CHAPTER 2

# The Object Model

Object-oriented design is built upon a sound engineering foundation, whose elements we collectively call 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 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. These differences arise from the fact that structured design methods build upon structured programming, whereas object-oriented design builds upon object-oriented programming. Unfortunately, object-oriented programming means different things to different people. As Rentsch correctly predicted, "My guess is that object-oriented programming will be in the 1980s what structured programming was in the 1970s. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is" [1].

from the book "Object Oriented Design With Applications" by Grady Booch
published by Benjamin Cummings Publishing Co., ISBN 0-8053-0091-0 (1991)

In this chapter, we will show clearly what object-oriented design is and what it is not, and how it differs from other design methods through its use of the seven elements of the object model.

## 2.1 The Evolution of the Object Model

### Trends in Software Engineering

**The Generations of Programming Languages.** As we look back upon the relatively brief yet colorful history of software engineering, we cannot help but notice two sweeping trends:

- The shift in focus from programming-in-the-small to programming-in-the-large
- 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 trend has been a move away from languages that tell the computer what to do (imperative languages) toward languages that describe the key abstractions in the problem domain (declarative languages).

Wegner has classified some of the more popular high-order programming languages in generations arranged according to the language features they first introduced:

- 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 |

- 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 [2]

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 upon 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 upon 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, 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 their 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. Thus, today we have Ada (a successor to ALGOL 68 and Pascal, with contributions from Simula, Alphard, and CLU), CLOS (which evolved from Lisp, LOOPS, and

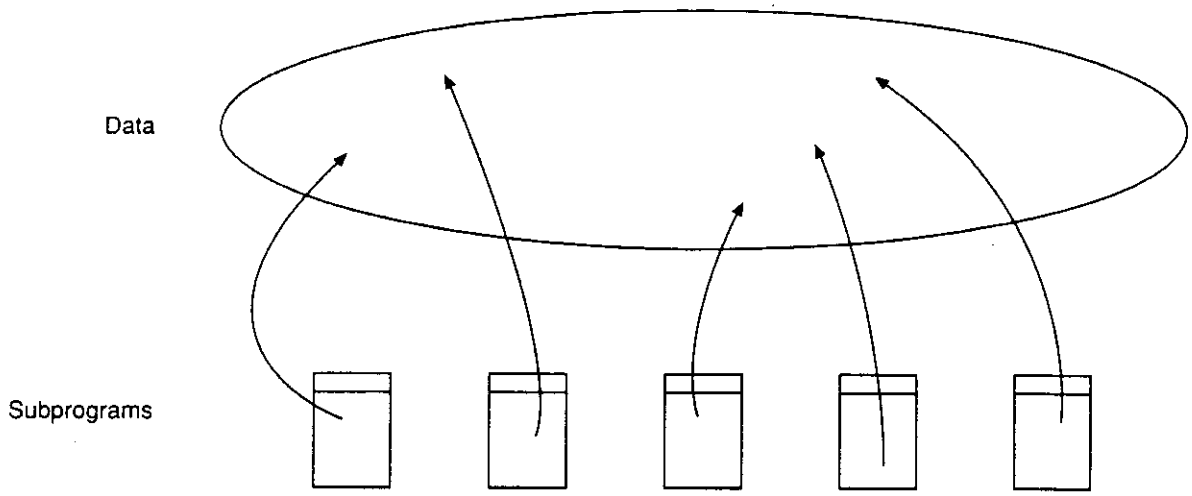



**Figure 2-1**
The Topology of First- and Early Second-Generation Programming Languages

Flavors), and C++ (derived from a marriage of C and Simula). 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 Topology of First- and Early Second-Generation Programming Languages.** To show precisely what we mean, let's look at each generation of programming languages in a slightly different way. In Figure 2-1, we see the topology of most first- and early second-generation programming languages. This topology shows 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. 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 endangering the reliability of the entire system and certainly reducing the overall clarity of the solution.

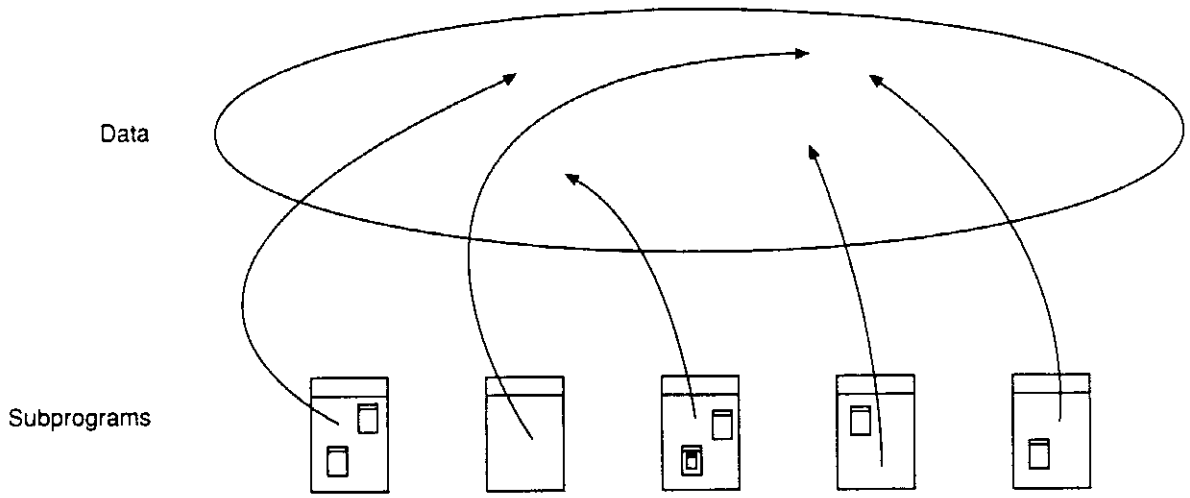



**Figure 2-2**
The Topology of Late Second- and Early Third-Generation Programming Languages

**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]. As Shaw points out, "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 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

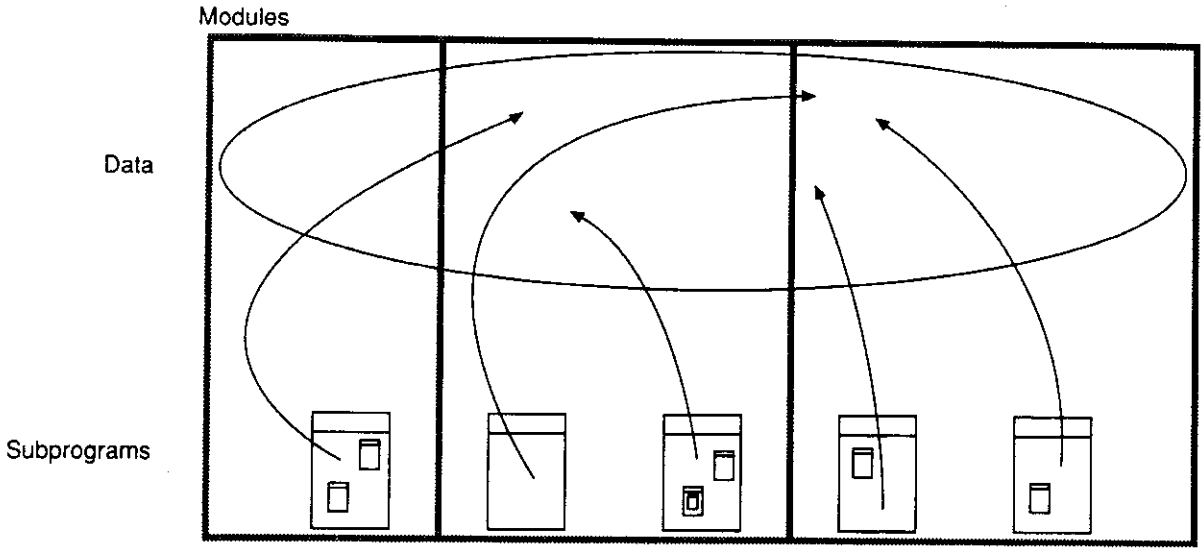



**Figure 2-3**
The Topology of Late Third-Generation Programming Languages

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 logically related subprograms. 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. 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.** The importance of data abstraction to mastering complexity is clearly stated by Shankar: "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

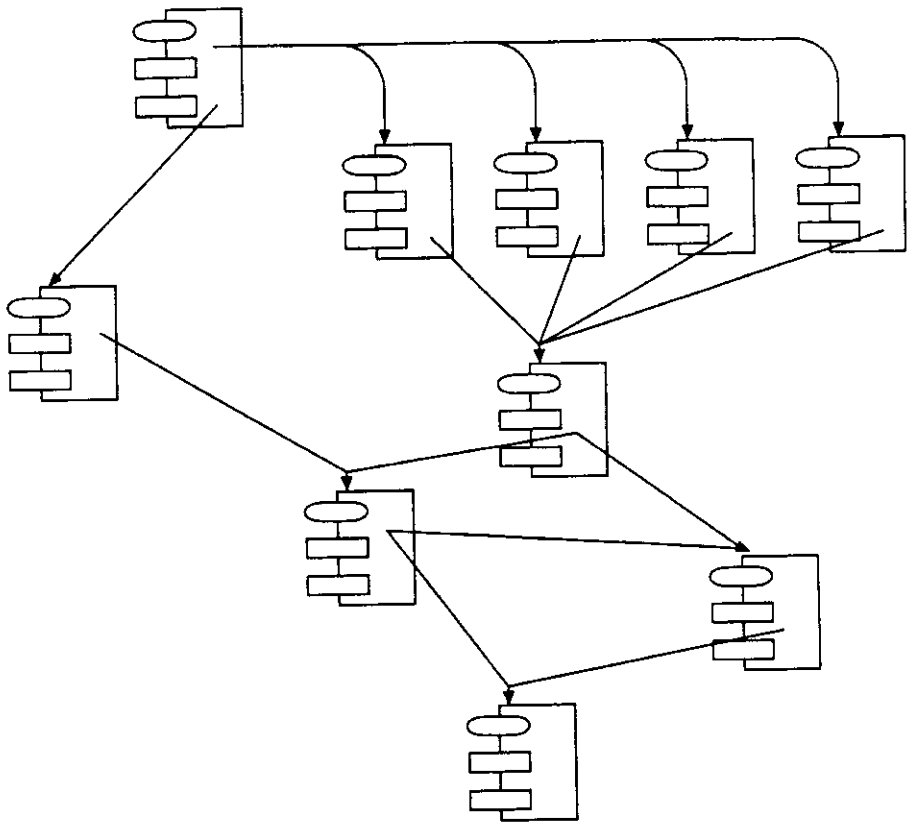


**Figure 2-4**
The Topology of Small- to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

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 during the period of the language generation gap, resulting in the relatively recent development of several languages such as Smalltalk, Object Pascal, C++, CLOS, and Ada. For reasons that we will explain shortly, these languages are called *object-based* or *object-oriented*. Figure 2-4 illustrates the topology of these languages for small- to moderate-sized applications. The physical building block in these 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 classes and objects.

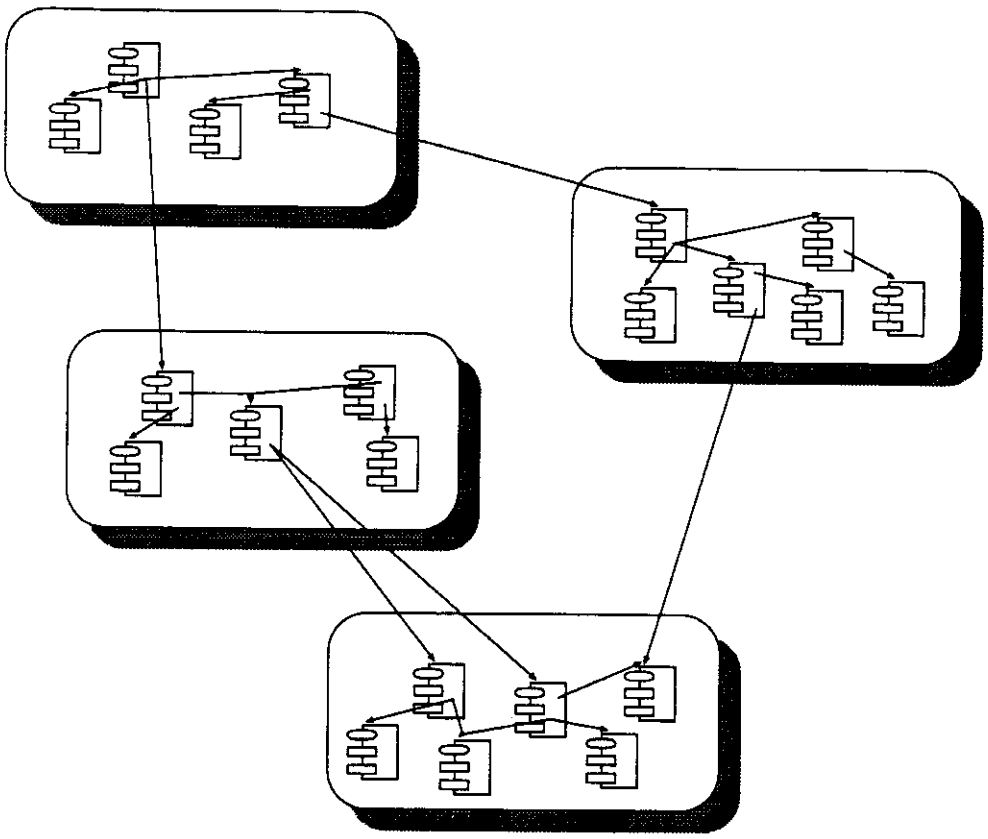


**Figure 2-5**
The Topology of Large Applications Using Object-Based and Object-Oriented Programming Languages

By now we have progressed beyond programming-in-the-large and must cope with programming-in-the-colossal. For very large systems, we find that classes, objects, and modules provide an essential yet insufficient means of decomposition. 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 cooperate 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; its topology is shown in Figure 2-5.

## 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 the sidebar further discusses, the object model has proven to be a unifying concept in computer science, applicable not only to programming languages, but to the design of user interfaces, databases, knowledge bases, 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 design thus represents an evolutionary development, not a revolutionary one; it does not break with advances from the past, but builds upon proven ones. Unfortunately, most programmers today are formally and informally trained only in the principles of structured design. Certainly, many good engineers have developed and deployed countless useful software systems using these 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 Ada 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.

## OOP, OOD, and OOA

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*. To minimize the confusion, let's define what is object-oriented and what is not. The glossary provides a summary of all the terms described here, plus many others.

Bhaskar has observed that 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 upon 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

## Foundations of 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 that have an object-oriented architecture include the Burroughs 5000, the Plessey 250, and the Cambridge CAP [13]; SWARD [14]; the Intel 432 [15], Caltech's COM [16], and the IBM System/38 [17]; the Rational R1000, and the BiiN 40 and 60.

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. More recently, language research has 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. Adding object-oriented programming mechanisms to Pascal has led to the languages Object Pascal, Eiffel, and Ada. Additionally, there are many dialects of Lisp that incorporate the object-oriented features of Simula and Smalltalk, including Flavors, LOOPS, and more recently, the Common Lisp Object System (CLOS). The appendix discusses 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 either in terms of 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 upon 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. 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].

**Object-Oriented Programming.** What then, is object-oriented programming (or *OOP,* as it is sometimes written)? 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: object-oriented programming (1) 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 are related to one another via *inheritance* relationships (the "kind of" 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; we call it *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 "that 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 "kind of" relationships among types, such as 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++, and CLOS are all object-oriented, and Ada is object-based. However, since objects and classes are elements of both kinds of languages, it is 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? We suggest that

> *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.

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. We will occasionally use the acronym *OOD* to designate the particular method of object-oriented design described in this book.

**Object-Oriented Analysis.** The object model has influenced even earlier phases of the software development life cycle. 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 upon the flow of data within a system. Object-oriented analysis (or *OOA*, as it is sometimes called) 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 can 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.

## 2.2 Elements of the Object Model

### Kinds of Programming Paradigms

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, here listed with the kinds of abstractions they employ:

- Procedure-oriented — Algorithms
- Object-oriented — Classes and objects
- Logic-oriented — Goals, often expressed in a predicate calculus
- Rule-oriented — If-then rules
- 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 for the design of a knowledge base. The object-oriented style, from our observations, is best suited to the broadest set of applications, namely, industrial-strength software in which complexity is the dominant issue.

Each of these styles of programming is based upon 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:

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

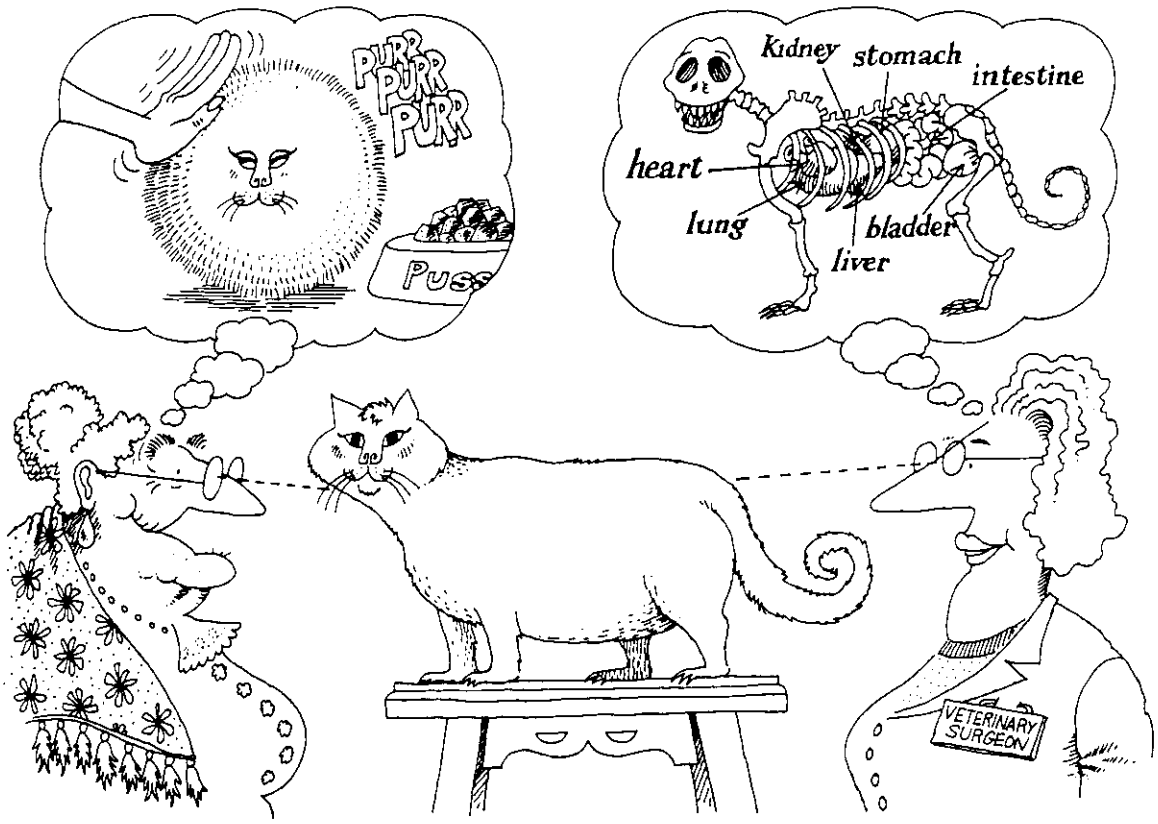
By *major*, we mean that a model without any one of these elements is not object-oriented.

There are three minor elements of the object model:

- Typing
- Concurrency
- 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++, CLOS, 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-based or




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

object-oriented language you are using for implementation. More importantly, you are not likely to have mastered the complexity of the problem at hand.

## Abstraction

**The Meaning of Abstraction.** Abstraction is one of the fundamental ways that we as humans cope with complexity. Hoare suggests 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.

Deciding upon 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.

Seidewitz and Stark suggest that "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 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 together 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. We characterize the behavior of an object by considering the operations that its clients may perform upon it, as well as the operations that it may perform upon other objects. This view forces us to concentrate upon the outside view of an object. We call the entire set of operations that a client may perform upon an object 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.

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, and 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 upon an object (in Smalltalk terminology, when we *send a message* to an object). Thus, invoking an operation upon an object elicits some reaction from the object. What operations we can meaningfully perform upon 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. Our purpose here is to show how we can concretely express abstractions, not so much how we find the right abstractions for the given problem. We defer a complete treatment of this latter topic 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 upon 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, and so we must have sensors for air and water temperature, humidity, light, pH, and nutrient concentrations, among other things. Viewed from the outside, an air 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 then 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 an air temperature sensor is not so much where it is located, but the fact that it has a unique location and identity from all other air temperature sensors. Now we are ready to ask What operations can a client perform upon an air temperature sensor? Our design decision is that a client can calibrate it, as well as ask what the current temperature is.

Let's use Ada to capture these design decisions. For those readers who are not familiar with Ada, or for that matter any of the other object-based and

object-oriented languages we use in this book, the appendix provides a brief overview of each language, with examples. In Ada, we might write the following package specification that captures our abstraction of an air temperature sensor:

```
package Temperature_Sensors is

    type Temperature is delta 0.01 range -10.0 .. 150.0;
    -- temperature in degrees Fahrenheit

    type Location is range 0 .. 63;
    -- a number denoting the location of a sensor

    type Air_Temperature_Sensor is limited private;
    -- the air temperature sensor class

    procedure Initialize (The_Sensor   : in Air_Temperature_Sensor;
                          Its_Location : in Location);
    procedure Calibrate (The_Sensor        : in out Air_Temperature_Sensor;
                         Actual_Temperature : in     Temperature);

    function Current_Temperature (The_Sensor : in Air_Temperature_Sensor)
      return Temperature;

private
    ...
end Temperature_Sensors;
```

This package exports three types, `Temperature`, `Location`, and `Air_Temperature_Sensor`. The type `Temperature` is a fixed point type representing temperature in degrees Fahrenheit. The type `Location` denotes the places where air temperature sensors may be deployed throughout the farm. Lastly, the type `Air_Temperature_Sensor` captures our abstraction of a sensor itself; its representation is hidden in the private part and body of the package.

Because each type represents a class and not an individual object, we must first create an *instance* so that we have something upon which to operate. For example, we might write:

```
with Temperature_Sensors;
use Temperature_Sensors;
...
    Greenhouse_1_Temperature_Sensor : Air_Temperature_Sensor;
    Greenhouse_2_Temperature_Sensor : Air_Temperature_Sensor;
    The_Temperature : Temperature;
begin
    Initialize(Greenhouse_1_Temperature_Sensor, Its_Location => 1);
    Initialize(Greenhouse_2_Temperature_Sensor, Its_Location => 2);
    The_Temperature := Current_Temperature(Greenhouse_1_Temperature_Sensor);
    ...
end;
```

The abstraction we have described thus far is passive; some other object must operate upon an air temperature sensor to determine its value. There is another possible abstraction that may be more or less appropriate depending upon the broader system-design decisions we might make. Rather than the air temperature sensor being passive, we might make it active, so that it is not acted upon but rather acts upon other objects whenever the temperature changes a certain number of degrees. This abstraction is almost the same as our first one, except that we must turn our interface inside out. Thus, we might write the following:

```
package Temperature_Sensors is

    type Temperature is delta 0.01 range -10.0 .. 150.0;
    -- temperature in degrees Fahrenheit

    type Location is range 0 .. 63;
    -- a number denoting the location of a sensor

    generic
        with procedure Temperature_Has_Changed (The_Location    : in Location;
                                                New_Temperature : in Temperature);
        with procedure Temperature_Alarm (The_Location    : in Location;
                                          New_Temperature : in Temperature);
    package Air_Temperature_Sensors is

        type Air_Temperature_Sensor is limited private;
        -- the air temperature sensor class

        procedure Initialize (The_Sensor       : in Air_Temperature_Sensor;
                              Its_Location     : in Location;
                              Lower_Alarm_Limit : in Temperature;
                              Upper_Alarm_Limit : in Temperature);
        procedure Calibrate (The_Sensor        : in out Air_Temperature_Sensor;
                             Actual_Temperature : in     Temperature);

    private
        ...
    end Air_Temperature_Sensors;

end Temperature_Sensors;
```

This package is a bit more complicated than the first, but it captures our new abstraction quite well. The only operations we may perform upon an air temperature sensor object are `Initialize` and `Calibrate`. During its lifetime, each air temperature sensor may itself invoke the operations `Temperature_Has_Changed` or `Temperature_Alarm` to notify some other object that an interesting event has occurred. This package thus shows how in Ada we can describe our design decisions regarding what operations we can perform upon an object, as well as the operations an object can perform upon others.

Let's consider a different abstraction, this time using C++. For each crop, there must be a growing plan that describes how temperature, light, nutrients, and other factors 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. Basically, a growing plan is a table of times versus actions. For example, on day 15 in the lifetime of a certain crop, our growing plan might be to maintain an air temperature of 78°F for 16 hours, turn on the lights for 14 of these hours, and then drop the air 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, yet maintain a slightly acidic pH.

From the perspective of the outside of each growing-plan object, we must be able to establish the details of a plan, modify a plan, and execute a plan. For example, there might be an object that sits at the boundary of the human/machine interface and translates human input into plans. This is the object that establishes the details of a growing plan, and 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 cooperates with other objects to achieve some behavior. Our design decisions about how these objects collaborate define the boundaries of each abstraction and thus the protocol of each object.

Using C++, we might capture our design decisions for a growing plan as follows:

```
typedef int   day;
typedef int   hour;
typedef float temperature;
typedef float ph;
typedef float concentration;
enum  boolean {OFF, ON};

class GrowingPlan {
    ...
public:
    GrowingPlan ();
    GrowingPlan (const GrowingPlan&);
    virtual ~GrowingPlan ();

    virtual void clearThePlan ();
    virtual void establish (day           theDay,
                            hour          theHour,
                            temperature   theTemperature,
                            boolean       lightsOn,
                            ph            thePh,
                            concentration theNutrientConcentration);

    virtual temperature   desiredTemperature (day theDay, hour theHour) const;
    virtual boolean       lightStatus        (day theDay, hour theHour) const;
    virtual ph            desiredPh          (day theDay, hour theHour) const;
    virtual concentration desiredNutrients   (day theDay, hour theHour) const;

};
```

Notice our use of the typedefs. Our style is always to explicitly declare types so that they are expressed in the vocabulary of our problem domain, unless there is some compelling reason to do otherwise. In the interface of the class `GrowingPlan`, we have intentionally left out the private members (designated by the ellipses), because at this point in our design we wish to focus only upon the behavior of the class, not its representation. In C++, members are private unless explicitly asserted otherwise. In the public part, we have exported *constructor* and *destructor member functions* (which provide for the birth and death of an object, respectively), two *modifiers* (the member functions `clearThePlan` and `establish`), and four *selectors*, one to query each of the interesting aspects of a growing plan at a given day and hour of the day. Our style is also to declare each member function as `virtual` unless there is a compelling reason to do otherwise, so that any subclasses of this class can redefine the operation as necessary.

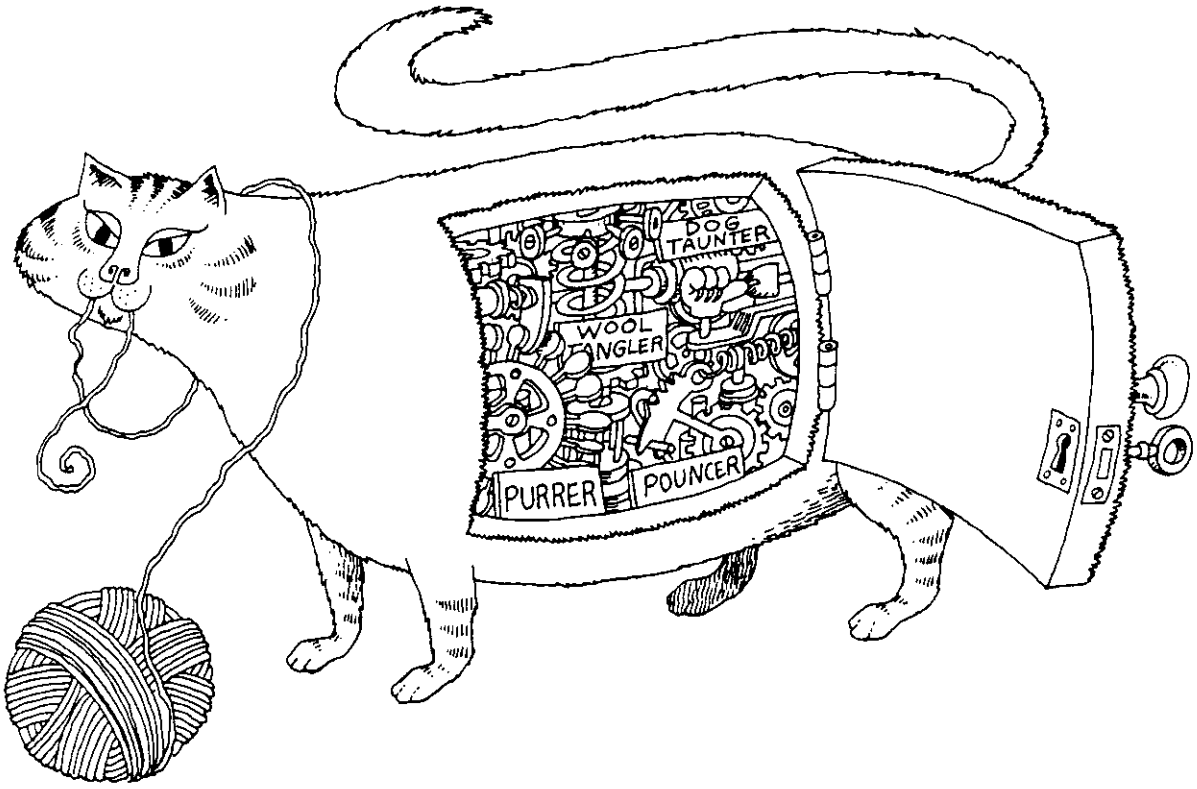
As in our Ada example, the class `GrowingPlan` represents only our abstraction, not an object upon which a client may operate. Therefore at some place in our program, we must create instances of this class.

## Encapsulation

**The Meaning of Encapsulation.** 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. As Ingalls wisely suggests, "No part of a complex system should depend on the internal details of any other part" [48]. Whereas abstraction "helps people to think about what they are doing," encapsulation "allows program changes to be reliably made with limited effort" [49].

Abstraction and encapsulation are complementary concepts: abstraction focuses upon the outside view of an object and *encapsulation* – also known as *information hiding* – prevents clients from seeing its inside view, where the behavior of the abstraction is implemented. In this manner, encapsulation provides explicit barriers among different abstractions. 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 roots or the mitochondria in plant cells. 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 upon a schema that denotes the data's logical view [50]. In both of these cases, objects at higher levels of abstraction are shielded from lower level implementation details.

Liskov goes as far as to suggest that "for abstraction to work, implementations must be encapsulated" [51]. 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. This explicit division of interface/implementation represents a




Encapsulation hides the details of the implementation of an object.

clear separation of concerns: 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. Britton and Parnas call these details the "secrets" of an abstraction [52].

To summarize, we define *encapsulation* as follows:

> *Encapsulation is the process of hiding all of the details of an object that do not contribute to its essential characteristics.*

In practice, one hides the representation of an object, as well as the implementation of its methods.

**Examples of Encapsulation.** To illustrate the use 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, used to maintain a fixed temperature in each greenhouse. 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 upon this object: turn it off, turn it on, and find out if it is running. As is common with our style, we also include metaoperations, namely, constructor and destructor operations that initialize and free instances of this class, respectively. Because our system might have multiple heaters, we use the initialize method to associate each software object with a physical heater. Given these

design decisions, we might write the interface of the class `Heater` in Smalltalk as follows:

```
Heater methodsFor: 'initialize-release'

    initialize: theLocation
    release

Heater methodsFor: 'modifiers'

    turnOff
    turnOn

Heater methodsFor: 'selectors'

    isOn
```

Together with suitable documentation that describes the meaning of each of these operations, this interface represents all that a client needs to know about instances of the class `Heater`.

Turning to the inside view of this class, we have an entirely different perspective. One reasonable implementation decision 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 serial ports from the computer. Sending a character with all bits set to one might turn on the heater, and sending a character with all bits set to zero might turn off the heater. We can thus complete the implementation of the class `Heater` as follows:

```
Object subclass: #Heater
    instanceVariableNames: 'thePort isOn'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Hydroponics Gardening System'

initialize: theLocation
    "Initialize the heater device driver by opening an RS232 port
    associated with the given location. theLocation is expected to
    be of the class Location."

    thePort ← RS232Port open: theLocation.
    isOn ← false

release
    "Release the RS232 port associated with this heater."

    thePort release.
    thePort ← nil

isOn
    "Return true if the heater is on, false otherwise."

    ↑isOn
```

```
turnOff
    "Turn off the heater by writing a character with all bits reset
    to the RS232 port."

    | aString |
    aString ← String new: 1.
    aString at: 1 put: (Character value: 0).
    thePort sendBuffer: aString.
    isOn ← false.
    aString release

turnOn
    "Turn on the heater by writing a character with all bits set
    to the RS232 port."

    | aString |
    aString ← String new: 1.
    aString at: 1 put: (Character value: 255).
    thePort sendBuffer: aString.
    isOn ← true.
    aString release
```

The two instance variables (`thePort` and `isOpen`) form the representation of this class which, according to the rules of Smalltalk, are encapsulated. If a developer writes code outside of this class that references these variables, Smalltalk refuses to accept the code by responding with an error message.

Suppose that for whatever reason the hardware architecture of our system changed, and its designers decided to use memory-mapped I/O instead of serial communication lines. We would not need to change the interface of this class; we would only need to modify its implementation. Because of Smalltalk's obsolescence rules, we would have to recompile this class and the closure of its clients, but because the functional behavior of this abstraction would not change, we would not have to modify any code that used this class unless a particular client depended upon the time or space characteristics of the original implementation (which would be highly undesirable and so very unlikely, in any case).

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 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 then storing certain data. This ability to change the representation of an abstraction without disturbing any of its clients is the essential benefit of encapsulation.

Ideally, attempts to access the underlying representation of an object should be detected at the time a client's code is compiled. How a particular language should address this matter is debated with great religious fervor in the object-oriented programming language community. As we have seen, Smalltalk prevents a client from directly accessing the instance variables of another class; violations are detected at the time of compilation. On the other hand, Object

Pascal does not encapsulate the representation of a class, so there is nothing in the language that prevents clients from referencing the fields of another object. CLOS takes an intermediate position, giving the developer explicit control over encapsulation. Each *slot* may have one of the slot options `:reader`, `:writer`, or `:accessor`, which grant a client read access, write access, or read/write access, respectively. If none of these options are used, then the slot is fully encapsulated. C++ offers even more flexible control over the visibility of member objects and member functions. Specifically, members may be placed in the public, private, or protected parts of a class. Members declared in the public parts are visible to all clients; members declared in the private parts are fully encapsulated; and members declared in the protected parts are visible only to the class itself and its subclasses. C++ also supports the notion of *friends*: cooperative classes that are permitted to see each other's private parts.

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" [53]. Of course, nothing in any of the programming languages we use in this book 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. In practice, there are times when one must study the implementation of a class to really understand its meaning, especially if the external documentation is lacking.

## Modularity

**The Meaning of Modularity.** As Myers observes, "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" [54]. In some languages, such as Smalltalk, there is no concept of a module, and so the class forms the only physical unit of decomposition. In many others, including Object Pascal, C++, CLOS, 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.

Liskov states that "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


Modularity packages abstractions into discrete units.

modules are the assumptions which the modules make about each other' " [55]. 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. As with encapsulation, particular languages support modularity in diverse ways. For example, modules in C++ are nothing more than separately compiled files. The traditional practice in the C/C++ community is to place module interfaces in files named with a *.h* suffix; these are called *header files*. Module implementations are placed in files named with a *.c* suffix. Dependencies among files can then be asserted using the #include macro. This approach is entirely one of convention; it is neither required nor enforced by the language itself. Object Pascal is a little more formal about the matter. In this language, the syntax for *units* (its name for modules) distinguishes between module interface and implementation. Dependencies among units may be asserted only in a module's interface. Ada goes one step further. A `package` (its name for modules) has two parts, the package specification and the package body. Unlike Object Pascal, Ada allows connections among modules to be asserted separately in the specification and body of a package. Thus, it is possible for a package body to depend upon modules that are otherwise not visible to the package's specification.

Deciding upon the right set of modules for a given problem is almost as hard a problem as deciding upon 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" [56].

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 board-level computer. NAND, NOR, and NOT gates might be used to construct the necessary logic, but these gates must be physically packaged in standard integrated circuits, such as a 7400, 7402, or 7404. 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 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, like that described in Chapter 12, 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 [57]. 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 from the design's logical structure, 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" [58]. 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 re-

compile the module interface, its body, all other modules that depend upon this interface, the modules that depend upon 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 many hours of recompilation. Obviously, a manager cannot often afford to allow this. For this reason, a module's interface should be as narrow as possible, yet still satisfy the needs of all using modules. Our style is to hide as much as we can in the implementation of a module. Incrementally shifting declarations from a module 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. Parnas, Clements, and Weiss offer the following guidance: "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" [59]. 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 chose 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, and 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 – and 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 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 guidelines 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 *entirely 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 that instead of building some special-purpose hardware, we decide to use a commercially available workstation for the user interface. At this workstation, an operator could create new growing plans, modify old ones, and follow the progress of currently active ones. Because Object Pascal is available on a variety of platforms, we might choose to use it to implement this part of the system.

One of the key abstractions here is that of a growing plan. We might therefore create a module called `UGrowingPlans`, whose purpose is to collect all of the classes associated with individual growing plans. In Object Pascal, we might write the framework of this unit as follows:

```
unit UGrowingPlans; interface
    ...
implementation
    {$I UGrowingPlans.incl.p}
end.
```

The ellipses mark the location of the declarations that must be exposed to other units. The implementations of these classes, objects, and free subprograms then appear in the implementation of this module, the unit we named `UGrowingPlans.incl.p`.

We might also define a module called `UGardeningDialogs`, whose purpose is to collect all of the code associated with dialog boxes. This unit most likely depends upon the classes declared in the interface of `UGrowingPlans`, and so we might write its framework as follows:

```
unit UGardeningDialogs; interface

uses
    MemTypes, QuickDraw, OSIntf, ToolIntf, PackIntf, CursorCtl,
    UMAUtil, UViewCoords, UFailure, UMemory, UMenuSetup,
    UObject, UList, UAssociation, UMacApp,
    UGrowingPlans;
    ...
implementation
    {$I UGardeningDialogs.incl.p}
end.
```

This unit implementation requires a number of lower level interfaces, and so its interface must import several other units in addition to `UGrowingPlans`.

Our design might include many other units, such as `UGardeningCommands`, `UGardeningViews`, `UGardeningDocuments`, and `UGardeningApplication`, each of which imports the interface of lower level units. Ultimately, we must define some main program from which we can invoke this application from the operating system. 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" [60].

## Hierarchy

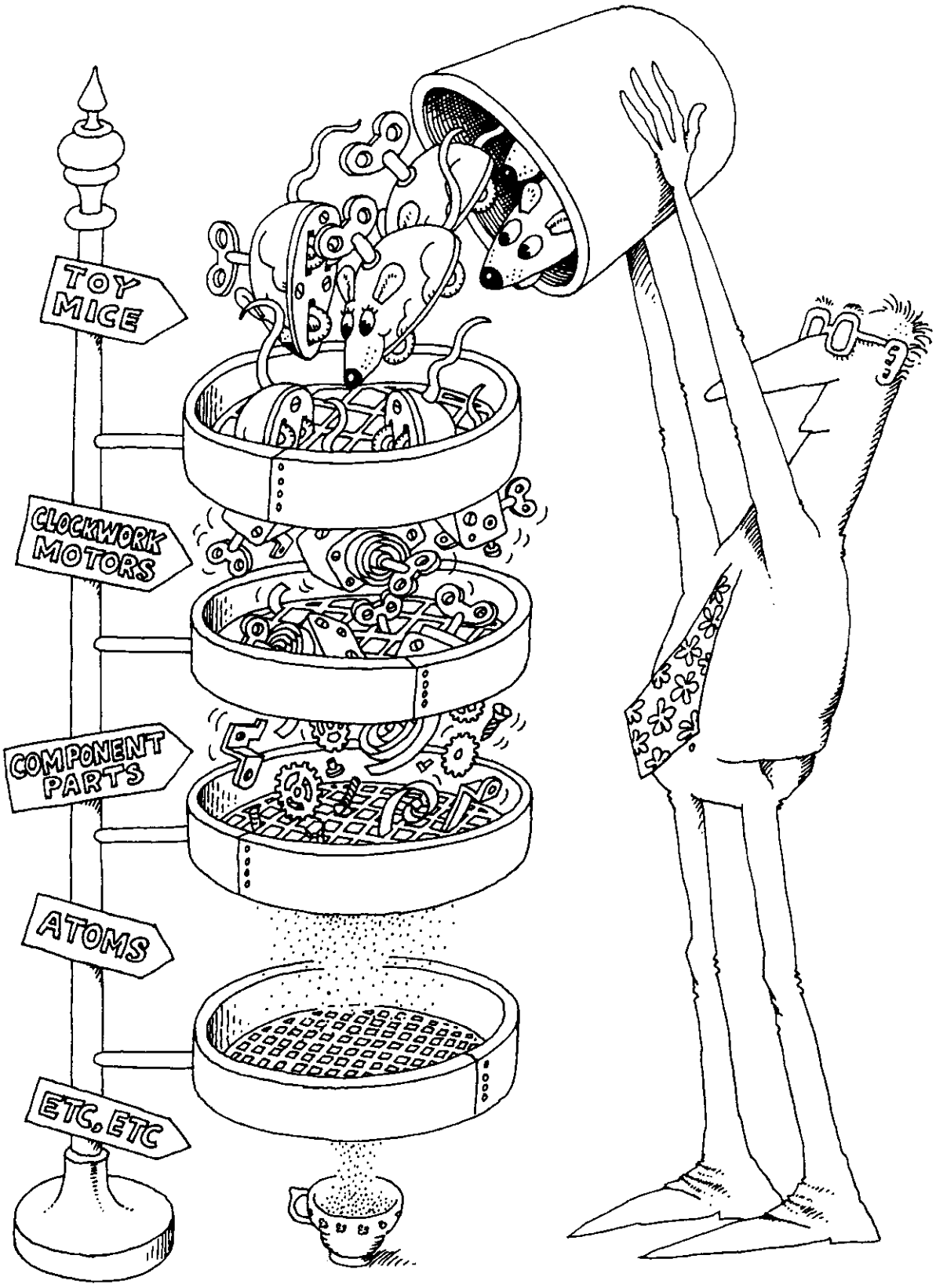
**The Meaning of Hierarchy.** Abstraction is a good thing, but in all except the most trivial applications, we may find many more different abstractions than we can comprehend at one time. Encapsulation helps manage this complexity by hiding the inside view of our abstractions. Modularity helps also, 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 "kind of" hierarchy) and its object structure (the "part of" hierarchy).

**Examples of Hierarchy: Single Inheritance.** Inheritance is the most important "kind of" 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 (called *single inheritance* and *multiple inheritance*, respectively).




Abstractions form a hierarchy.

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.

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 crops, however, might demand specialized growing plans. For example, the growing plan for all fruits might be different from the plan for all vegetables, or for all floral crops. A standard fruit-growing plan is therefore a kind of growing plan that encapsulates specialized behavior, such as the knowledge of when to harvest the fruit. We can assert this "kind of" relationship among abstractions in C++ as follows:

```
class StandardFruitGrowingPlan : public GrowingPlan {

public:
    StandardFruitGrowingPlan ();
    StandardFruitGrowingPlan (const StandardFruitGrowingPlan&);
    virtual ~StandardFruitGrowingPlan ();

    virtual int daysUntilHarvest (day currentDay) const;

private:
    day timeToHarvest;

};
```

This class declaration means that the class `StandardFruitGrowingPlan` is just like its superclass, `GrowingPlan`, except that objects of this specialized class have an additional member object (`timeToHarvest`), a different constructor and destructor, and a new virtual member function (`daysUntilHarvest`). Using this class, we could declare even more specialized subclasses, such as the class `StandardAppleGrowingPlan`.

As we evolve our inheritance hierarchy, the structure and behavior that are the same for different classes will tend to migrate to common superclasses. This is why we often speak of inheritance as being a *generalization/specialization* hierarchy; in some circles, inheritance is called the *is a* 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 "kind of" hierarchies that exist can lead to bloated, inelegant designs. As Cox points out, "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" [61].

There is a healthy tension among the principles of abstraction, encapsulation, and hierarchy. As Danforth and Tomlinson point out, "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" [62]. For a given class, there are usually two kinds of clients: objects that invoke operations upon 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" [63]. Different programming languages trade off support for encapsulation and inheritance in different ways, but among the languages used in this book, C++ offers the greatest flexibility. Specifically, the interface of a class may have three parts: *private* parts, which declare members that are visible only to the class itself, *protected* parts, which declare members that are visible only to the class and its subclasses, and *public* parts, which are visible to all clients.

**Examples of Hierarchy: Multiple Inheritance.** The previous example illustrated the use of single inheritance: the subclass `StandardFruitGrowingPlan` had exactly one superclass, the class `GrowingPlan`. For some 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. In CLOS, we might declare this class as follows:

```
(defclass plant ()
    ((name                :initarg  :name
                          :reader   plant-name)
     (date-planted        :initarg  :date-planted
                          :reader   date-planted)
     (germination-time    :initarg  :germination-time
                          :reader   germination-time)
     (actual-germination  :initform nil
                          :accessor actual-germination))
    (:documentation "The base class of all plants."))
```

According to this class definition, each instance of the class `plant` will have four slots (`name`, `date-planted`, `germination-time`, and `actual-germination`). Methods for initializing each slot are provided (the :initarg and :initform slot options), as well as methods for reading the first three slots (the :reader slot option), and reading and writing the fourth slot (the :accessor slot option).

Our analysis of the problem domain might suggest that flowering plants and fruits and vegetables have specialized properties. 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 `flowering-plant` class and a `fruit/vegetable-plant` 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 model, we would need to invent a third class, a `flowering/fruit/vegetable-plant`, that duplicated information from the `flowering-plant` and `fruit/ vegetable-plant` classes.

A better way to express our abstractions and thereby avoid this redundancy is to use multiple inheritance. First, we would invent classes that independently capture the properties unique to flowering plants and fruits and vegetables:

```
(defclass flowering-plant-mixin ()
    ((time-to-flower :initarg :time-to-flower
                     :reader  time-to-flower)
     (time-to-seed   :initarg :time-to-seed
                     :reader  time-to-seed))
    (:documentation "A mixin class for flowering plants."))

(defclass fruit/vegetable-plant-mixin ()
    ((time-to-harvest :initarg :time-to-harvest
                      :reader  time-to-harvest))
    (:documentation "A mixin class for fruits and vegetables."))
```

Notice that 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 `flowering-plant` class as follows:

```
(defclass flowering-plant (plant flowering-plant-mixin)
    ()
    (:documentation "A flowering plant class."))
```

Similarly, a `fruit/vegetable-plant` class can be declared as follows:

```
(defclass fruit/vegetable-plant (plant fruit/vegetable-plant-mixin)
    ()
    (:documentation "A fruit or vegetable plant."))
```

In both cases, we form the subclass by inheriting from two superclasses. Instances of the subclass `flowering-plant` thus include the slot `germination-time` (inherited from the class `plant`) as well as the slot `time-to-flower` (inherited from the class `flowering-plant-mixin`). Now, suppose we want to declare a class for a plant that has both flowers and fruit. We might write the following:

```
(defclass flowering/fruit/vegetable-plant (plant
                                           flowering-plant-mixin
                                           fruit/vegetable-plant-mixin)
    ()
    (:documentation "A flowering fruit or vegetable plant."))
```

**Examples of Hierarchy: Aggregation.** Whereas these "kind of" hierarchies denote generalization/specialization relationships, "part of" hierarchies describe aggre-

gation relationships. For example, a flowering plant object is built up of six subobjects (the four slots defined in the class `plant`, and the two slots defined in the class `flowering-plant-mixin`). When dealing with hierarchies such as these, we often speak of *levels of abstraction*, a concept first described by Dijkstra [64]. In terms of its "kind of" hierarchy, a high-level abstraction is generalized, and a low-level abstraction is specialized. Therefore, we say that a plant class is at a higher level of abstraction then a flowering-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 `StandardFruitGrowingPlan` is at a higher level of abstraction than the type `day`, upon which it builds.

## Typing

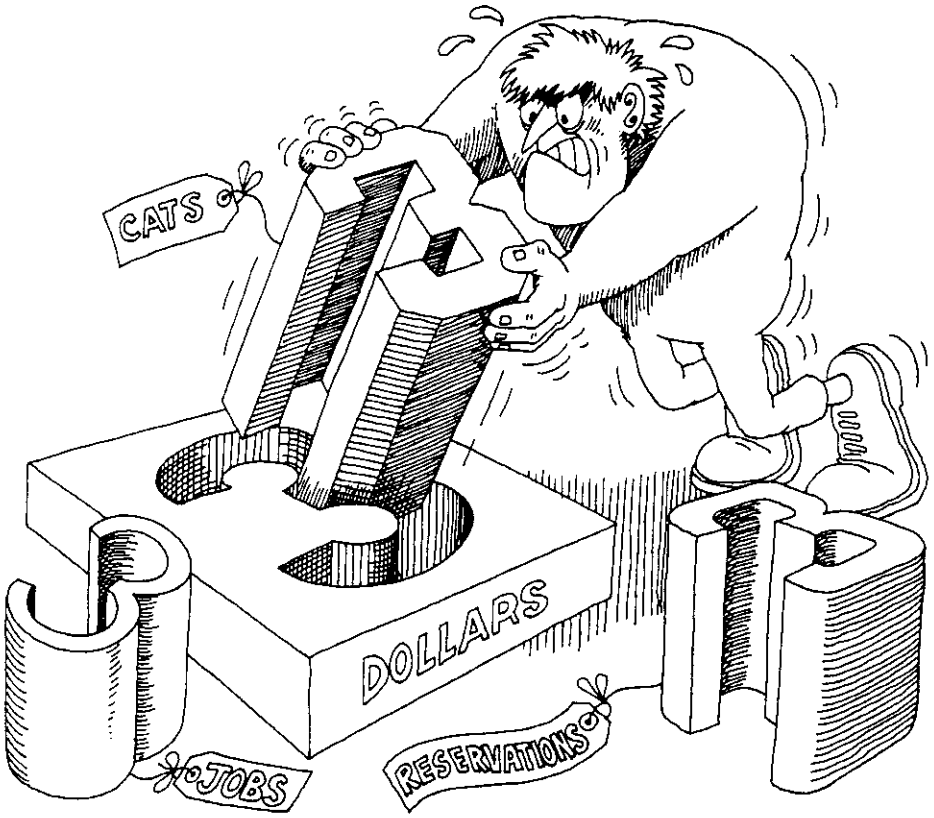
**The Meaning of Typing.** The concept of a *type* derives primarily from the theories of abstract data types. As Deutsch suggests, "A type is a precise characterization of structural or behavioral properties which a collection of entities all share" [65]. For our purposes, we will use the terms *type* and *class* interchangeably.* Although the concepts of a type and a class are similar, we include typing as a separate element of the object model because the concept of a type places a very different emphasis upon 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. Wegner observes that this kind of enforcement is essential for programming-in-the-large [67]. We consider it a minor element, however, because a given programming language may be strongly typed, weakly typed, or even untyped, yet still be called object-based or object-oriented.

---

* A type and a class are not exactly the same thing; some languages actually distinguish these two concepts. For example, early versions of the language Trellis/Owl permitted an object to have both a class and a type. Even in Smalltalk, objects of the classes `SmallInteger`, `LargeNegativeInteger`, and `LargePositiveInteger` are all of the same type, `Integer`, although not of the same class [66]. 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.




Strong typing prevents mixing abstractions.

**Examples of Typing: Strong and Weak Typing.** Returning to the user interface segment of the hydroponics gardening system, suppose that we have the following (incomplete) classes written in Object Pascal:

```
TShape = object (TObject)
    fPosition : Point;
    procedure TShape.Draw (Area : Rect);
    function TShape.IsVisible : Boolean;
end;
TText = object (TShape)
    fValue : Str255;
    procedure TText.Draw (Area : Rect); override;
end;
TGreenhouse = object (TShape)
    fHydroponics_Tanks : array[1 .. 10] of THydroponicsTank;
    procedure TGreenhouse.Initialize;
```

```
    procedure TGreenhouse.Draw (Area : Rect); override;

end;
```

In Object Pascal, all fields and methods declared in the interface of a class are public. The implementation of such classes is therefore visible, which is why we say that Object Pascal does not allow us to fully encapsulate our abstractions.

The class `TShape` inherits from the base class `TObject` and serves as the superclass of any object that can be drawn on a workstation screen. `TText` and `TGreenhouse` are both subclasses of this class, and represent more specialized objects that can be drawn. `TShape` thus declares the common method `Draw`, so that all drawable objects have this behavior; the subclasses `TText` and `TGreenhouse` specialize this method to draw text and an iconic representation of a greenhouse, respectively.

Since Object Pascal is a strongly typed language, we must explicitly assert the type of each variable, subprogram parameter, and class field when we declare it. For example, suppose that we have the following declarations:

```
AnObject    : TObject;
AShape      : TShape;
ATextString : TText;
AGreenhouse : TGreenhouse;
```

We might then create new objects with the following statements:

```
new (AnObject);
new (AShape);
new (ATextString);
new (AGreenhouse);
```

Variables such as `ATextString` are not objects. To be precise, `ATextString` is simply a name we use to designate an object of the class `TText`: when we say "the object `ATextString`," we really mean the instance of `TText` denoted by the variable `ATextString`. We will explain this subtlety again in the next chapter.

Because Object Pascal is strongly typed, statements that invoke methods are checked for type correctness at the time of compilation. For example, the following statements are legal:

```
AShape.Draw(SomeArea);      {Draw is defined for the class TShape}
ATextString.Draw(SomeArea); {Draw is defined for the class TText}
```

However, the following statements are not legal and would be rejected at compilation time:

```
AnObject.Draw(SomeArea); {Illegal}
ATextString.Initialize;  {Illegal}
```

Neither of these two statements is legal because the methods `Draw` and `Initialize` are not defined for the class of the corresponding variable, nor for any superclasses of its class. On the other hand, the following statement is legal:

```
if AGreenhouse.IsVisible then
    ...
```

Although `IsVisible` is not defined in the class `TGreenhouse`, it is defined in the class `TShape`, from which the class `TGreenhouse` inherits its structure and behavior.

Consider this same example in Smalltalk, an untyped language. Variables are untyped, as in the following local declaration:

```
|anObject aShape ATextString aGreenhouse|
```

A statement such as

```
anObject draw: ScreenArea.
```

would be accepted at compilation time, but its exact meaning could not be known until execution time. If the variable `anObject` happened to denote an object of the class `Shape` (whose class had knowledge of the method `draw`), then execution would proceed normally. On the other hand, if `anObject` denoted an instance of the predefined Smalltalk class `Bag` (which has no knowledge of the method `draw`), then evaluating this statement would ultimately lead to a "message not understood" error at runtime.

A strongly typed language is one in which all expressions are guaranteed to be type-consistent. The meaning of type consistency is best illustrated by the following example, using the previously declared Object Pascal variables. The following assignment statements are legal:

```
AnObject := AnObject;
AShape := ATextString;
```

The first statement is legal because the class of the variable on the left side of the statement (`TObject`) is the same as the class of the expression on the right side. The second statement is also legal because the class of the variable on the left side (`TShape`) is a superclass of the variable on the right side (`TText`).

Consider the following statements:

```
AGreenhouse := AShape;      {Illegal}
ATextString := AGreenhouse; {Illegal}
```

Neither of these statements is legal because the class of the variable on the left side of the assignment statement is a subclass of the class of the expression on the right.

In some situations, it is necessary to convert a value from one type to another. For example, given the predefined class `TList` from a class library for Object Pascal, we have the operation `Each`, which allows us to visit every element in the list:

```
procedure TList.Each (procedure DoToItem (Item : TObject));
```

If we know that our list object will always contain objects of the class `TGreenhouse`, then we may explicitly coerce the value of one type to another, as in the following legal assignment statement:

```
procedure DoToItem (Item : TObject);
begin
    ...
    AGreenhouse := TGreenhouse(Item);
    ...
end;
```

This assignment statement is type-consistent, although it is not completely type-safe. For example, if the list happened to contain an object of the class `TText` at runtime, then the coercion would fail with an execution error.

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" [68].

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" [69]. In practice, the safety offered by strongly typed languages usually more then compensates for the flexibility lost by not using an untyped language, especially for programming-in-the-large.

**Examples of Typing: Static and Dynamic Binding.** The concepts of strong typing and static typing are entirely different. Strong typing refers to type consistency, whereas static typing – also known as *static binding* or *early binding* – refers to the time when names are bound to types. Static binding means that the types of all variables and expressions are fixed at the time of compilation; *dynamic binding* (also called *late binding*) means that the types of all variables and expressions are not known until runtime. Because strong typing and binding are independent concepts, a language may be both strongly and statically typed (Ada), strongly typed yet support dynamic binding (Object Pascal and C++), or untyped yet support dynamic binding (Smalltalk). CLOS fits somewhere between C++ and Smalltalk, in that an implementation may either enforce or ignore any type declarations asserted by a programmer.

Let's again illustrate these concepts with an example from Object Pascal. Earlier, we used the class `TList`, which is found in a class library for Object Pascal, and represents a singly linked list. Its (highly elided) declaration follows:

```
TList = object(TObject)
    ...
    procedure TList.InsertFirst (Item : TObject);
    procedure TList.Each (procedure DoToItem (Item : TObject));
    ...
end;
```

Here we see two methods: the first for adding items to the list (a *modifier*), and the second for visiting every item in the list (an *iterator*). Because Object Pascal supports dynamic binding, we may have either homogeneous lists, in which all elements are of the same class, or heterogeneous lists whose elements are all of different classes, as long as each item is an instance of the class `TObject` or any of its subclasses.

Assume that we have an object of the class `TList` named `AList`, representing a heterogeneous list of objects that are all of the class `TShape` or its subclasses. We can thus write statements such as the following:

```
AList.InsertFirst(AShape);
AList.InsertFirst(ATextString);
AList.InsertFirst(AGreenhouse);
```

All of these statements are type-consistent because the type of each actual parameter is of the same class (or subclass) as the corresponding formal parameter (`TObject`).

Suppose now that we want to draw each object in the list. We might write the following procedure and statement:

```
procedure Draw_Item (Item : TObject);
begin
    TShape(Item).Draw(SomeArea);
end;
...
AList.Each(Draw_Item);
```

The call to the operation `Each` invokes the list iterator, which in turn calls the procedure `Draw_Item` for each object in the list. Notice that in the procedure `Draw_Item`, we have to coerce the variable `Item` to the class `TShape` so that we can invoke the operation `Draw`, which we defined for the class `TShape` and its subclasses. At the time of compilation, however, we cannot know the exact subclass of the object designated by the formal parameter `Item`: it might be of the class `TText` or `TGreenhouse`, for instance. This is an example of dynamic binding.

Fortunately, because we can control what goes into the list, it is safe for us to coerce the formal parameter `Item` to an object of the class `TShape`. Because the method `Draw` is defined in the class `TShape`, it is type-consistent to invoke this method. Thus, the effect of invoking the iterator `Each` is to walk down the

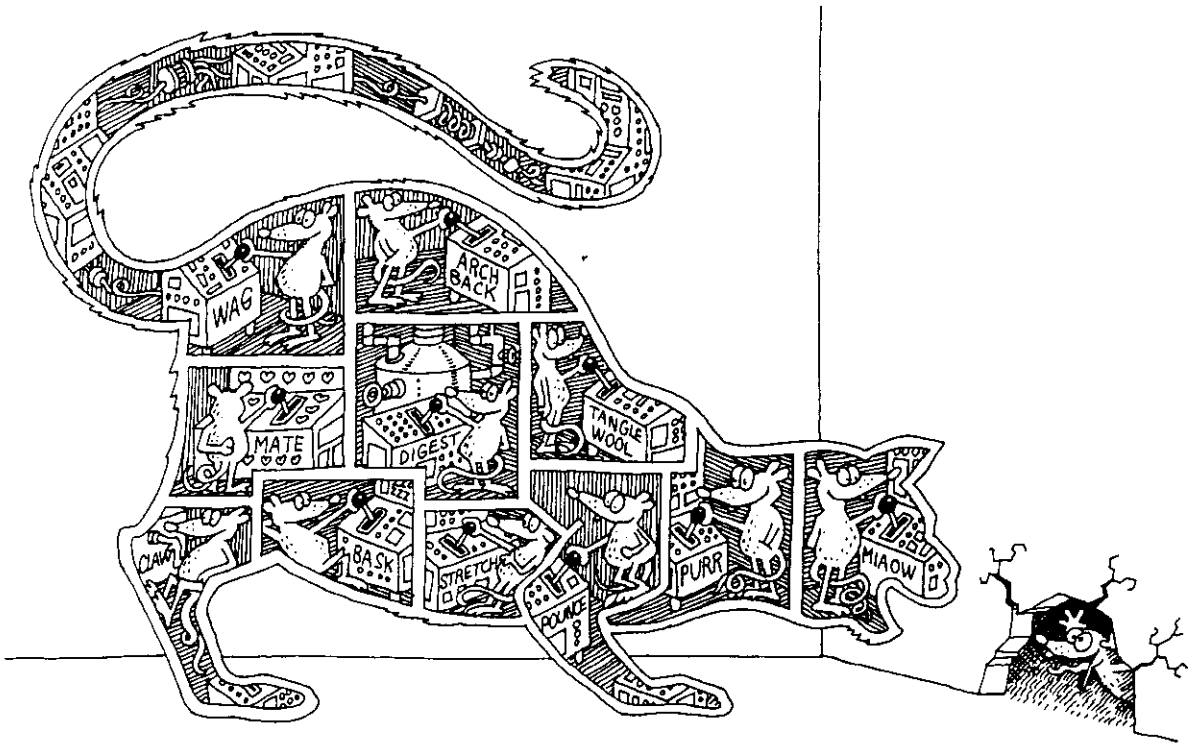
list and invoke the `Draw` method of each object we find along the way. Because each object may be of a different class, each object may respond differently to the invocation of the `Draw` method. Ultimately, this means that we cannot know until runtime what `Draw` method is actually called. This feature is called *polymorphism*; it 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 [70]. The opposite of polymorphism is *monomorphism*, which is found in all languages that are both strongly typed and statically bound, such as Ada.

Polymorphism exists when the features of inheritance and dynamic binding interact. It 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 an important concept in object-oriented design.

## Concurrency

**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 processors capable of multitasking. A single process – also known as a *thread of control* – 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 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.

Lim and Johnson point out that "designing features for concurrency in OOP languages is not much different from [doing so in] other kinds of languages – concurrency is orthogonal to OOP at the lowest levels of abstraction. OOP or not, all the traditional problems in concurrent programming still remain" [71]. Indeed, 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. Fortunately, as Lim and Johnson also point out, "At the highest levels of abstraction, OOP can alleviate the concurrency problem for the majority of programmers by hiding the concurrency inside reusable abstractions" [72]. Black et al. therefore suggest that "an object model is appropriate for a




Concurrency allows different objects to act at the same time.

distributed system because it implicitly defines (1) the units of distribution and movement and (2) the entities that communicate" [73].

Whereas object-oriented programming focuses upon data abstraction, encapsulation, and inheritance, concurrency focuses upon process abstraction and synchronization [74]. 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.** Our discussion of abstraction introduced an Ada package specification for a sequential class representing air temperature sensors. Our variation of this package captured the design of an active sensor class, whose behavior was to periodically determine the current temperature and then send a message to another object whenever the temperature changed a certain number of degrees. Ada's mechanism for expressing a concurrent process is the task, and therefore, we might complete this representation of the type

`Air_Temperature_Sensor` as follows (ignoring the need for the calibration operation):

```
task type Air_Temperature_Sensor is
   entry Initialize (Its_Location      : in Location;
                     Lower_Alarm_Limit : in Temperature;
                     Upper_Alarm_Limit : in Temperature);
end Air_Temperature_Sensor;
```

For every instance of this task type, we generate a new process. From an object-oriented perspective, there is exactly one operation that we can perform upon objects of this type. Specifically, we can initialize it, by telling the object its location and its upper and lower temperature limits, outside of which other objects are notified (via the generic formal subprogram parameter, `Temperature_Alarm`).

Suppose that each physical sensor uses memory-mapped I/O. To read the value of a particular sensor, we need only reference some location in memory. In Ada, we might express these design decisions by hiding the following declarations in the body of the package `Air_Temperature_Sensors`:

```
type Word is range -(2**15 -1) .. (2**15 - 1);
for Word'Size use 16;

Sensor_Memory_Map : array(Location) of Word;
for Sensor_Memory_Map use at 16#377FF0#;

Time_Interval : constant Duration := 2.0;
```

These declarations are hidden because they are part of the secrets of this abstraction, such as the fact that the memory map starts at hexidecimal location 377FF0 and that each raw sensor value is a 16-bit number. The constant `Time_Interval` represents how often we want to read the physical sensor (in this case, once every two seconds).

The basic idea of the sensor task is to read the memory map every `Time_Interval` seconds and then report to other objects any changes or alarm conditions. Ignoring the issues of calibration and termination, we might implement the body of this task type as follows:

```
task body Air_Temperature_Sensor is
   Current_Location     : Location;
   Sensor_Value         : Word;
   Current_Temperature  : Temperature;
   Previous_Temperature : Temperature := Temperature'Last;
   Lower_Limit          : Temperature;
   Upper_Limit          : Temperature;
   Next_Time            : Calendar.Time := Calendar.Clock;
begin
   accept Initialize (Its_Location      : in Location;
                      Lower_Alarm_Limit : in Temperature;
                      Upper_Alarm_Limit : in Temperature) do
      Current_Location := Its_Location;
```

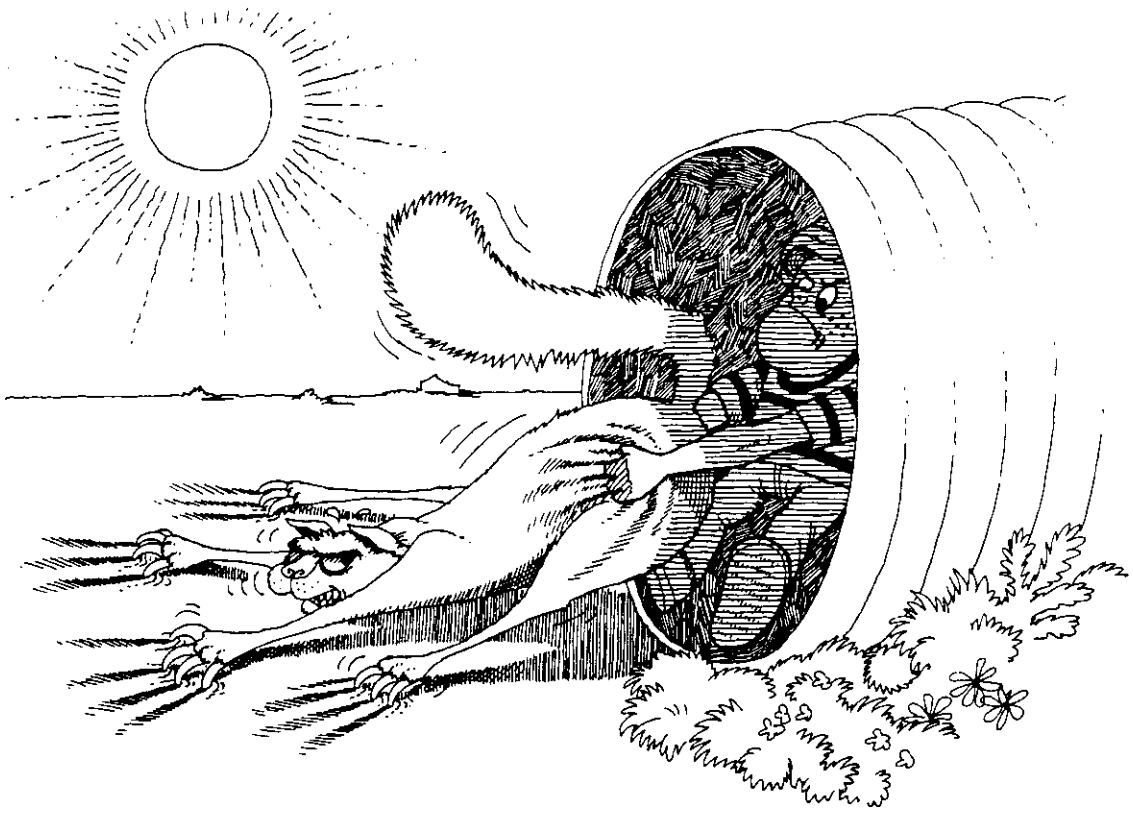
```
        Lower_Limit := Lower_Alarm_Limit;
        Upper_Limit := Upper_Alarm_Limit;
    end Initialize;
    loop
        delay (Next_Time - Calendar.Clock);
        Sensor_Value:= Sensor_Memory_Map(Current_Location);
        Current_Temperature := Temperature(Float(Sensor_Value) * Temperature'Delta);
        if (Current_Temperature /= Previous_Temperature) then
            if (Current_Temperature < Lower_Limit)  or
               (Current_Temperature > Upper_Limit) then
                Temperature_Alarm(Current_Location, Current_Temperature);
            else
                Previous_Temperature := Current_Temperature;
                Temperature_Has_Changed(Current_Location, Current_Temperature);
            end if;
        end if;
        Next_Time := Next_Time + Time_Interval;
    end loop;
end;
```

For clarity, this implementation has several more local variables than one would probably use in production code.

Once the active object is initialized, its process loops every two seconds, during which time the physical sensor value is read. If this value is different than the previous reading, the algorithm continues. If this new reading also exceeds the upper or lower temperatures established upon initialization, then the procedure `Temperature_Alarm` is called. Otherwise, the procedure `Temperature_Has_Changed` is called, to notify some other object of the change.

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 upon 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.

There are a number of experimental concurrent object-oriented programming languages, such as Actors, Orient 84/K, and ABCL/1, that provide mechanisms for active objects and synchronization. The appendix provides references to these and other languages. Among the languages used in this book, only Smalltalk and Ada directly support multitasking (Smalltalk has the class Process and Ada incorporates the concept of a task type). Concurrent objects in C++ are possible through the use of the Unix system call *fork*. Object Pascal and CLOS are typically used for sequential applications only; they do not have primitives for concurrency.



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

## 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" [75]

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 designs: 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 [76].

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 upon 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 lifetime transcends 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.

There are only a handful of object-oriented databases, such as TAXIS, SDM, DAPLEX, and GEM [77]. None of the five languages we use in the applications support persistence directly, so we have no examples to offer here. As we will see in Chapters 9 and 10, however, it is possible to achieve the illusion of persistence in these languages.

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 upon 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. We examine this kind of persistence further in the application in Chapter 12.

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).*

## 2.3 Applying the Object Model

### Benefits of the Object Model

As we have shown, the object model is fundamentally different than 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 upon 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. In our experience, there are five other practical benefits to be derived from the application of the object model.

First, the use of the object model helps us to exploit the expressive power of all 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" [78]. Our experience has been that, without the application of the elements of the object model, the more powerful features of languages such as Smalltalk, Object Pascal, C++, CLOS, and Ada are either ignored or greatly misused.

Next, the use of the object model encourages the reuse not only of software but of entire designs [79]. 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.

Third, the use of the object model produces systems that are built upon stable intermediate forms, and thus 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 explains further how the object model reduces the risk of developing complex systems, primarily because integration is spread out across the life cycle rather than occurring as one big bang 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, for as Robson suggests, "Many people who have no idea how a computer works find the idea of object-oriented systems quite natural" [80].

## Applications of the Object Model

The object model has proven applicable to a wide variety of problem domains. Figure 2-6 lists many of the domains for which systems exist that may properly be called object-oriented. The Bibliography provides an extensive list of references to these and other applications.

Object-oriented design may be the only method we have today that can be employed to attack the complexity inherent in very large systems. In all fairness, however, the use of object-oriented design may be ill-advised for some domains, not for any technical reasons, but for nontechnical ones, such as the absence of a suitably trained staff or good development environment.

Air traffic control
Animation
Avionics
Banking and insurance software
Business data processing
Chemical process control
Command and control systems
Computer aided design
Computer aided education
Computer integrated manufacturing
Databases
Document preparation
Expert systems
Film and stage storyboarding
Hypermedia
Image recognition
Investment strategies
Mathematical analysis
Medical electronics
Music composition
Office automation
Operating systems
Petroleum engineering
Reusable software components
Robotics
Software development environments
Space station software
Spacecraft and aircraft simulation
Telecommunications
Telemetry systems
User interface design
VLSI design

**Figure 2-6**
Applications of the Object Model

## 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.

- The object model provides the conceptual framework for object-oriented methods; the object model encompasses the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence.
- 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 hiding all of the details of an object that do not contribute to its essential characteristics.
- 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, they 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.
- The application of the object model leads to systems that embody the five attributes of well-structured complex systems.

## Further Readings

The concept of the object model was first introduced by Jones [F 1979] and Williams [F 1986]. Kay's Ph.D. thesis [F 1969] established the direction for much of the work in object-oriented programming that followed.

Shaw [J 1984] provides an excellent summary regarding abstraction mechanisms in high-order programming languages. The theoretical foundation of abstraction may be found in the work of Liskov and Guttag [H 1986], Guttag [J 1980], and Hilfinger [J 1982]. Parnas [F 1979] provides the seminal work on information hiding. The meaning and importance of hierarchy are discussed in the work edited by Pattee [J 1973].

There is a wealth of literature regarding object-oriented programming. Cardelli and Wegner [J 1985] and Wegner [J 1987] provide an excellent survey of object-based and object-oriented programming languages. The tutorial papers of Stefik and Bobrow [G 1986], Stroustrup [G 1988], and Nygaard [G 1986] are good starting points on the important issues of object-oriented programming. The books by Cox [G 1986], Meyer [F 1988], Schmucker [G 1986], and Kim and Lochovsky [F 1989] offer extended coverage of these topics.

Object-oriented design methods were first formalized by Booch [F 1981, 1982, 1986, 1987, 1989]. Variations of this method include HOOD [F 1987], as used in the European Space Station project, and GOOD, as introduced by Seidewitz and Stark [F 1988]. Similar object-oriented design methods have been proposed by Wirfs-Brock and Wilkerson [F 1989] (emphasizing a responsibility-driven approach), Constantine

[F 1989], and Wasserman [F 1989]. Related works include Ross [F 1987] on the topic of entity modeling, and Abelson and Sussman [H 1985] on the general topic of programming.

Object-oriented analysis methods were introduced by Shlaer and Mellor [B 1988] and Bailin [B 1988], with later contributions by Coad and Yourdon [B 1990].

An excellent collection of papers dealing with all topics of object-oriented computing may be found in Peterson [G 1987] and Schriver and Wegner [G 1987]. The proceedings of several yearly conferences on object-oriented computing are also excellent sources of material. Three of the more interesting forums include the USENIX C++ conferences, OOPSLA (Object-Oriented Programming Systems, Languages, and Applications), and ECOOP (European Conference on Object-Oriented Programming).