important kinds of objects are inventions of the design process whose collaborations with other such objects serve as the mechanisms that provide some higher-level behavior [3]. Jacobson et al. define *control objects* as "the ones that unite courses of events and thus will carry on communication with other objects" [62]. This leads us to the more refined definition of Smith and Tockey, who suggest that "an object represents an individual, identifiable item, unit, or entity, either real or abstract, with a well-defined role in the problem domain" [4].

Consider for a moment a manufacturing plant that processes composite materials for making such diverse items as bicycle frames and airplane wings. Manufacturing plants are often divided into separate shops: mechanical, chemical, electrical, and so forth. Shops are further divided into cells, and in each cell we have some collection of machines, such as die stamps, presses, and lathes. Along a manufacturing line, we might find vats containing raw materials, which are used in a chemical process to produce blocks of composite materials, and which in turn are formed and shaped to produce bicycle frames, airplane wings, and other end items. Each of the tangible things we have mentioned thus far is an object. A lathe has a crisply defined boundary that separates it from the block of composite material it operates on; a bicycle frame has a crisply defined boundary that distinguishes it from the cell of machines that produced the frame itself.

Some objects may have crisp conceptual boundaries yet represent intangible events or processes. For example, a chemical process in a manufacturing plant

may be treated as an object because it has a crisp conceptual boundary, interacts with certain other objects through a well-ordered collection of operations that unfolds over time, and exhibits a well-defined behavior. Similarly, consider a CAD/CAM system for modeling solids. Where two solids such as a sphere and a cube intersect, they may form an irregular line of intersection. Although it does not exist apart from the sphere or cube, this line is still an object with crisply defined conceptual boundaries.




An object has state, exhibits some well-defined behavior, and has a unique identity.

Some objects may be tangible yet have fuzzy physical boundaries. Objects such as rivers, fog, and crowds of people fit this definition.[1] Just as the person holding a hammer tends to see everything in the world as a nail, so the developer with an object-oriented mindset begins to think that everything in the world is an object. This perspective is a little naive because some things are distinctly not objects. For example, attributes such as beauty or color are not objects, nor are emotions such as love and anger. On the other hand, these things are all potentially properties of other objects. For example, we might say that a man (an object) loves his wife (another object), or that a particular cat (yet another object) is gray.

1. This is true only at a sufficiently high level of abstraction. To a person walking through a fog bank, it is generally futile to distinguish "my fog" from "your fog." However, consider a weather map: A fog bank over San Francisco is a distinctly different object than a fog bank over London.

Thus, it is useful to say that an object is something that has crisply defined boundaries, but this is not enough to guide us in distinguishing one object from another, nor does it allow us to judge the quality of our abstractions. Our experience therefore suggests the following definition.

> An object is an entity that has state, behavior, and identity. The structure and behavior of similar objects are defined in their common class. The terms *instance* and *object* are interchangeable.

We will consider the concepts of state, behavior, and identity in more detail in the sections that follow.

# State

Consider a vending machine that dispenses soft drinks. The usual behavior of such objects is that when someone puts money in a slot and pushes a button to make a selection, a drink emerges from the machine. What happens if a user first makes a selection and then puts money in the slot? Most vending machines just sit and do nothing because the user has violated the basic assumptions of their operation. Stated another way, the vending machine was in a state (of waiting for money) that the user ignored (by making a selection first). Similarly, suppose that the user ignores the warning light that says, "Correct change only," and puts in extra money. Most machines are user-hostile; they will happily swallow the excess money.

In each of these circumstances, we see how the behavior of an object is influenced by its history: The order in which one operates on the object is important. The reason for this event- and time-dependent behavior is the existence of state within the object. For example, one essential state associated with the vending machine is the amount of money currently entered by a user but not yet applied to a selection. Other important properties include the amount of change available and the quantity of soft drinks on hand.

From this example, we may form the following low-level definition.

> The state of an object encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties.

Another property of a vending machine is that it can accept money. This is a static (i.e., fixed) property, meaning that it is an essential characteristic of a vending machine. In contrast, the actual quantity of money accepted at any given moment represents the dynamic value of this property and is affected by the order of operations on the machine. This quantity increases as a user inserts money and then decreases when a product is vended. We say that values are "usually dynamic"

because in some cases values are static. For example, the serial number of a vending machine is a static property and value.

A property is an inherent or distinctive characteristic, trait, quality, or feature that contributes to making an object uniquely that object. For example, one essential property of an elevator is that it is constrained to travel up and down and not horizontally. Properties are usually static because attributes such as these are unchanging and fundamental to the nature of the object. We say "usually static" because in some circumstances the properties of an object may change. For example, consider an autonomous robot that can learn about its environment. It may first recognize an object that appears to be a fixed barrier, only to learn later that this object is in fact a door that can be opened. In this case, the object created by the robot as it builds its conceptual model of the world gains new properties as new knowledge is acquired.

All properties have some value. This value might be a simple quantity, or it might denote another object. For example, part of the state of an elevator might have the value 3, denoting the current floor on which the elevator is located. In the case of the vending machine, its state encompasses many other objects, such as a collection of soft drinks. The individual drinks are in fact distinct objects; their properties are different from those of the machine (they can be consumed, whereas a vending machine cannot), and they can be operated on in distinctly different ways. Thus, we distinguish between objects and simple values: Simple quantities such as the number 3 are "atemporal, unchangeable, and non-instantiated," whereas objects "exist in time, are changeable, have state, are instantiated, and can be created, destroyed, and shared" [6].

The fact that every object has state implies that every object takes up some amount of space, be it in the physical world or in computer memory.

We may say that all objects within a system encapsulate some state and that all of the state within a system is encapsulated by objects. Encapsulating the state of an object is a start, but it is not enough to allow us to capture the full intent of the abstractions we discover and invent during development (refer to Example 3–1, which shows how a simple abstraction may evolve). For this reason, we must also consider how objects behave.

---

**Example 3–1**

Consider an abstraction of an employee record. Figure 3–1 depicts this abstraction using the Unified Modeling Language notation for a class. (For more on the UML notation, see Chapter 5.)

Each part of this abstraction denotes a particular property of our abstraction of an employee. This abstraction is not an object because it does not represent a

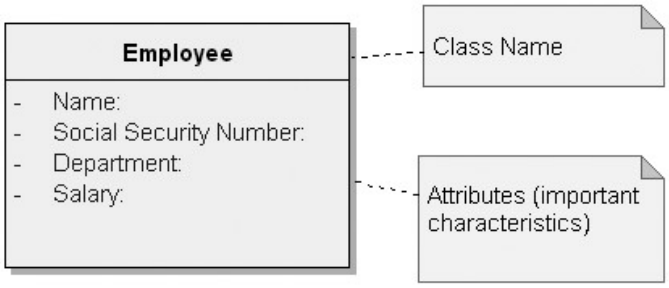



**Figure 3–1** `Employee` Class with Attributes

specific instance. When made specific, we may have, for example, two distinct objects: `Tom` and `Kaitlyn`, each of which takes up some amount of space in memory (see Figure 3–2).


**Figure 3–2** Employee Objects `Tom` and `Kaitlyn`

None of these objects shares its space with any other object, although each of them has the same properties; thus, their states have a common representation.

It is good engineering practice to encapsulate the state of an object rather than expose it. For example, we might change the abstraction (class) as shown in Figure 3–3.

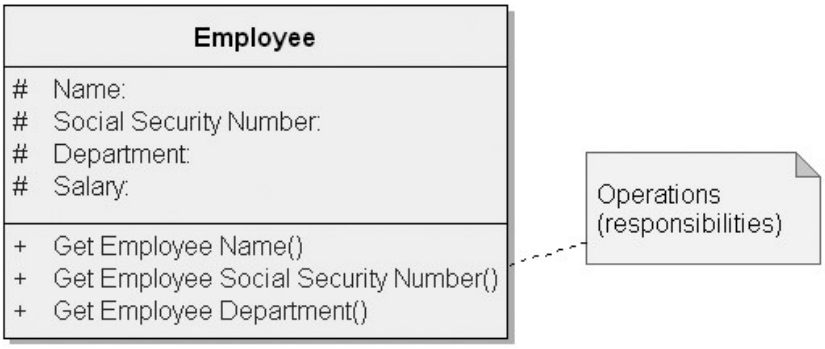



**Figure 3–3** `Employee` Class with Protected Attributes and Public Operations

This abstraction is slightly more complicated than the previous one, but it is superior for a number of reasons. Specifically, its internal representation is hidden (protected, indicated by #) from all other outside clients. If we change its representation, we will have to recompile some code, but semantically, no outside client will be affected by this change (in other words, existing code will not break).

Also, we have captured certain decisions about the problem space by explicitly stating some of the operations (responsibilities) that clients may perform on objects of this class. In particular, we grant all clients the (public, indicated by +) right to retrieve the name, social security number, and department of an employee. We will discuss visibility (i.e., public, protected, private, and package) later in this chapter.

# Behavior

No object exists in isolation. Rather, objects are acted on and themselves act on other objects. Thus, we may say the following:

> Behavior is how an object acts and reacts, in terms of its state changes and message passing.

In other words, the behavior of an object represents its outwardly visible activity.

An operation is some action that one object performs on another in order to elicit a reaction. For example, a client might invoke the operations `append` and `pop` to grow and shrink a queue object, respectively. A client might also invoke the operation `length`, which returns a value denoting the size of the queue object but does not alter the state of the queue itself.

In Java, operations that clients may perform on an object are typically declared as methods. In languages such as C++, which derive from more procedural ancestors, we speak of one object invoking the member function of another. In pure object-oriented languages such as Smalltalk, we speak of one object passing a message to another. Generally, a message is simply an operation that one object performs on another, although the underlying dispatch mechanisms are different. For our purposes, the terms *operation* and *message* are interchangeable.

Message passing is one part of the equation that defines the behavior of an object; our definition for behavior also notes that the state of an object affects its behavior as well. Consider again the vending machine example. We may invoke some operation to make a selection, but the vending machine will behave differently depending on its state. If we do not deposit change sufficient for our selection, the machine will probably do nothing. If we provide sufficient change, the machine

will take our change and then give us our selection (thereby altering its state). Thus, we may say that the behavior of an object is a function of its state as well as the operation performed on it, with certain operations having the side effect of altering the object's state. This concept of side effect thus leads us to refine our definition of state.

> The state of an object represents the cumulative results of its behavior.

Most interesting objects do not have state that is static; rather, their state has properties whose values are modified and retrieved as the object is acted on. The behavior of an object is embodied in the sum of its operations. Next we will discuss operations, how they relate to an object's roles, and how they enable objects to meet their responsibilities.

## Operations

An operation denotes a service that a class offers to its clients. In practice, we have found that a client typically performs five kinds of operations on an object.[2] The three most common kinds of operations are the following:

- Modifier: an operation that alters the state of an object
- Selector: an operation that accesses the state of an object but does not alter the state
- Iterator: an operation that permits all parts of an object to be accessed in some well-defined order

Two other kinds of operations are common; they represent the infrastructure necessary to create and destroy instances of a class.

- Constructor: an operation that creates an object and/or initializes its state
- Destructor: an operation that frees the state of an object and/or destroys the object itself

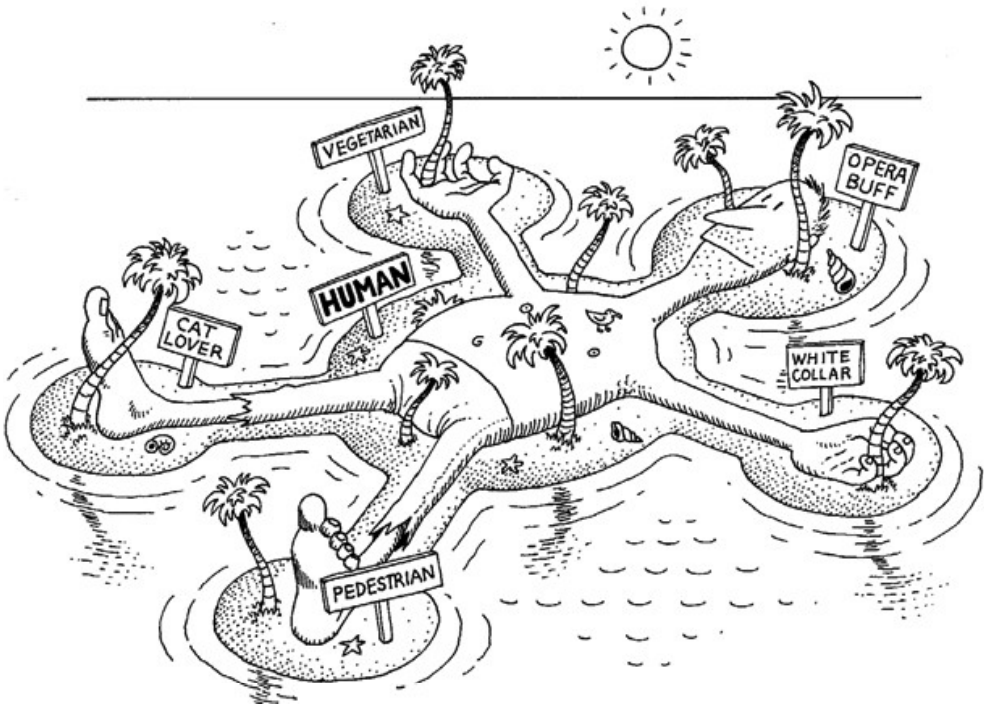
In C++, constructors and destructors are declared as part of the definition of a class, whereas in Java there are constructors, but no destructors. In Smalltalk, such operations are typically part of the protocol of a metaclass (i.e., the class of a class).

---

2. Lippman suggests a slightly different categorization: manager functions, implementer functions, helping functions (all kinds of modifiers), and access functions (equivalent to selectors) [7].

## Roles and Responsibilities

Collectively, all of the methods associated with a particular object comprise its protocol. The protocol of an object thus defines the envelope of an object's allowable behavior and so comprises the entire static and dynamic view of the object. For most nontrivial abstractions, it is useful to divide this larger protocol into logical groupings of behavior. These collections, which thus partition the behavior space of an object, denote the roles that an object can play. A role is a mask that an object wears [8] and so defines a contract between an abstraction and its clients.




Objects can play many different roles.

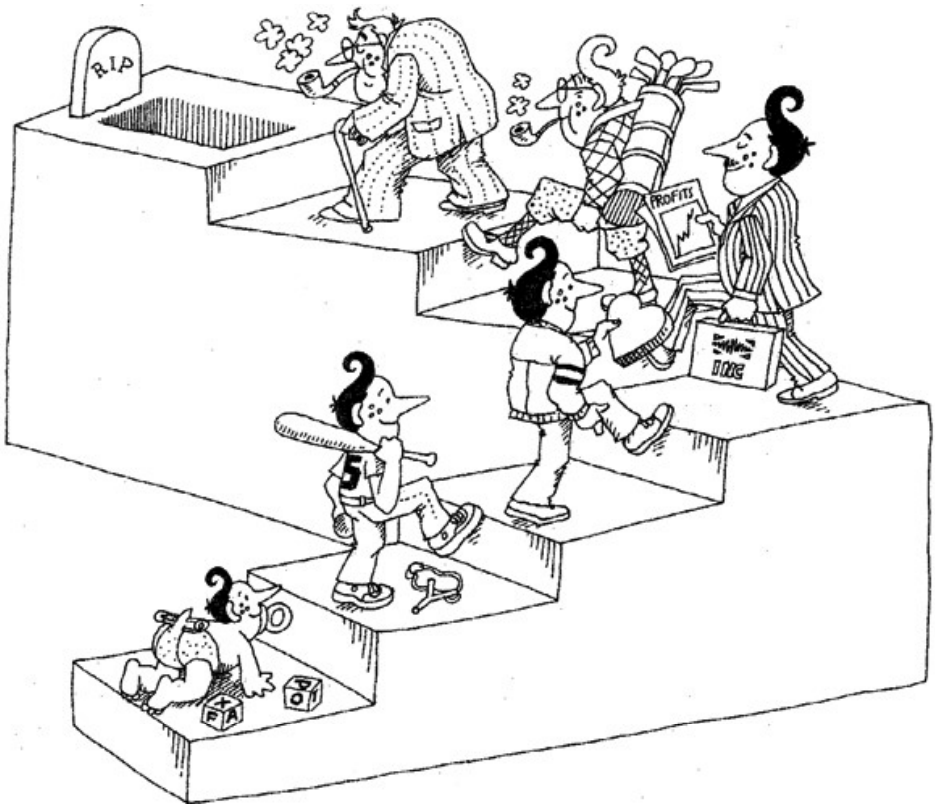
"Responsibilities are meant to convey a sense of the purpose of an object and its place in the system. The responsibilities of an object are all the services it provides for all of the contracts it supports" [9]. In other words, we may say that the state and behavior of an object collectively define the roles that an object may play in the world, which in turn fulfill the abstraction's responsibilities.

Indeed, most interesting objects play many different roles during their lifetime. Consider the following examples [10].

- A bank account may have the role of a monetary asset to which the account owner may deposit or withdraw money. However, to a taxing authority, the account may play the role of an entity whose dividends must be reported on annually.

- To a trader, a share of stock represents an entity with value that may be bought or sold; to a lawyer, the same share denotes a legal instrument to which are attached certain rights.
- In the course of one day, the same person may play the role of mother, doctor, gardener, and movie critic.

The roles played by objects are dynamic yet can be mutually exclusive. In the case of the share of stock, its roles overlap slightly, but each role is static relative to the client that interacts with the share. In the case of the person, her roles are quite dynamic and may change from moment to moment.




Objects play many different roles during their lifetimes.

As we will discuss further in later chapters, we often start our analysis of a problem by examining the various roles that an object plays. During design, we refine these roles by inventing the particular operations that carry out each role's responsibilities.

## *Objects as Machines*

The existence of state within an object means that the order in which operations are invoked is important. This gives rise to the idea that each object is like a tiny,

independent machine [11]. Indeed, for some objects, this event and time ordering of operations is so pervasive that we can best formally characterize the behavior of such objects in terms of an equivalent finite state machine. In Chapter 5, we will show a particular notation for hierarchical finite state machines that we may use for expressing these semantics.

Continuing the machine metaphor, we may classify objects as either active or passive. An active object is one that encompasses its own thread of control, whereas a passive object does not. Active objects are generally autonomous, meaning that they can exhibit some behavior without being operated on by another object. Passive objects, on the other hand, can undergo a state change only when explicitly acted on. In this manner, the active objects in our system serve as the roots of control. If our system involves multiple threads of control, we will usually have multiple active objects. Sequential systems, on the other hand, usually have exactly one active object, such as a main object responsible for managing an event loop that dispatches messages. In such architectures, all other objects are passive, and their behavior is ultimately triggered by messages from the one active object. In other kinds of sequential system architectures (such as transaction-processing systems), there is no obvious central active object, so control tends to be distributed throughout the system's passive objects.

# Identity

Khoshafian and Copeland offer the following definition for identity: "Identity is that property of an object which distinguishes it from all other objects" [12].

They go on to note that "most programming and database languages use variable names to distinguish temporary objects, mixing addressability and identity. Most database systems use identifier keys to distinguish persistent objects, mixing data value and identity." The failure to recognize the difference between the name of an object and the object itself is the source of many kinds of errors in object-oriented programming.

Example 3–2 demonstrates the importance of maintaining the identity of the objects you create and shows how easily the identity can be irrecoverably lost.

---

**Example 3–2**

Consider a class that denotes a display item. A display item is a common abstraction in all GUI-centric systems: It represents the base class of all objects that have a visual representation on some window and so captures the structure and behavior common to all such objects. Clients expect to be able to draw, select, and move display items, as well as query their selection

state and location. Each display item has a location designated by the coordinates x and y.

Let us assume we instantiate a number of `DisplayItem` classes as indicated in Figure 3–4a. Specifically, the manner in which we instantiate these classes sets aside four locations in memory whose names are `item1`, `item2`, `item3`, and `item4`, respectively. Here, `item1` is the name of a distinct `DisplayItem` object, but the other three names each denote a pointer to a `DisplayItem` object. Only `item2` and `item3` actually point to distinct `DisplayItem` objects (because in their declarations we allocated a new `DisplayItem` object); `item4` designates no such object. Furthermore, the names of the objects pointed to by `item2` and `item3` are anonymous: We can refer to these distinct objects only indirectly, via their pointer value.

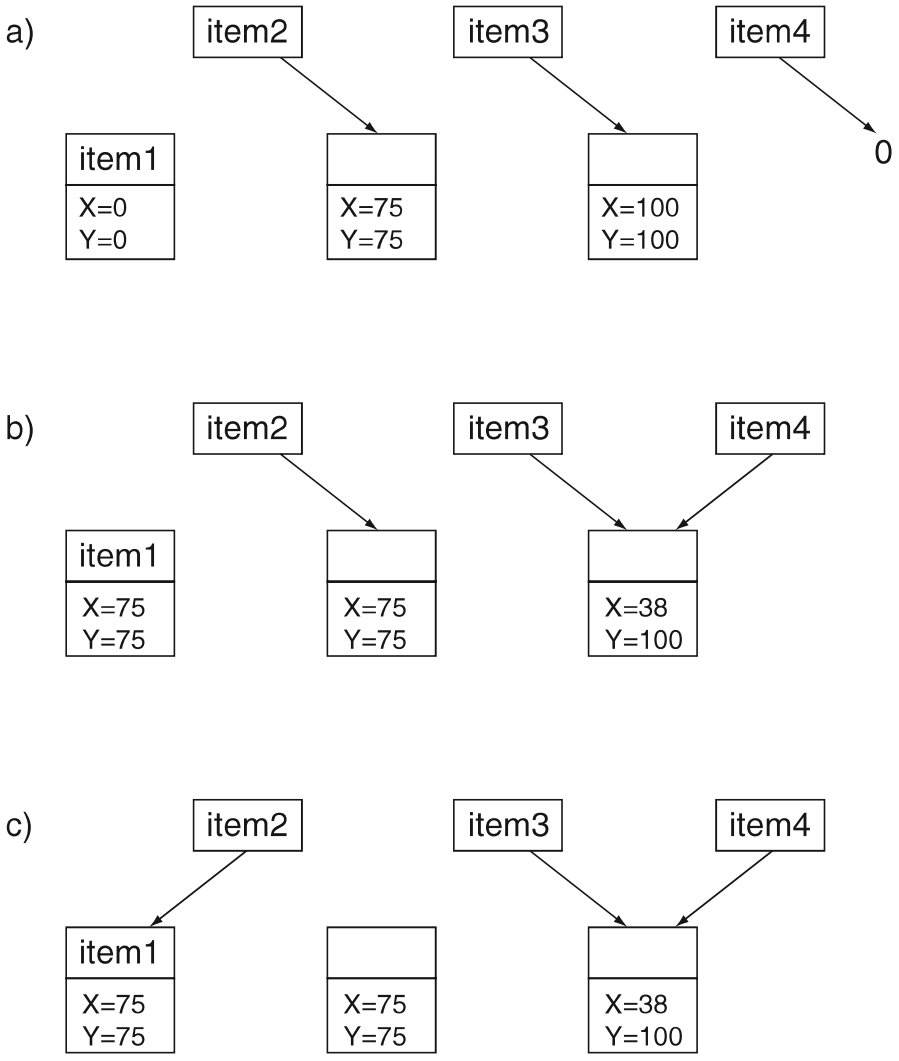



**Figure 3–4** Object Identity

The unique identity (but not necessarily the name) of each object is preserved over the lifetime of the object, even when its state is changed. This is like the Zen question about a river: Is a river the same river from one day to the next, even though the same water never flows through it? For example, let's move `item1`. We can access the object designated by `item2`, get its location, and move `item1` to that same location.

Also, if we equate `item4` to `item3`, we can now reference the object designated by `item3` by using `item4` also. Using `item4` we can then move that object to a new location, say, X=38, Y=100. Figure 3–4b illustrates these results. Here we see that `item1` and the object designated by `item2` both have the same location state and that `item4` now also designates the same object as does `item3`. Notice that we use the phrase "the object designated by `item2`" rather than saying "the object `item2`." The first phrase is more precise, although we will sometimes use these phrases interchangeably.

Although `item1` and the object designated by `item2` have the same state, they represent distinct objects. Also, note that we have changed the state of the object designated by `item3` by operating on it through its new indirect name, `item4`. This is a situation we call *structural sharing*, meaning that a given object can be named in more than one way; in other words, there are aliases to the object. Structural sharing is the source of many problems in object-oriented programming. Failure to recognize the side effects of operating on an object through aliases often leads to memory leaks, memory-access violations, and, even worse, unexpected state changes. For example, if we destroyed the object designated by `item3`, then `item4`'s pointer value would be meaningless; this is a situation we call a *dangling reference*.

Consider also Figure 3–4c, which illustrates the results of modifying the value of the `item2` pointer to point to `item1`. Now `item2` designates the same object as `item1`. Unfortunately, we have introduced a memory leak: The object originally designated by `item2` can no longer be named, either directly or indirectly, and so its identity is lost. In languages such as Smalltalk and Java, such objects will be garbage-collected and their storage reclaimed automatically, but in languages such as C++, their storage will not be reclaimed until the program that created them finishes. Especially for long-running programs, memory leaks such as this are either bothersome or disastrous.[3]

3. Consider the effects of a memory leak in software controlling a satellite or a pacemaker. Restarting the computer in a satellite several million miles away from earth is quite inconvenient. Similarly, the unpredictable occurrence of automatic garbage collection in a pacemaker's software is likely to be fatal. For these reasons, real-time system developers often steer away from the unrestrained allocation of objects on the heap.

# 3.2 Relationships among Objects

An object by itself is intensely uninteresting. Objects contribute to the behavior of a system by collaborating with one another. "Instead of a bit-grinding processor raping and plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires" [13]. For example, consider the object structure of an airplane, which has been defined as "a collection of parts having an inherent tendency to fall to earth, and requiring constant effort and supervision to stave off that outcome" [14]. Only the collaborative efforts of all the component objects of an airplane enable it to fly.

The relationship between any two objects encompasses the assumptions that each makes about the other, including what operations can be performed and what behavior results. We have found that two kinds of object relationships are of particular interest in object-oriented analysis and design, namely:

1. Links
2. Aggregation

## Links

The term *link* derives from Rumbaugh et al., who define it as a "physical or conceptual connection between objects" [16]. An object collaborates with other objects through its links to these objects. Stated another way, a link denotes the specific association through which one object (the client) applies the services of another object (the supplier), or through which one object may navigate to another.

Figure 3–5 illustrates several different links. In this figure, a line between two object icons represents the existence of a link between the two and means that messages may pass along this path. Messages are shown as small directed lines representing the direction of the message, with a label naming the message itself. For example, in Figure 3–5 we show part of a simplified flow control system. This may be controlling the flow of water through a pipe in a manufacturing plant. You can see that the `FlowController` object has a link to a `Valve` object. The `Valve` object has a link to the `DisplayPanel` object in which its status will be displayed. Only across these links may one object send messages to another.

Message passing between two objects is typically unidirectional, although it may occasionally be bidirectional. In our example, the `FlowController` object

invokes operations on the `Valve` object (to change its setting) and the `DisplayPanel` (to change what it displays), but these objects do not themselves operate on the `FlowController` object. This separation of concerns is quite common in well-structured object-oriented systems. Notice also that although message passing is initiated by the client (such as `FlowController`) and is directed toward the supplier (such as the `Valve` object), data may flow in either direction across a link. For example, when `FlowController` invokes the operation `adjust` on the `Valve` object, data (i.e., the setting to change to) flows from the client to the supplier. However, if `FlowController` invokes a different operation, `isClosed`, on the `Valve` object, the result (i.e., whether the valve is in the fully closed position) passes from the supplier to the client.

As a participant in a link, an object may play one of three roles.

1. Controller: This object can operate on other objects but is not operated on by other objects. In some contexts, the terms *active object* and *controller* are interchangeable.
2. Server: This object doesn't operate on other objects; it is only operated on by other objects.
3. Proxy: This object can both operate on other objects and be operated on by other objects. A proxy is usually created to represent a real-world object in the domain of the application.

In the context of Figure 3–5, `FlowController` acts as a controller object, `DisplayPanel` acts as a server object, and `Valve` acts as a proxy. Example 3–3 illustrates how responsibilities can be properly separated across a group of collaborating objects.

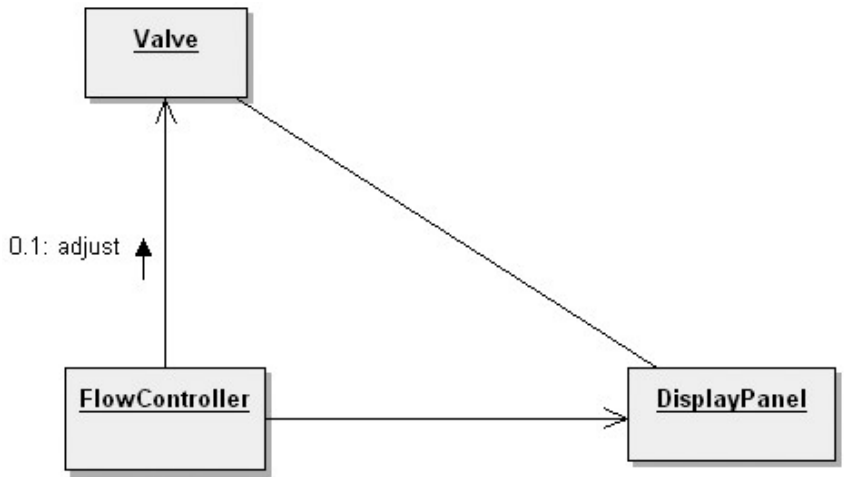



**Figure 3–5** Links

**Example 3–3**

In many different kinds of industrial processes, certain reactions require a temperature ramp, wherein we raise the temperature of some substance, hold it at that temperature for a fixed period, and then let it cool to ambient temperature. Different processes require different profiles: Some objects (such as telescope mirrors) must be cooled slowly, whereas other materials (such as steel) must be cooled rapidly. This abstraction of a temperature ramp has a sufficiently well-defined behavior that it warrants the creation of a class. Thus we provide the class `TemperatureRamp`, which is conceptually a time/temperature mapping (see Figure 3–6).

Actually, the behavior of this abstraction is more than just a literal time/temperature mapping. For example, we might set a temperature ramp that requires the temperature to be 250°F at time 60 (one hour into the temperature ramp) and 150°F at time 180 (three hours into the process), but then we would like to know what the temperature should be at time 120. This requires linear interpolation, which is therefore another behavior (i.e., interpolate) we expect of this abstraction.

One behavior we explicitly do not require of this abstraction is the control of a heater to carry out a particular temperature ramp. Rather, we prefer a greater separation of concerns, wherein this behavior is achieved through the collaboration of three objects: a temperature ramp instance, a heater, and a temperature controller (see Figure 3–6). The operation `process` provides the central behavior of this abstraction; its purpose is to carry out the given temperature ramp for the heater at the given location.

A comment regarding our style: At first glance, it may appear that we have devised an abstraction whose sole purpose is to wrap a functional decomposition inside a class to make it appear noble and object-oriented. The operation `schedule` suggests that this is not the case. Objects of the class `TemperatureController` have sufficient knowledge to determine when a particular profile should be scheduled, so we expose this operation as an additional behavior of our abstraction. In some high-energy industrial processes (such as steel making), heating a substance is a costly event, and it is important to take into account any lingering heat from a previous process, as well as the normal cool-down of an unattended heater. The operation `schedule` exists so that clients can query a `TemperatureController` object to determine the next optimal time to process a particular temperature ramp.

## *Visibility*

Consider two objects, `A` and `B`, with a link between the two. In order for `A` to send a message to `B`, `B` must be visible to `A` in some manner. During our analysis of a problem, we can largely ignore issues of visibility, but once we begin to devise

concrete implementations, we must consider the visibility across links because our decisions here dictate the scope and access of the objects on each side of a link. We will discuss this further later in this chapter.

### *Synchronization*

Whenever one object passes a message to another across a link, the two objects are said to be synchronized. For objects in a completely sequential application, this synchronization is usually accomplished by simple method invocation. However, in the presence of multiple threads of control, objects require more sophisticated message passing in order to deal with the problems of mutual exclusion that can occur in concurrent systems. As we described earlier, active objects embody their own thread of control, so we expect their semantics to be guaranteed in the presence of other active objects. However, when one active object has a link to a passive one, we must choose one of three approaches to synchronization.

1. Sequential: The semantics of the passive object are guaranteed only in the presence of a single active object at a time.
2. Guarded: The semantics of the passive object are guaranteed in the presence of multiple threads of control, but the active clients must collaborate to achieve mutual exclusion.
3. Concurrent: The semantics of the passive object are guaranteed in the presence of multiple threads of control, and the supplier guarantees mutual exclusion.

## Aggregation

Whereas links denote peer-to-peer or client/supplier relationships, aggregation denotes a whole/part hierarchy, with the ability to navigate from the whole (also called the *aggregate*) to its parts. In this sense, aggregation is a specialized kind of association. For example, as shown in Figure 3–6, the object `TemperatureController` has a link to the object `TemperatureRamp` as well as to `Heater`. The object `TemperatureController` is thus the whole, and `Heater` is one of its parts. The notation shown for an aggregation relationship will be further explained in Chapter 5.

By implication, an object that is a part of another object has a link to its aggregate. Across this link, the aggregate may send messages to its parts. Given the object `TemperatureController`, it is possible to find its corresponding `Heater`. Given an object such as `Heater`, it is possible to navigate to its enclosing object (also called its *container*) if and only if this knowledge is a part of the state of `Heater`.

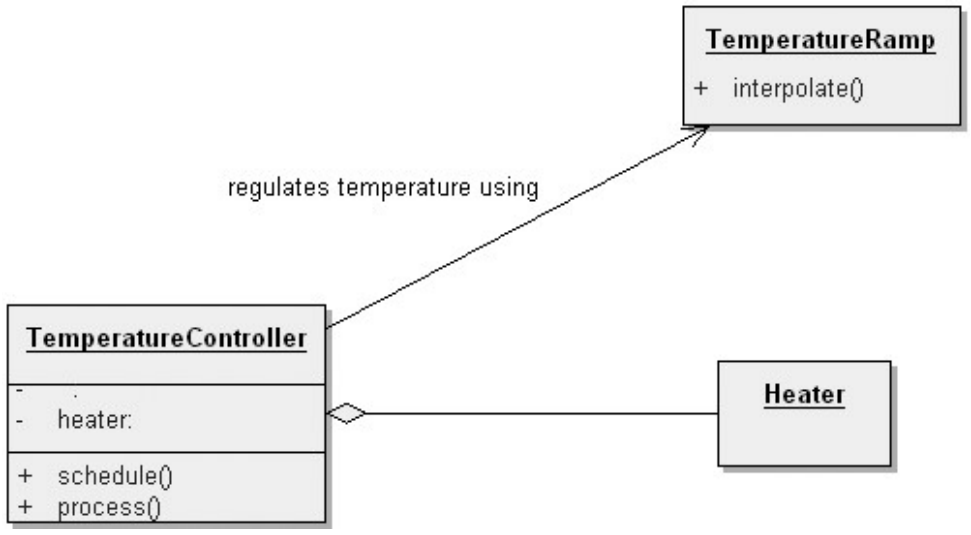



**Figure 3–6**
Aggregation

Aggregation may or may not denote physical containment. For example, an airplane is composed of wings, engines, landing gear, and so on: This is a case of physical containment. On the other hand, the relationship between a shareholder and his or her shares is an aggregation relationship that does not require physical containment. The shareholder uniquely owns shares, but the shares are by no means a physical part of the shareholder. Rather, this whole/part relationship is more conceptual and therefore less direct than the physical aggregation of the parts that form an airplane.

There are clear trade-offs between links and aggregation. Aggregation is sometimes better because it encapsulates parts as secrets of the whole. Links are sometimes better because they permit looser coupling among objects. Intelligent engineering decisions require careful weighing of these two factors.

# 3.3 The Nature of a Class

The concepts of a class and an object are tightly interwoven, for we cannot talk about an object without regard for its class. However, there are important differences between these two terms.

## What Is and What Isn't a Class

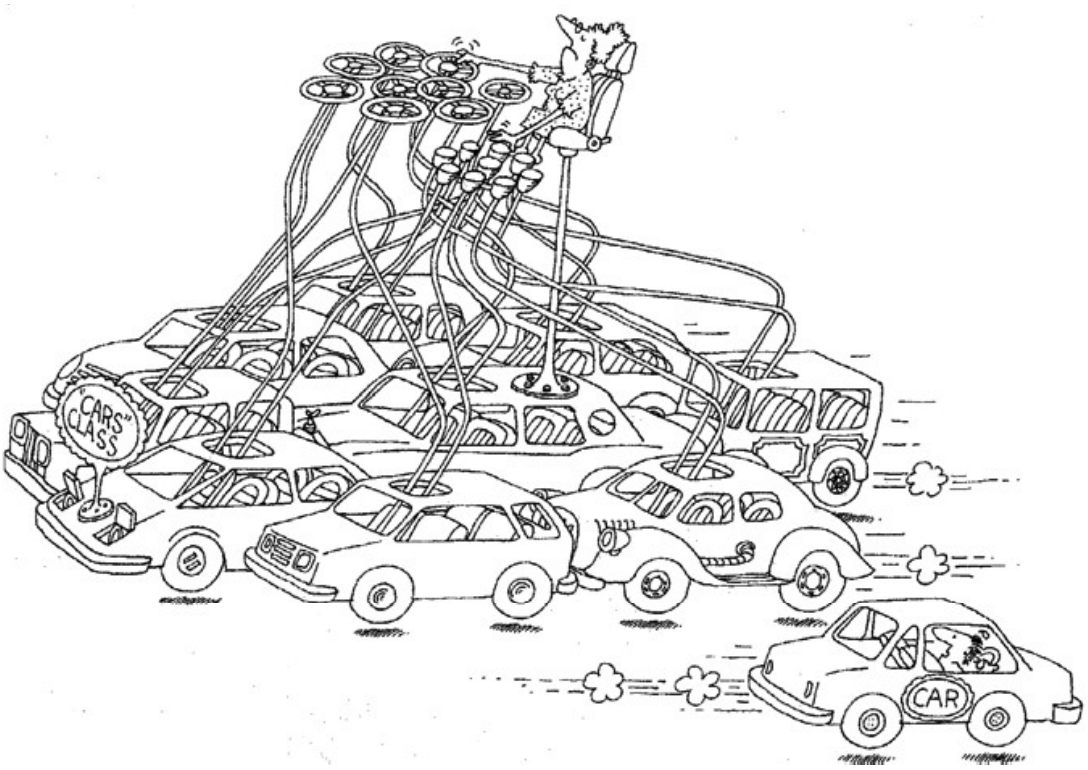
Whereas an object is a concrete entity that exists in time and space, a class represents only an abstraction, the "essence" of an object, as it were. Thus, we may speak of the class `Mammal`, which represents the characteristics common to all mammals. To identify a particular mammal in this class, we must speak of "this mammal" or "that mammal."

In everyday terms, *Webster's Third New International Dictionary* defines a class as "a group, set, or kind marked by common attributes or a common attribute; a group division, distinction, or rating based on quality, degree of competence, or condition" [17].

In the context of object-oriented analysis and design, we define a class as follows:

> A class is a set of objects that share a common structure, common behavior, and common semantics.

A single object is simply an instance of a class.




A class represents a set of objects that share a common structure and a common behavior.

What isn't a class? An object is not a class. Objects that share no common structure and behavior cannot be grouped in a class because, by definition, they are unrelated except by their general nature as objects.

It is important to note that the class—as defined by most programming languages—is a necessary but insufficient vehicle for decomposition. Sometimes abstractions are so complex that they cannot be conveniently expressed in terms of a single class declaration. For example, at a sufficiently high level of abstraction, a GUI framework, a database, and an entire inventory system are all conceptually

individual objects, none of which can be expressed as a single class.[4] Instead, it is far better for us to capture these abstractions as a cluster of classes whose instances collaborate to provide the desired structure and behavior. Stroustrup calls such a cluster a component [18].

## Interface and Implementation

Meyer [19] and Snyder [20] have both suggested that programming is largely a matter of "contracting": The various functions of a larger problem are decomposed into smaller problems by subcontracting them to different elements of the design. Nowhere is this idea more evident than in the design of classes.

Whereas an individual object is a concrete entity that performs some role in the overall system, the class captures the structure and behavior common to all related objects. Thus, a class serves as a sort of binding contract between an abstraction and all of its clients. By capturing these decisions in the interface of a class, a strongly typed programming language can detect violations of this contract during compilation.

This view of programming as contracting leads us to distinguish between the outside view and the inside view of a class. The interface of a class provides its outside view and therefore emphasizes the abstraction while hiding its structure and the secrets of its behavior. This interface primarily consists of the declarations of all the operations applicable to instances of this class, but it may also include the declaration of other classes, constants, variables, and exceptions as needed to complete the abstraction. By contrast, the implementation of a class is its inside view, which encompasses the secrets of its behavior. The implementation of a class primarily consists of the implementation of all of the operations defined in the interface of the class.

We can further divide the interface of a class into four parts:

1. Public: a declaration that is accessible to all clients
2. Protected: a declaration that is accessible only to the class itself and its subclasses

4. One might be tempted to express such abstractions in a single class, but the granularity for reuse and change is too coarse. Having a "fat" interface is bad practice because most clients will want to reference only a small subset of the services provided. Furthermore, changing one part of a huge interface obsolesces every client, even those that don't care about the parts that changed. Nesting classes doesn't eliminate these problems; it only defers them.

3. Private: a declaration that is accessible only to the class itself
4. Package: a declaration that is accessible only by classes in the same package

The detailed semantics of these forms of visibility can vary based on the implementation language used.

### Visibility and Friendship

Different programming languages provide different mixtures of public, protected, private, and package parts, which developers can choose among to establish specific access rights for each part of a class's interface and thereby exercise control over what clients can see and what they can't see (i.e., visibility).

In particular, C++ allows a developer to make explicit distinctions among all four of these different parts.[5] The C++ *friendship* mechanism permits a class to distinguish certain privileged classes that are given the rights to see the class's protected and private parts. Friendships break a class's encapsulation and so, as in life, must be chosen carefully. Java does not have friendship. Instead, Java has a somewhat similar type of visibility called *package access*, where all classes in the same package can access each other. Aside from friendship, public, protected, and private access operate in Java as they do in C++. By contrast, Ada permits declarations to be public or private but not protected. In Smalltalk, all instance variables are private, and all methods are public. In Object Pascal, both fields and operations are public and hence unencapsulated.

The constants and variables that form the representation of a class are known by various terms, depending on the particular language we use. For example, Smalltalk uses the term *instance variable*, Object Pascal and Java use the term *field*, C++ uses the term *data member*. We will use these terms interchangeably to denote the parts of a class that serve as the representation of its instance's state.

The state of an object must have some representation in its corresponding class and so is typically expressed as constant and variable declarations placed in the protected or private part of a class's interface. In this manner, the representation common to all instances of a class is encapsulated, and changes to this representation do not functionally affect any outside clients.

5. The C++ `struct` is a special case, in the sense that a `struct` is a kind of class with all of its elements public.

### Class Lifecycle

We may come to understand the behavior of a simple class just by understanding the semantics of its distinct public operations in isolation. However, the behavior of more interesting classes (such as moving an instance of the class `DisplayItem` or scheduling an instance of the class `TemperatureController`) involves the interaction of their various operations over the lifetime of each of their instances. As described earlier in this chapter, the instances of such classes act as little machines, and since all such instances embody the same behavior, we can use the class to capture these common event- and time-ordered semantics. As we discuss in Chapter 5, we may describe such dynamic behavior for certain interesting classes by using finite state machines.

## 3.4 Relationships among Classes

Consider for a moment the similarities and differences among the following classes of objects: flowers, daisies, red roses, yellow roses, petals, and ladybugs. We can make the following observations.

- A daisy is a kind of flower.
- A rose is a (different) kind of flower.
- Red roses and yellow roses are both kinds of roses.
- A petal is a part of both kinds of flowers.
- Ladybugs eat certain pests such as aphids, which may be infesting certain kinds of flowers.

From this simple example we conclude that classes, like objects, do not exist in isolation. Rather, for a particular problem domain, the key abstractions are usually related in a variety of interesting ways, forming the class structure of our design [21].

We establish relationships between two classes for one of two reasons. First, a class relationship might indicate some sort of sharing. For example, daisies and roses are both kinds of flowers, meaning that both have brightly colored petals, both emit a fragrance, and so on. Second, a class relationship might indicate some kind of semantic connection. Thus, we say that red roses and yellow roses are more alike than are daisies and roses, and daisies and roses are more closely related than are petals and flowers. Similarly, there is a symbiotic connection between ladybugs and flowers: Ladybugs protect flowers from certain pests, which in turn serve as a food source for the ladybug.

In all, there are three basic kinds of class relationships [22]. The first of these is generalization/specialization, denoting an "is a" relationship. For instance, a rose is a kind of flower, meaning that a rose is a specialized subclass of the more general class, flower. The second is whole/part, which denotes a "part of" relationship. Thus, a petal is not a kind of a flower; it is a part of a flower. The third is association, which denotes some semantic dependency among otherwise unrelated classes, such as between ladybugs and flowers. As another example, roses and candles are largely independent classes, but they both represent things that we might use to decorate a dinner table.

# Association

Of these different kinds of class relationships, associations are the most general but also the most semantically weak. The identification of associations among classes is often an activity of analysis and early design, at which time we begin to discover the general dependencies among our abstractions. As we continue our design and implementation, we will often refine these weak associations by turning them into one of the other more concrete class relationships.

## *Semantic Dependencies*

As Example 3–4 suggests, an association only denotes a semantic dependency and does not state the direction of this dependency (unless otherwise stated, an association implies bidirectional navigation, as in our example), nor does it state the exact way in which one class relates to another (we can only imply these semantics by naming the role each class plays in relationship with the other). However, these semantics are sufficient during the analysis of a problem, at which time we need only to identify such dependencies. Through the creation of associations, we come to capture the participants in a semantic relationship, their roles, and their cardinality.

---

**Example 3–4**

For a vehicle, two of our key abstractions include the vehicle and wheels. As shown in Figure 3–7, we may show a simple association between these two classes: the class `Wheel` and the class `Vehicle`. (Arguably, an aggregation would be better.) By implication, this association suggests bidirectional navigation. Given an instance of `Wheel`, we should be able to locate the object denoting its `Vehicle`, and given an instance of `Vehicle`, we should be able to locate all the wheels.

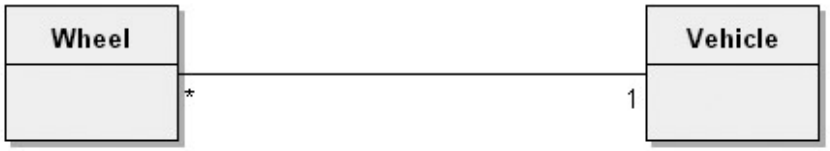



**Figure 3–7** Association

Here we show a one-to-many association: Each instance of `Wheel` relates to one `Vehicle`, and each instance of `Vehicle` may have many `Wheels` (noted by the *).

### *Multiplicity*

Our example introduced a one-to-many association, meaning that for each instance of the class `Vehicle`, there are zero (a boat, which is a vehicle, has no wheels) or more instances of the class `Wheel`, and for each `Wheel`, there is exactly one `Vehicle`. This denotes the multiplicity of the association. In practice, there are three common kinds of multiplicity across an association:

1. One-to-one
2. One-to-many
3. Many-to-many

A one-to-one relationship denotes a very narrow association. For example, in retail telemarketing operations, we would find a one-to-one relationship between the class `Sale` and the class `CreditCardTransaction`: Each sale has exactly one corresponding credit card transaction, and each such transaction corresponds to one sale. Many-to-many relationships are also common. For example, each instance of the class `Customer` might initiate a transaction with several instances of the class `SalesPerson`, and each such salesperson might interact with many different customers. As we will discuss further in Chapter 5, there are variations on these three common forms of multiplicity.

## Inheritance

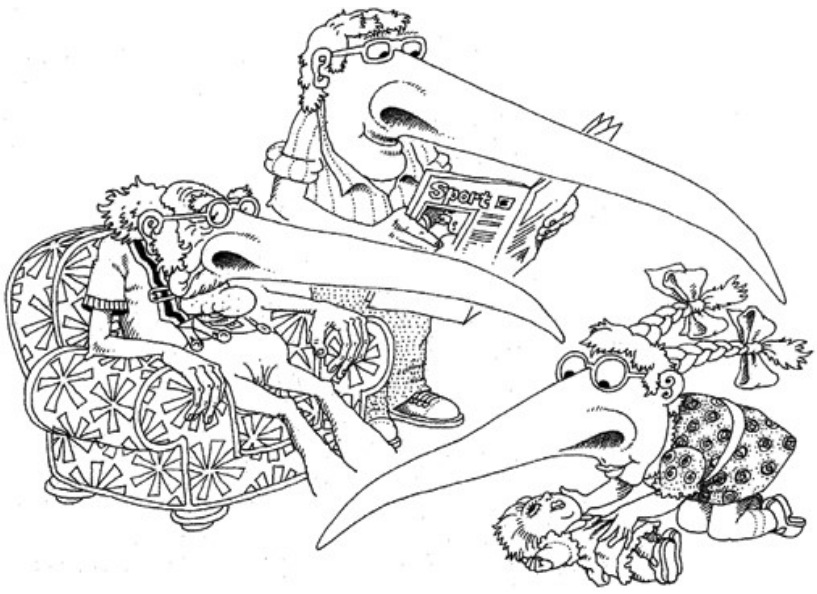
Inheritance, perhaps the most semantically interesting of these concrete relationships, exists to express generalization/specialization relationships. In our experience, however, inheritance is an insufficient means of expressing all of the rich relationships that may exist among the key abstractions in a given problem domain. An alternate approach to inheritance involves a language mechanism called *delegation*, in which objects delegate their behavior to related objects.

### Example 3–5

After space probes are launched, they report back to ground stations with information regarding the status of important subsystems (such as electrical power and propulsion systems) and different sensors (such as radiation sensors, mass spectrometers, cameras, micrometeorite collision detectors, and so on). Collectively, this relayed information is called *telemetry data*. Telemetry data is commonly transmitted as a bitstream consisting of a header, which includes a timestamp and some keys identifying the kind of information that follows, plus several frames of processed data from the various subsystems and sensors. This appears to be a straightforward aggregation of different kinds of data.

This critical data needs to be encapsulated. Otherwise, there is nothing to prevent a client from changing the value of important data such as `timestamp` or `currentPower`. Likewise, the representation of this data is exposed, so if we were to change the representation (e.g., by adding new elements or changing the bit alignment of existing ones), every client would be affected. At the very least, we would certainly have to recompile every reference to this structure. More importantly, such changes might violate the assumptions that clients had made about this representation and cause the logic in our program to break.

Lastly, suppose our analysis of the system's requirements reveals the need for several hundred different kinds of telemetry data, including electrical data that encompassed the preceding information and also included voltage readings from various test points throughout the system. We would find that declaring these additional structures would create a considerable amount of redundancy, in terms of both replicated structures and common functions.




A subclass may inherit the structure and behavior of its superclass.

A slightly better way to capture our decisions would be to declare one class for each kind of telemetry data. In this manner, we could hide the representation of each class and associate its behavior with its data. Still, this approach does not address the problem of redundancy.

A far better solution, therefore, is to capture our decisions by building a hierarchy of classes, in which specialized classes inherit the structure and behavior defined by more generalized classes, as shown in Figure 3–8.

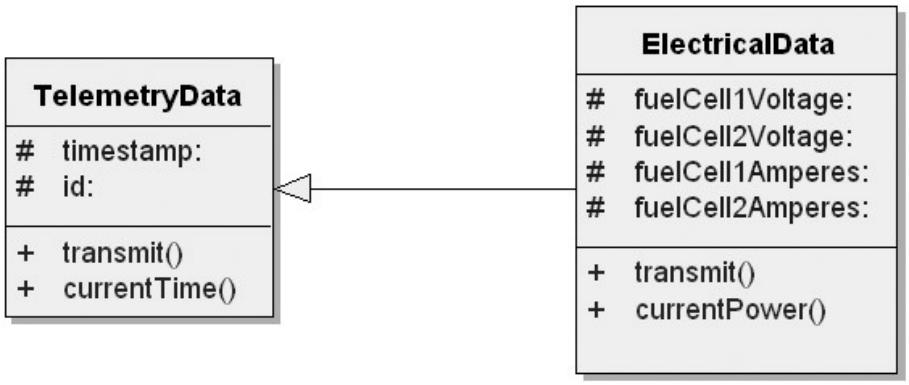



**Figure 3–8** `ElectricalData` Inherits from the Superclass `TelemetryData`

As for the class `ElectricalData`, this class inherits the structure and behavior of the class `TelemetryData` but adds to its structure (the additional voltage data), redefines its behavior (the function `transmit`) to transmit the additional data, and can even add to its behavior (the function `currentPower`, a function to provide the current power level).

## *Single Inheritance*

Simply stated, inheritance is a relationship among classes wherein one class shares the structure and/or behavior defined in one (single inheritance) or more (multiple inheritance) other classes. We call the class from which another class inherits its *superclass*. In Example 3–5, `TelemetryData` is a superclass of `ElectricalData`. Similarly, we call a class that inherits from one or more classes a *subclass*; `ElectricalData` is a subclass of `TelemetryData`. Inheritance therefore defines an "is a" hierarchy among classes, in which a subclass inherits from one or more superclasses. This is in fact the litmus test for inheritance. Given classes `A` and `B`, if `A` is not a kind of `B`, then `A` should not be a subclass of `B`. In this sense, `ElectricalData` is a specialized kind of the more generalized class `TelemetryData`. The ability of a language to support this kind of inheritance distinguishes object-oriented from object-based programming languages.

A subclass typically augments or restricts the existing structure and behavior of its superclasses. A subclass that augments its superclasses is said to use inheritance for extension. For example, the subclass `GuardedQueue` might extend the behavior of its superclass `Queue` by providing extra operations that make instances of this class safe in the presence of multiple threads of control. In contrast, a subclass that constrains the behavior of its superclasses is said to use inheritance for restriction. For example, the subclass `UnselectableDisplayItem` might constrain the behavior of its superclass, `DisplayItem`, by prohibiting clients from selecting its instances in a view. In practice, it is not always so clear whether or not a subclass augments or restricts its superclass; in fact, it is common for a subclass to do both.

Figure 3–9 illustrates the single inheritance relationships deriving from the superclass `TelemetryData`. Each directed line denotes an "is a" relationship. For example, `CameraData` "is a" kind of `SensorData`, which in turn "is a" kind of `TelemetryData`.

This is identical to the hierarchy one finds in a semantic net, a tool often used by researchers in cognitive science and artificial intelligence to organize knowledge about the world [25]. Indeed, as we discuss further in Chapter 4, designing a suitable inheritance hierarchy among abstractions is largely a matter of intelligent classification.

We expect that some of the classes in Figure 3–9 will have instances and some will not. For example, we expect to have instances of each of the most specialized classes (also known as *leaf classes* or *concrete classes*), such as `ElectricalData` and `SpectrometerData`. However, we are not likely to have any instances of the intermediate, more generalized classes, such as `SensorData` or even `TelemetryData`. Classes with no instances are called *abstract classes*. An abstract class is written with the expectation that its subclasses will add to its structure and behavior, usually by completing the implementation of its (typically) incomplete methods.

There is a very real tension between inheritance and encapsulation. To a large degree, the use of inheritance exposes some of the secrets of an inherited class. Practically, this means that to understand the meaning of a particular class, you must often study all of its superclasses, sometimes including their inside views.

Inheritance means that subclasses inherit the structure of their superclass. Thus, in Example 3–5, the instances of the class `ElectricalData` include the data members of the superclass (such as `id` and `timestamp`), as well as those of the more specialized classes (such as `fuelCell1Voltage`, `fuelCell2Voltage`, `fuelCell1Amperes`, and `fuelCell2Amperes`).

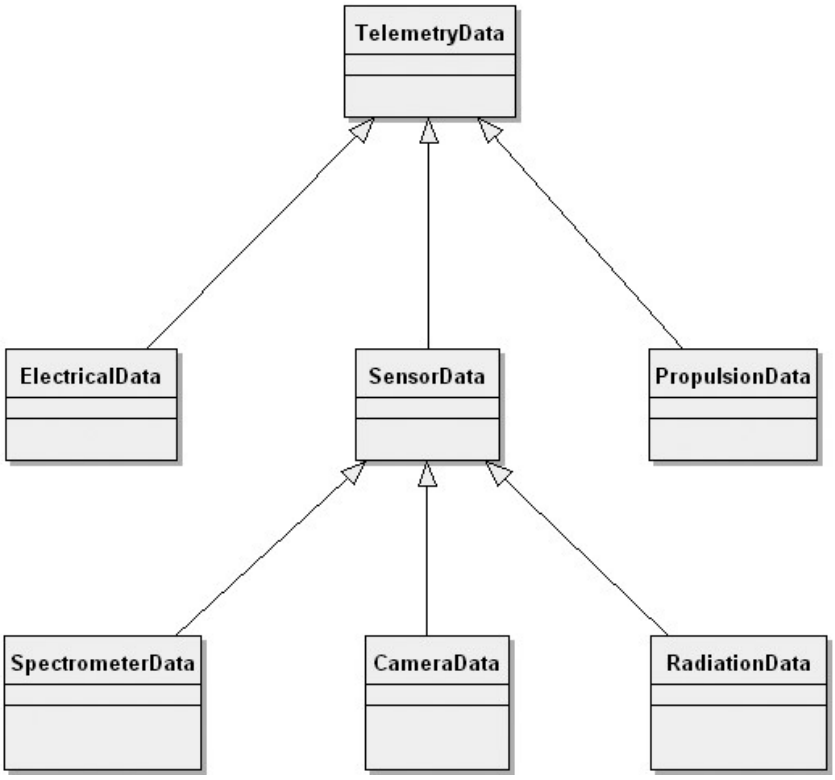



**Figure 3–9** Single Inheritance

Subclasses also inherit the behavior of their superclasses. Thus, instances of the class `ElectricalData` may be acted on with the operations `currentTime` (inherited from its superclass), `currentPower` (defined in the class itself), and `transmit` (redefined in the subclass).

## *Polymorphism*

For the class `TelemetryData`, the function `transmit` may transmit the identifier of the telemetry stream and its timestamp. But the same function for the class `ElectricalData` may invoke the `TelemetryData` `transmit` function and also transmit its voltage and current values.

This behavior is due to polymorphism. In a generalization, such operations are called *polymorphic*. Polymorphism is a concept in type theory wherein a name may denote instances of many different classes as long as they are related by some common superclass. Any object denoted by this name is thus able to respond to some common set of operations in different ways. With polymorphism, an operation can be implemented differently by the classes in the hierarchy. In this manner, a subclass can extend the capabilities of its superclass or override the parent's operation, as `ElectricalData` does in Example 3–5.

The concept of polymorphism was first described by Strachey [29], who spoke of ad hoc polymorphism, by which symbols such as + could be defined to mean different things. We call this concept *overloading*. In C++, one may declare functions having the same names, as long as their invocations can be distinguished by their signatures, consisting of the number and types of their arguments (in C++, unlike Ada, the type of a function's returned value is not considered in overload resolution). By contrast, Java does not permit overloaded operators. Strachey also spoke of parametric polymorphism, which today we simply call polymorphism.

Without polymorphism, the developer ends up writing code consisting of large case or switch statements.[6] Without it, we cannot create a hierarchy of classes for the various kinds of telemetry data; rather, we have to define a single, monolithic variant record encompassing the properties associated with all the kinds of data. To distinguish one variant from another, we have to examine the tag associated with the record.

To add another kind of telemetry data, we would have to modify the variant record and add it to every case statement that operated on instances of this record. This is particularly error-prone and, furthermore, adds instability to the design.

In the presence of inheritance, there is no need for a monolithic type since we may separate different kinds of abstractions. As Kaplan and Johnson note, "Polymorphism is most useful when there are many classes with the same protocols" [30]. With polymorphism, large case statements are unnecessary because each object implicitly knows its own type.

Inheritance without polymorphism is possible, but it is certainly not very useful.

Polymorphism and late binding go hand in hand. In the presence of polymorphism, the binding of a method to a name is not determined until execution. In C++, the developer may control whether a member function uses early or late binding. Specifically, if the method is declared as virtual, then late binding is employed, and the function is considered to be polymorphic. If this virtual declaration is omitted, then the method uses early binding and thus can be resolved at the time of compilation. Java simply performs late binding without the need for an explicit declaration such as `virtual`. How an implementation selects a particular method for execution is described in the sidebar, Invoking a Method.

6. This is in fact the litmus test for polymorphism. The existence of a switch statement that selects an action based on the type of an object is often a warning sign that the developer has failed to apply polymorphic behavior effectively.

## Invoking a Method

In traditional programming languages, invoking a subprogram is a completely static activity. In Pascal, for example, for a statement that calls the subprogram `P`, a compiler will typically generate code that creates a new stack frame, places the proper arguments on the stack, and then changes the flow of control to begin executing the code associated with `P`. However, in languages that support some form of polymorphism, such as Smalltalk and C++, invoking an operation may require a dynamic activity because the class of the object being operated on may not be known until runtime. Matters are even more interesting when we add inheritance to the situation. The semantics of invoking an operation in the presence of inheritance without polymorphism is largely the same as for a simple static subprogram call, but in the presence of polymorphism, we must use a much more sophisticated technique.

Consider the class hierarchy in Figure 3–10, which shows the base class `DisplayItem` along with three subclasses named `Circle`, `Triangle`, and `Rectangle`. `Rectangle` also has one subclass, named `SolidRectangle`. In the class `DisplayItem`, suppose that we define the instance variable `theCenter` (denoting the coordinates for the center of the displayed item), along with the following operations as in our earlier example:

- `draw` Draw the item.
- `move` Move the item.
- `location` Return the location of the item.

The operation `location` is common to all subclasses and therefore need not be redefined, but we expect the operations `draw` and `move` to be redefined since only the subclasses know how to draw and move themselves.

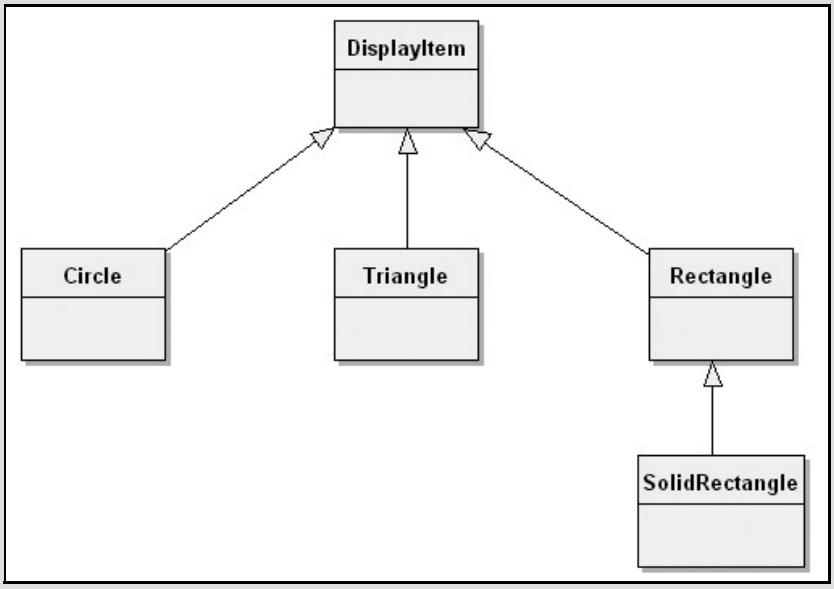



**Figure 3–10** `DisplayItem` Class Diagram

The class `Circle` must include the instance variable `theRadius` and appropriate operations to `set` and `retrieve` its value. For this subclass, the redefined operation `draw` draws a circle of the given radius, centered on `theCenter`. Similarly, the class `Rectangle` must include the instance variables `theHeight` and `theWidth`, along with appropriate operations to `set` and `retrieve` their values. For this subclass, the operation `draw` draws a rectangle with the given height and width, again centered on `theCenter`. The subclass `SolidRectangle` inherits all characteristics of the class `Rectangle` but again redefines the behavior of the operation `draw`. Specifically, the implementation of `draw` for the class `SolidRectangle` first calls `draw` as defined in its superclass `Rectangle` (to draw the outline of the rectangle) and then fills in the shape. The invocation of `draw` demands polymorphic behavior.

Suppose now that we have some client object that wishes to draw all of the subclasses. In this situation, the compiler cannot statically generate code to invoke the proper `draw` operation because the class of the object being operated on is not known until runtime. Let's consider how various object-oriented programming languages deal with this situation.

Because Smalltalk is a typeless language, method dispatch is completely dynamic. When the client sends the message `draw` to an item found in the list, here is what happens.

- The item object looks up the message in its class's message dictionary.
- If the message is found, the code for that locally defined method is invoked.
- If the message is not found, the search for the method continues in the superclass.

This process continues up the superclass hierarchy until the message is found or until we reach the topmost base class, `Object`, without finding the message. In the latter case, Smalltalk ultimately passes the message `doesNotUnderstand` to signal an error.

The key to this algorithm is the message dictionary, which is part of each class's representation and is therefore hidden from the client. This dictionary is created when the class is created and contains all the methods to which instances of this class may respond. Searching for the message is time-consuming; method lookup in Smalltalk takes about 1.5 times as long as a simple subprogram call. All production-quality Smalltalk implementations optimize method dispatch by supplying a cached message dictionary, so that commonly passed messages may be invoked quickly. Caching typically improves performance by 20% to 30% [31].

The operation `draw` defined in the subclass `SolidRectangle` poses a special case. We said that its implementation of `draw` first calls `draw` as

defined in the superclass `Rectangle`. In Smalltalk, we specify a superclass method by using the keyword `super`. Then, when we pass the message `draw` to `super`, Smalltalk uses the same method-dispatch algorithm as mentioned earlier, except that the search begins in the superclass of the object instead of its class.

Studies by Deutsch suggest that polymorphism is not needed about 85% of the time, so message passing can often be reduced to simple procedure calls [32]. Duff notes that in such cases, the developer often makes implicit assumptions that permit an early binding of the object's class [33]. Unfortunately, typeless languages such as Smalltalk have no convenient means for communicating these implicit assumptions to the compiler.

More strongly typed languages such as C++ do let the developer assert such information. Because we want to avoid method dispatch wherever possible but must still allow for the occurrence of polymorphic dispatch, invoking a method in these languages proceeds a little differently than in Smalltalk.

In C++, the developer can decide whether a particular operation is to be bound late by declaring it to be `virtual`; all other methods are considered to be bound early, and thus the compiler can statically resolve the method call to a simple subprogram call.

To handle virtual member functions, most C++ implementations use the concept of a *vtable*, which is defined for each object requiring polymorphic dispatch, when the object is created (and thus when the class of the object is fixed). This table typically consists of a list of pointers to virtual functions. For example, if we create an object of the class `Rectangle`, then the vtable will have an entry for the virtual function `draw`, pointing to the closest implementation of `draw`. If, for example, the class `DisplayItem` included the virtual function `Rotate`, which was not redefined in the class `Rectangle`, then the vtable entry for `Rotate` would point to the implementation of `Rotate` in the class `DisplayItem`. In this manner, runtime searching is eliminated: Referring to a virtual member function of an object is just an indirect reference through the appropriate pointer, which immediately invokes the correct code without searching [34].

## Multiple Inheritance

With single inheritance, each subclass has exactly one superclass. However, as Vlissides and Linton point out, although single inheritance is very useful, "it often forces the programmer to derive from one of two equally attractive classes. This limits the applicability of predefined classes, often making it necessary to duplicate code. For example, there is no way to derive a graphic that is both a circle and a picture; one must derive from one or the other and reimplement the functionality of the class that was excluded" [40].

Consider for a moment how one might organize various assets such as savings accounts, real estate, stocks, and bonds. Savings accounts and checking accounts are both kinds of assets typically managed by a bank, so we might classify both of them as kinds of bank accounts, which in turn are kinds of assets. Stocks and bonds are managed quite differently than bank accounts, so we might classify stocks, bonds, mutual funds, and the like as kinds of securities, which in turn are also kinds of assets.

However, there are many other equally satisfactory ways to classify savings accounts, real estate, stocks, and bonds. For example, in some contexts, it may be useful to distinguish insurable items such as real estate and certain bank accounts (which, in the United States, are insured up to certain limits by the Federal Deposit Insurance Corporation). It may also be useful to identify assets that return a dividend or interest, such as savings accounts, checking accounts, and certain stocks and bonds.

Unfortunately, single inheritance is not expressive enough to capture this lattice of relationships, so we must turn to multiple inheritance.[7] Figure 3–11 illustrates such a class structure. Here we see that the class `Security` is a kind of `Asset` as well as a kind of `InterestBearingItem`. Similarly, the class `BankAccount` is a kind of `Asset`, as well as a kind of `InsurableItem` and `InterestBearingItem`.

Designing a suitable class structure involving inheritance, and especially involving multiple inheritance, is a difficult task. This is often an incremental and iterative process. Two problems present themselves when we have multiple inheritance: How do we deal with name collisions from different superclasses, and how do we handle repeated inheritance?

Name collisions are possible when two or more different superclasses use the same name for some element of their interfaces, such as instance variables and methods. For example, suppose that the classes `InsurableItem` and `Asset` both have attributes named `presentValue`, denoting the present value of the item. Since the class `RealEstate` inherits from both of these classes, what does

7. In fact, this is the litmus test for multiple inheritance. If we encounter a class lattice wherein the leaf classes can be grouped into sets denoting orthogonal behavior (such as insurable and interest-bearing items), and these sets overlap, this is an indication that, within a single inheritance lattice, no intermediate classes exist to which we can cleanly attach these behaviors without violating our abstraction of certain leaf classes by granting them behaviors that they should not have. We can remedy this situation by using multiple inheritance to mix in these behaviors only where we want them.

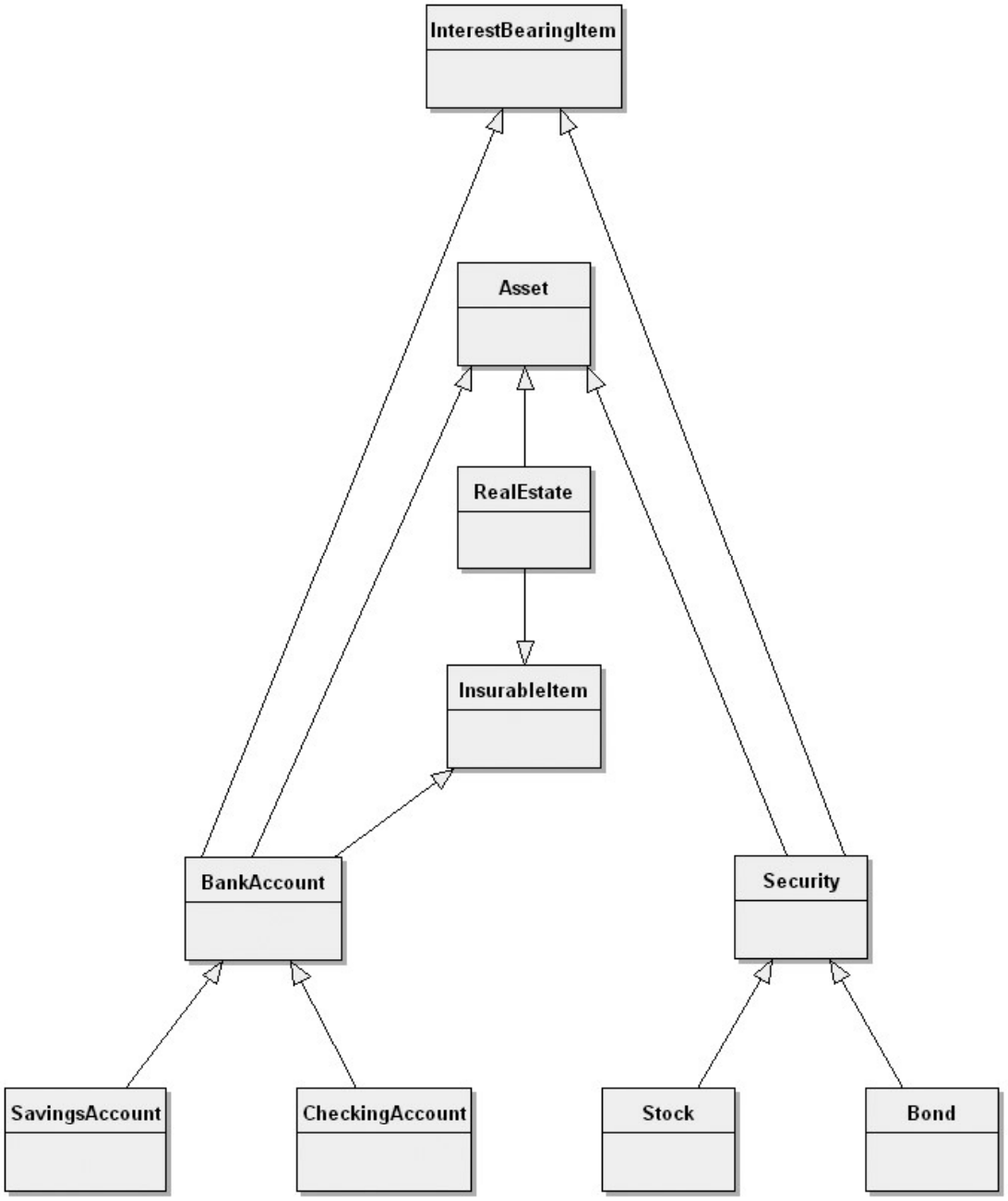



**Figure 3–11** Multiple Inheritance

it mean to inherit two operations with the same name? This in fact is the key difficulty with multiple inheritance: Clashes may introduce ambiguity in the behavior of the multiply inherited subclass.

There are three basic approaches to resolving this kind of clash. First, the language semantics might regard such a clash as illegal and reject the compilation of the class. Second, the language semantics might regard the same name introduced by different classes as referring to the same attribute. Third, the language seman-

tics might permit the clash but require that all references to the name fully qualify the source of its declaration.

The second problem is repeated inheritance, which Meyer describes as follows: "One of the delicate problems raised by the presence of multiple inheritance is what happens when a class is an ancestor of another in more than one way. If you allow multiple inheritance into a language, then sooner or later someone is going to write a class `D` with two parents `B` and `C`, each of which has a class `A` as a parent—or some other situation in which `D` inherits twice (or more) from `A`. This situation is called repeated inheritance and must be dealt with properly" [41]. As an example, suppose that we define the (ill-conceived) `MutualFund` class as a subclass of the classes `Stock` and `Bond`. This class introduces repeated inheritance of the class `Security`, which is a superclass of both `Stock` and `Bond` (see Figure 3–11).

There are various approaches to dealing with the problem of repeated inheritance. First, we can treat occurrences of repeated inheritance as illegal. Second, we can permit duplication of superclasses but require the use of fully qualified names to refer to members of a specific copy. Third, we can treat multiple references to the same class as denoting the same class. Different languages handle this approach differently.

The existence of multiple inheritance gives rise to a style of classes called *mixins*. Mixins derive from the programming culture surrounding the language Flavors: One would combine (mix in) little classes to build classes with more sophisticated behavior. "A mixin is syntactically identical to a regular class, but its intent is different. The purpose of such a class is solely to . . . [add] functions to other flavors [classes]—one never creates an instance of a mixin" [44]. In Figure 3–11, the classes `InsurableItem` and `InterestBearingItem` are mixins. Neither of these classes can stand alone; rather, they are used to augment the meaning of some other class. Thus, we may define a mixin as a class that embodies a single, focused behavior and is used to augment the behavior of some other class via inheritance. The behavior of a mixin is usually completely orthogonal to the behavior of the classes with which it is combined. A class that is constructed primarily by inheriting from mixins and does not add its own structure or behavior is called an *aggregate class*.

# Aggregation

We also need aggregation relationships, which provide the whole/part relationships manifested in the class's instances. Aggregation relationships among classes have a direct parallel to aggregation relationships among the objects corresponding to these classes.

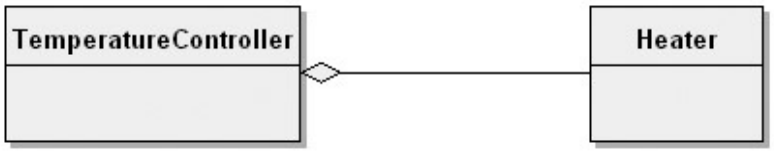



**Figure 3–12** Aggregation

As we show in Figure 3–12, the class `TemperatureController` denotes the whole, and the class `Heater` is one of its parts. This corresponds exactly to the aggregation relationship among the instances of these classes illustrated earlier in Figure 3–6.

## *Physical Containment*

In the case of the class `TemperatureController`, we have aggregation as containment by value, a kind of physical containment meaning that the `Heater` object does not exist independently of its enclosing `TemperatureController` instance. Rather, the lifetimes of these two objects are intimately connected: When we create an instance of `TemperatureController`, we also create an instance of the class `Heater`. When we destroy our `TemperatureController` object, by implication we also destroy the corresponding `Heater` object.

A less direct kind of aggregation is also possible, called *composition*, which is containment by reference. In this case, the class `TemperatureController` still denotes the whole, and an instance of the class `Heater` is still one of its parts, although that part must now be accessed indirectly. Hence, the lifetimes of these two objects are not so tightly coupled as before: We may create and destroy instances of each class independently.

Aggregation asserts a direction to the whole/part relationship. For example, the `Heater` object is a part of the `TemperatureController` object, and not vice versa. Of course, as we described in an earlier example, aggregation need not require physical containment. For example, although shareholders own stocks, a shareholder does not physically contain the owned stocks. Rather, the lifetimes of these objects may be completely independent, although there is still conceptually a whole/part relationship (each share is always a part of the shareholder's assets). Representation of this aggregation can be very indirect.

This is still aggregation, although not physical containment. Ultimately, the litmus test for aggregation is this: If and only if there exists a whole/part relationship between two objects, we must have an aggregation relationship between their corresponding classes.

Multiple inheritance is often confused with aggregation. When considering inheritance versus aggregation, remember to apply the litmus test for each. If you cannot honestly affirm that there is an "is a" relationship between two classes, aggregation or some other relationship should be used instead of inheritance.

### Dependencies

Aside from inheritance, aggregation, and association, there is another group of relationships called *dependencies*. A dependency indicates that an element on one end of the relationship, in some manner, depends on the element on the other end of the relationship. This alerts the designer that if one of these elements changes, there could be an impact to the other. There are many different kinds of dependency relationships (refer to the Object Management Group's latest UML specification for the full list [45]). You will often see dependencies used in architectural models (one system component, or package, is dependent on another) or at the implementation level (one module is dependent on another).

## 3.5 The Interplay of Classes and Objects

Classes and objects are separate yet intimately related concepts. Specifically, every object is the instance of some class, and every class has zero or more instances. For practically all applications, classes are static; therefore, their existence, semantics, and relationships are fixed prior to the execution of a program. Similarly, the class of most objects is static, meaning that once an object is created, its class is fixed. In sharp contrast, however, objects are typically created and destroyed at a furious rate during the lifetime of an application.

### Relationships between Classes and Objects

For example, consider the classes and objects in the implementation of an air traffic control system. Some of the more important abstractions include planes, flight plans, runways, and air spaces. By their very definition, the meanings of these classes and objects are relatively static. They must be static, for otherwise one could not build an application that embodied knowledge of such commonsense facts as that planes can take off, fly, and then land, and that two planes should not occupy the same space at the same time. Conversely, the instances of these classes are dynamic. At a fairly slow rate, new runways are built, and old ones are deactivated. Faster yet, new flight plans are filed, and old ones are filed away. With great frequency, new planes enter a particular air space, and old ones leave.

### The Role of Classes and Objects in Analysis and Design

During analysis and the early stages of design, the developer has two primary tasks:

1. Identify the classes that form the vocabulary of the problem domain
2. Invent the structures whereby sets of objects work together to provide the behaviors that satisfy the requirements of the problem

Collectively, we call such classes and objects the *key abstractions* of the problem, and we call these cooperative structures the *mechanisms* of the implementation.

During these phases of development, the developer must focus on the outside view of these key abstractions and mechanisms. This view represents the logical framework of the system and therefore encompasses the class structure and object structure of the system. In the later stages of design and then moving into implementation, the task of the developer changes: The focus is on the inside view of these key abstractions and mechanisms, involving their physical representation.

## 3.6 On Building Quality Classes and Objects

Ingalls suggests that "a system should be built with a minimum set of unchangeable parts; those parts should be as general as possible; and all parts of the system should be held in a uniform framework" [51]. With object-oriented development, these parts are the classes and objects that make up the key abstractions of the system, and the framework is provided by its mechanisms.

In our experience, the design of classes and objects is an incremental, iterative process. Frankly, except for the most trivial abstractions, we have never been able to define a class exactly right the first time. It takes time to smooth the conceptual jagged edges of our initial abstractions. Of course, there is a cost to refining these abstractions, in terms of recompilation, understandability, and the integrity of the fabric of our system design. Therefore, we want to come as close as we can to being right the first time.

### Measuring the Quality of an Abstraction

How can one know if a given class or object is well designed? We suggest five meaningful metrics:

1. Coupling
2. Cohesion
3. Sufficiency
4. Completeness
5. Primitiveness

Coupling is a notion borrowed from structured design, but with a liberal interpretation it also applies to object-oriented design. Stevens, Myers, and Constantine define coupling as "the measure of the strength of association established by a connection from one module to another. Strong coupling complicates a system since a module is harder to understand, change, or correct by itself if it is highly interrelated with other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules" [52]. A counterexample to good coupling is given by Page-Jones in his description of a modular stereo system in which the power supply is located in one of the speaker cabinets [53].

Coupling with regard to modules still applies to object-oriented analysis and design, but coupling with regard to classes and objects is equally important. However, there is tension between the concepts of coupling and inheritance because inheritance introduces significant coupling. On the one hand, weakly coupled classes are desirable; on the other hand, inheritance—which tightly couples superclasses and their subclasses—helps us to exploit the commonality among abstractions.

The idea of cohesion also comes from structured design. Simply stated, cohesion measures the degree of connectivity among the elements of a single module (and for object-oriented design, a single class or object). The least desirable form of cohesion is coincidental cohesion, in which entirely unrelated abstractions are thrown into the same class or module. For example, consider a class comprising the abstractions of dogs and spacecraft, whose behaviors are quite unrelated. The most desirable form of cohesion is functional cohesion, in which the elements of a class or module all work together to provide some well-bounded behavior. Thus, the class `Dog` is functionally cohesive if its semantics embrace the behavior of a dog, the whole dog, and nothing but the dog.

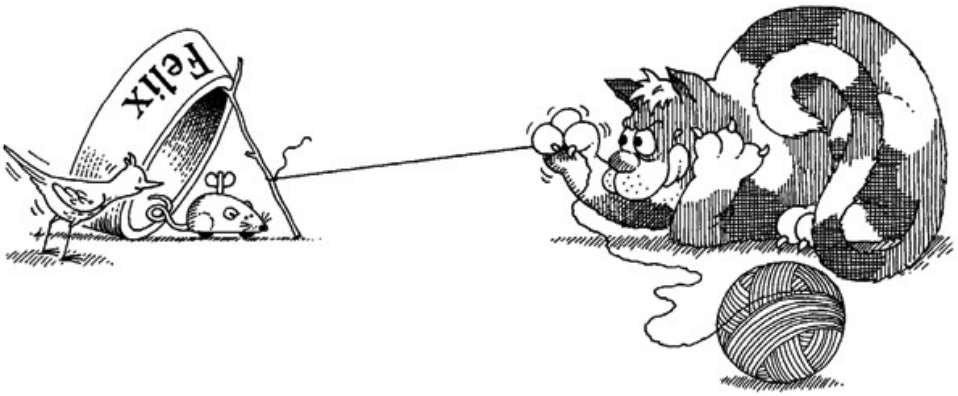
Closely related to the ideas of coupling and cohesion are the criteria that a class or module should be sufficient, complete, and primitive. By sufficient, we mean that the class or module captures enough characteristics of the abstraction to permit meaningful and efficient interaction. To do otherwise renders the component useless. For example, if we are designing the class `Set`, it is wise to include an operation that removes an item from the set, but our wisdom is futile if we neglect an operation that adds an item. In practice, violations of this characteristic are detected very early; such shortcomings rise up almost every time we build a client that must use this abstraction.

By complete, we mean that the interface of the class or module captures all of the meaningful characteristics of the abstraction. Whereas sufficiency implies a minimal interface, a complete interface is one that covers all aspects of the abstraction. A complete class or module is thus one whose interface is general enough to be commonly usable to any client. Completeness is a subjective matter, and it can be overdone. Providing all meaningful operations for a particular abstraction overwhelms the user and is generally unnecessary since many high-level operations can be composed from low-level ones. For this reason, we also suggest that classes and modules be primitive.

Primitive operations are those that can be efficiently implemented only if given access to the underlying representation of the abstraction. Thus, adding an item to a set is primitive because to implement this operation `Add`, the underlying representation must be visible. On the other hand, an operation that adds four items to a set is not primitive because it can be implemented just as efficiently on the more primitive `Add` operation, without having access to the underlying representation. Of course, efficiency is also a subjective measure. An operation is indisputably primitive if we can implement it only through access to the underlying representation. An operation that could be implemented on top of existing primitive operations, but at the cost of significantly more computational resources, is also a candidate for inclusion as a primitive operation.

## Choosing Operations

Crafting the interface of a class or module is plain hard work. Typically, we make a first attempt at the design of a class, and then, as we and others create clients, we find it necessary to augment, modify, and further refine this interface. Eventually, we may discover patterns of operations or patterns of abstractions that lead us to invent new classes or to reorganize the relationships among existing ones.




We often can identify patterns of abstraction, structure, or behavior.

## Functional Semantics

Within a given class, it is our style to keep all operations primitive, so that each exhibits a small, well-defined behavior. We call such methods *fine-grained*. We also tend to separate methods that do not communicate with one another. In this manner, it is far easier to construct subclasses that can meaningfully redefine the behavior of their superclasses. The decision to contract out a behavior to one versus many methods may be made for two competing reasons: Lumping a particular behavior in one method leads to a simpler interface but larger, more complicated methods; spreading a behavior across methods leads to a more complicated interface but simpler methods. As Meyer observes, "A good designer knows how to find the appropriate balance between too much contracting, which produces fragmentation, and too little, which yields unmanageably large modules" [54].

It is common in object-oriented development to design the methods of a class as a whole because all these methods cooperate to form the entire protocol of the abstraction. Thus, given some desired behavior, we must decide in which class to place it. Halbert and O'Brien offer the following criteria to be considered when making such a decision [55].

- Reusability: Would this behavior be more useful in more than one context?
- Complexity: How difficult is it to implement the behavior?
- Applicability: How relevant is the behavior to the type in which it might be placed?
- Implementation knowledge: Does the behavior's implementation depend on the internal details of a type?

We usually choose to declare the meaningful operations that we may perform on an object as methods in the definition of that object's class (or superclass).

## Time and Space Semantics

Once we have established the existence of a particular operation and defined its functional semantics, we must decide on its time and space semantics. This means that we must specify our decisions about the amount of time it takes to complete an operation and the amount of storage it needs. Such decisions are often expressed in terms of best, average, and worst cases, with the worst case specifying an upper limit on what is acceptable.

Earlier, we also mentioned that whenever one object passes a message to another across a link, the two objects must be synchronized in some manner. In the presence of multiple threads of control, this means that message passing is much more than a subprogram-like dispatch. In most of the languages we use, synchronization

among objects is simply not an issue because our programs contain exactly one thread of control, meaning that all objects are sequential. We speak of message passing in such situations as simple because its semantics are most akin to simple subprogram calls. However, in languages that support concurrency, we must concern ourselves with more sophisticated forms of message passing, so as to avoid the problems created if two threads of control act on the same object in unrestrained ways. As we described earlier, objects whose semantics are preserved in the presence of multiple threads of control are either guarded or synchronized objects.

# Choosing Relationships

Choosing the relationships among classes and among objects is linked to the selection of operations. If we decide that object `X` sends message `M` to object `Y`, then either directly or indirectly, `Y` must be accessible to `X`; otherwise, we could not name the operation `M` in the implementation of `X`. By accessible, we mean the ability of one abstraction to see another and reference resources in its outside view. Abstractions are accessible to one another only where their scopes overlap and only where access rights are granted (e.g., private parts of a class are accessible only to the class itself and its friends). Coupling is thus a measure of the degree of accessibility.

## *The Law of Demeter*

One useful guideline in choosing the relationships among objects is called the Law of Demeter, which states that "the methods of a class should not depend in any way on the structure of any class, except the immediate (top-level) structure of their own class. Further, each method should send messages to objects belonging to a very limited set of classes only" [56]. The basic effect of applying this law is the creation of loosely coupled classes, whose implementation secrets are encapsulated. Such classes are fairly unencumbered, meaning that to understand the meaning of one class, you need not understand the details of many other classes.

In looking at the class structure of an entire system, we may find that its inheritance hierarchy is wide and shallow, narrow and deep, or balanced. Class structures that are wide and shallow usually represent forests of free-standing classes that can be mixed and matched [57]. Class structures that are narrow and deep represent trees of classes that are related by a common ancestor [58]. There are advantages and disadvantages to each approach. Forests of classes are more loosely coupled, but they may not exploit all the commonality that exists. Trees of classes exploit this commonality, so that individual classes are smaller than in forests. However, to understand a particular class, it is usually necessary to under-

stand the meaning of all the classes it inherits from or uses. The proper shape of a class structure is highly problem-dependent.

We must make similar trade-offs among inheritance, aggregation, and dependency relationships. For example, should the class `Car` inherit, contain, or use the classes named `Engine` and `Wheel`? In this case, we suggest that an aggregation relationship is more appropriate than an inheritance relationship. Meyer states that between the classes `A` and `B`, "inheritance is appropriate if every instance of `B` may also be viewed as an instance of `A`. The client relationship is appropriate when every instance of `B` simply possesses one or more attributes of `A`" [59]. From another perspective, if the behavior of an object is more than the sum of its individual parts, creating an aggregation relationship rather than an inheritance relationship between the appropriate classes is probably superior.

### *Mechanisms and Visibility*

Deciding on the relationship among objects is mainly a matter of designing the mechanisms whereby these objects interact. The question the developer must ask is simply this: Where does certain knowledge go? For example, in a manufacturing plant, materials (called lots) enter manufacturing cells to be processed. As they enter certain cells, we must notify the room's manager to take appropriate action. We now have a design choice: Is the entry of a lot into a room an operation on the room, an operation on the lot, or an operation on both? If we decide that it is an operation on the room, the room must be visible to the lot. If we decide that it is an operation on the lot, the lot must be visible to the room because the lot must know what room it is in. Lastly, if we consider this to be an operation on both the room and the lot, we must arrange for mutual visibility. We must also decide on some visibility relationship between the room and the manager (and not the lot and the manager); either the manager must know the room it manages, or the room must know of its manager.

## Choosing Implementations

Only after we stabilize the outside view of a given class or object do we turn to its inside view. This perspective involves two different decisions: a choice of representation for a class or object and the placement of the class or object in a module.

### *Representation*

The representation of a class or object should almost always be one of the encapsulated secrets of the abstraction. This makes it possible to change the representation (e.g., to alter the time and space semantics) without violating any of the

functional assumptions that clients may have made. As Wirth wisely states, "The choice of representation is often a fairly difficult one, and it is not uniquely determined by the facilities available. It must always be taken in light of the operations that are to be performed upon the data" [60]. For example, given a class whose objects denote a set of flight-plan information, do we optimize the representation for fast searching or for fast insertion and deletion? We cannot optimize for both, so our choice must be based on the expected use of these objects. Sometimes it is not easy to choose, and we end up with families of classes whose interfaces are virtually identical but whose implementations are radically different, in order to provide different time and space behavior.

One of the more difficult trade-offs when selecting the implementation of a class is between computing the value of an object's state versus storing it as a field. For example, suppose we have the class `Cone`, which includes the method `Volume`. Invoking this method returns the volume of the object. As part of the representation of this class, we are likely to use fields for the height of the cone and the radius of its base. Should we have an additional field in which we store the volume of the object, or should the method `Volume` just calculate it every time [61]? If we want this method to be fast, we should store the volume as a field. If space efficiency is more important to us, we should calculate the value. Which representation is better depends entirely on the particular problem. In any case, we should be able to choose an implementation independently of the class's outside view; indeed, we should even be able to change this representation without its clients caring.

## Packaging

Similar issues apply to the declaration of classes and objects within modules. The competing requirements of visibility and information hiding usually guide our design decisions about where to declare classes and objects. Generally, we seek to build functionally cohesive, loosely coupled modules. Many nontechnical factors influence these decisions, such as matters of reuse, security, and documentation. Like the design of classes and objects, module design is not to be taken lightly. As Parnas, Clements, and Weiss note with regard to information hiding, "Applying this principle is not always easy. It attempts to minimize the expected cost of software over its period of use and requires that the designer estimate the likelihood of changes. Such estimates are based on past experience and usually require knowledge of the application area as well as an understanding of hardware and software technology" [63].

# Summary

- An object has state, behavior, and identity.
- The structure and behavior of similar objects are defined in their common class.
- The state of an object encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties.
- Behavior is how an object acts and reacts in terms of its state changes and message passing.
- Identity is the property of an object that distinguishes it from all other objects.
- A class is a set of objects that share a common structure and a common behavior.
- The three kinds of relationships include association, inheritance, and aggregation.
- Key abstractions are the classes and objects that form the vocabulary of the problem domain.
- A mechanism is a structure whereby a set of objects work together to provide a behavior that satisfies some requirement of the problem.
- The quality of an abstraction may be measured by its coupling, cohesion, sufficiency, completeness, and primitiveness.