Chapter 2
The Object Model

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

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

2.1. The Evolution of the Object Model

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

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

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

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

The Generations of Programming Languages

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

- First-generation languages (1954–1958)

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

- Second-generation languages (1959–1961)

FORTRAN II	Subroutines, separate compilation
ALGOL 60	Block structure, data types
COBOL	Data description, file handling

Lisp	List processing, pointers, garbage collection

- Third-generation languages (1962–1970)

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

- The generation gap (1970–1980)
  Many different languages were invented, but few endured.
  However, the following are worth noting:

C	Efficient; small executables
FORTRAN 77	ANSI standardization

Let's expand on Wegner's categories.

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

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

- Emergence of frameworks (1990–today)
  Much language activity, revisions, and standardization have occurred, leading to programming frameworks.

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

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

Among second-generation languages, the emphasis was on algorithmic abstractions. By this time, machines were becoming more and more powerful, and the economics of the computer industry meant that more kinds of problems could be automated, especially for business applications. Now, the focus was largely on telling the machine what

to do: read these personnel records first, sort them next, and then print this report. Again, this new generation of high-order programming languages moved us a step closer to the problem space and further away from the underlying machine.

By the late 1960s, especially with the advent of transistors and then integrated circuit technology, the cost of computer hardware had dropped dramatically, yet processing capacity had grown almost exponentially. Larger problems could now be solved, but these demanded the manipulation of more kinds of data. Thus, third-generation languages such as ALGOL 60 and, later, Pascal evolved with support for data abstraction. Now a programmer could describe the meaning of related kinds of data (their type) and let the programming language enforce these design decisions. This generation of high-order programming languages again moved our software a step closer to the problem domain and further away from the underlying machine.

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

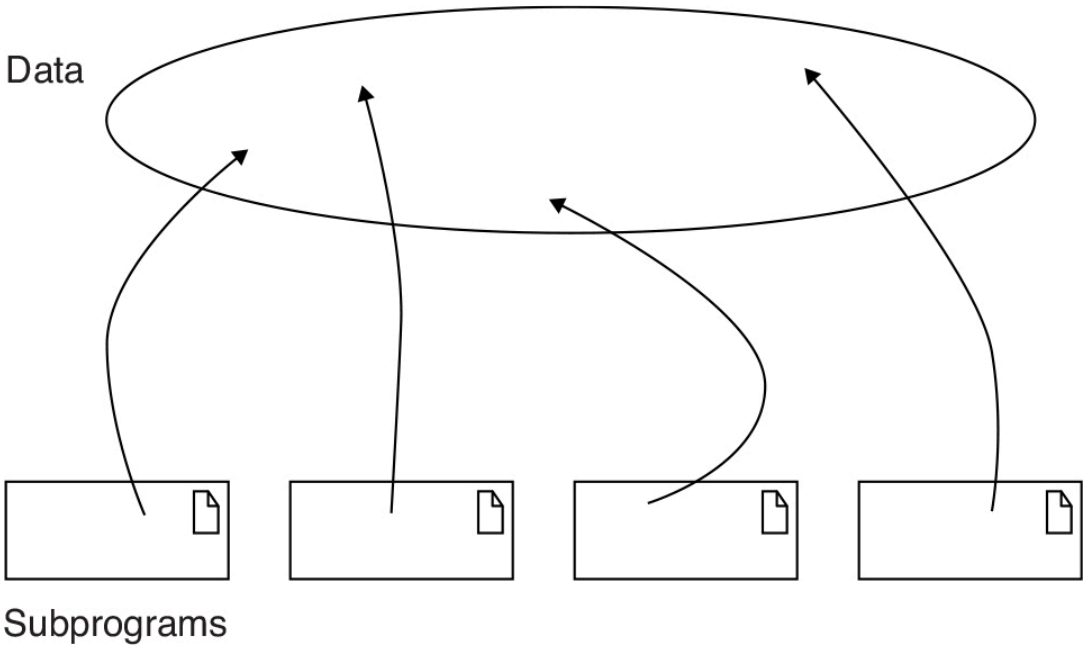
What is of the greatest interest to us is the class of languages we call *object-based* and *object-oriented*. Object-based and object-oriented programming languages best support the object-oriented decomposition of software. The number of these languages (and the number of "objectified" variants of existing languages) boomed in the 1980s and early 1990s. Since 1990 a few languages have emerged as mainstream OO languages with the backing of commercial programming tool vendors (e.g., Java, C++). The emergence of programming frameworks

(e.g., J2EE, .NET), which provide a tremendous amount of support to the programmer by offering components and services that simplify the common and often mundane programming tasks, has greatly boosted productivity and demonstrated the elusive promise of component reuse.

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

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

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



Applications written in these languages exhibit a relatively flat physical structure, consisting only of global data and subprograms. The ar-

rows 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 threatening the reliability of the entire system and certainly reducing the overall clarity of the solution.

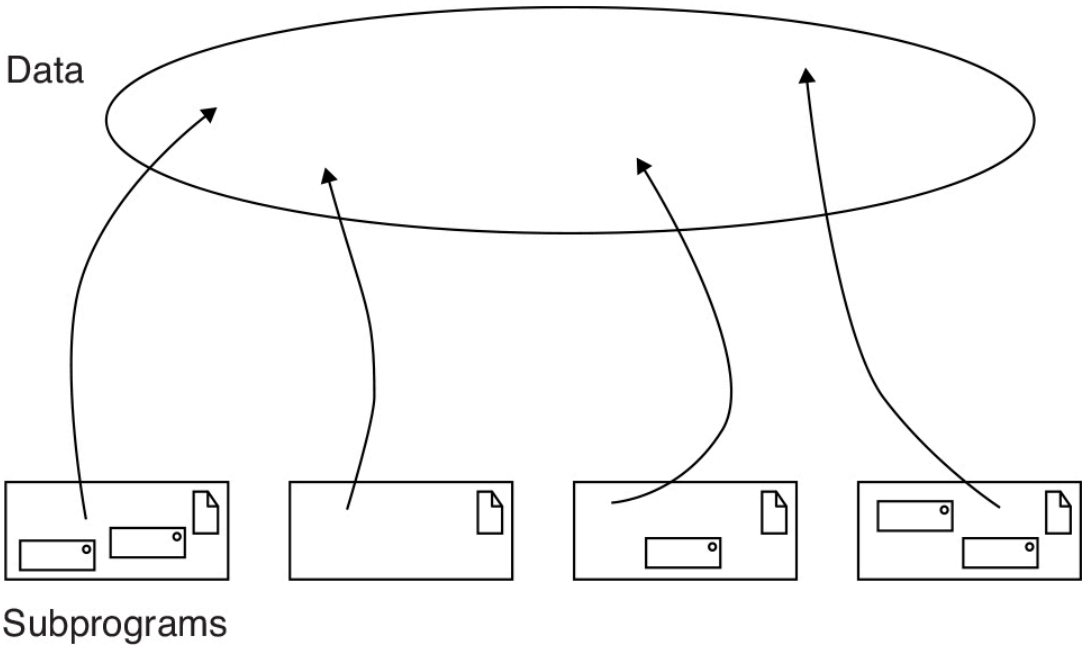
The Topology of Late Second- and Early Third-Generation Programming Languages

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

The realization that subprograms could serve as an abstraction mechanism had three important consequences. First, languages were invented that supported a variety of parameter-passing mechanisms. Second, the foundations of structured programming were laid, manifesting themselves in language support for the nesting of subprograms and the development of theories regarding control structures and the scope and visibility of declarations. Third, structured design methods emerged, offering guidance to designers trying to build large systems

using subprograms as basic physical building blocks. Thus, it is not surprising, as **Figure 2-2** shows, that the topology of late second- and early third-generation languages is largely a variation on the theme of earlier generations. This topology addresses 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.

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

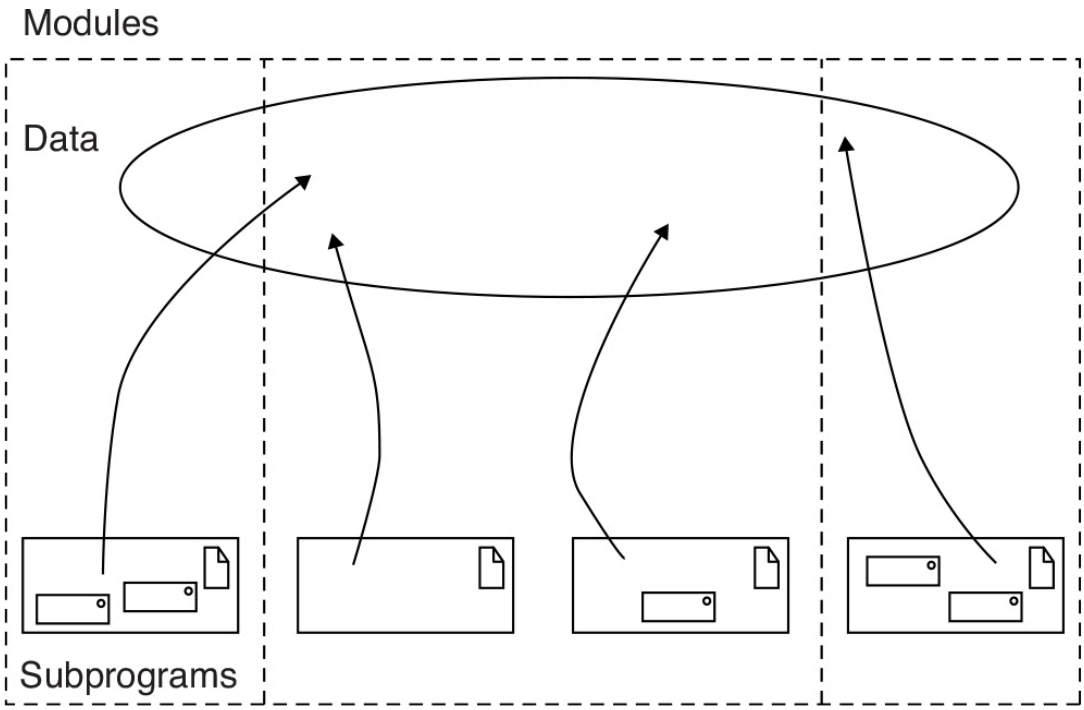


The Topology of Late Third-Generation Programming Languages

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

they were used simply to group subprograms that were most likely to change together.

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



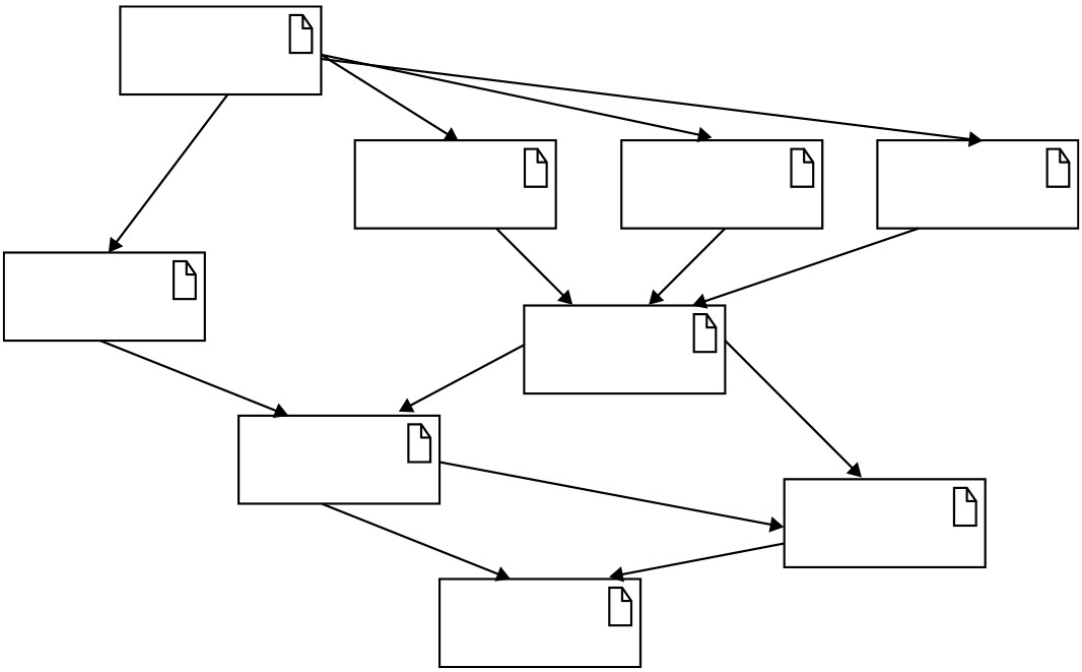
Most languages of this generation, while supporting some sort of modular structure, had few rules that required semantic consistency among module interfaces. A developer writing a subprogram for one module might assume that it would be called with three different parameters: a floating-point number,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

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

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

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

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

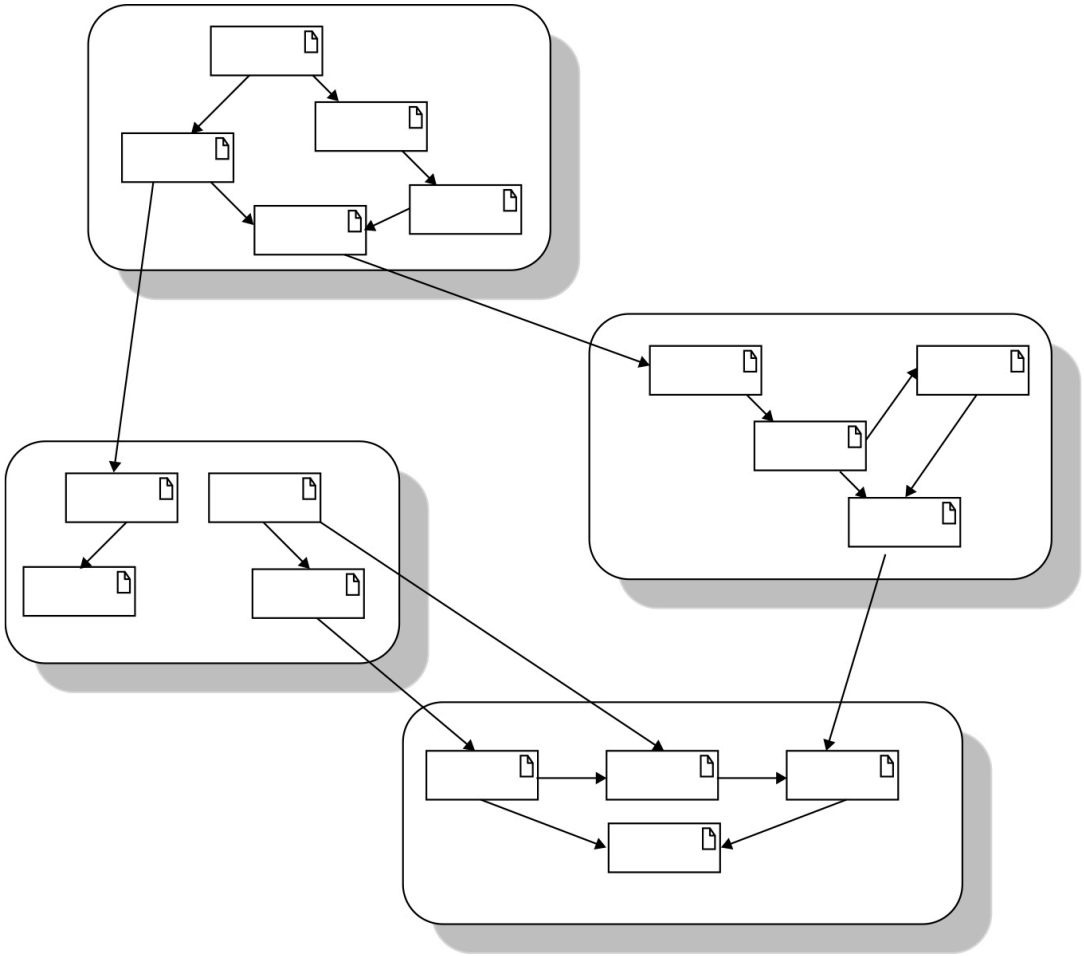


The physical building block in such languages is the module, which represents a logical collection of classes and objects instead of subprograms, as in earlier languages. To state it another way, "If procedures and functions are verbs and pieces of data are nouns, a procedure-oriented program is organized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

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

This is exactly the organization of complexity described in **Chapter 1**; this topology is shown in **Figure 2-5**.

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



2.2. Foundations of the Object Model

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

Actually, the object model has been influenced by a number of factors, not just object-oriented programming. Indeed, as further discussed in

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

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

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

The phrase *object-oriented* "has been bandied about with carefree abandon with much the same reverence accorded 'motherhood,' 'apple pie,' and 'structured programming'"[7]. What we can agree on is

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

FOUNDATIONS—THE OBJECT MODEL

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

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

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

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

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

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

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

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

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

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

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

Object-Oriented Programming

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

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

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

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

[A] language is object-oriented if and only if it satisfies the following requirements:

- It supports objects that are data abstractions with an interface of named operations and a hidden local state.
- Objects have an associated type [class].
- Types [classes] may inherit attributes from supertypes [superclasses]. [34]

For a language to support inheritance means that it is possible to express "is a" relationships among types, for example, a red rose is a

kind of flower, and a flower is a kind of plant. If a language does not provide direct support for inheritance, then it is not object-oriented. Cardelli and Wegner distinguish such languages by calling them *object-based* rather than *object-oriented*. Under this definition, Smalltalk, Object Pascal, C++, Eiffel, CLOS, C#, and Java are all object-oriented, and Ada83 is object-based (support for object orientation was later added to Ada95). However, since objects and classes are elements of both kinds of languages, it is both possible and highly desirable for us to use object-oriented design methods for both object-based and object-oriented programming languages.

Object-Oriented Design

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

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

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

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

oriented design to refer to any method that leads to an object-oriented decomposition.

Object-Oriented Analysis

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

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

How are OOA, OOD, and OOP related? Basically, the products of object-oriented analysis serve as the models from which we may start an object-oriented design; the products of object-oriented design can then be used as blueprints for completely implementing a system using object-oriented programming methods.

2.3. Elements of the Object Model

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

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

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

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

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

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

There are three minor elements of the object model:

1. Typing
2. Concurrency
3. Persistence

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

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

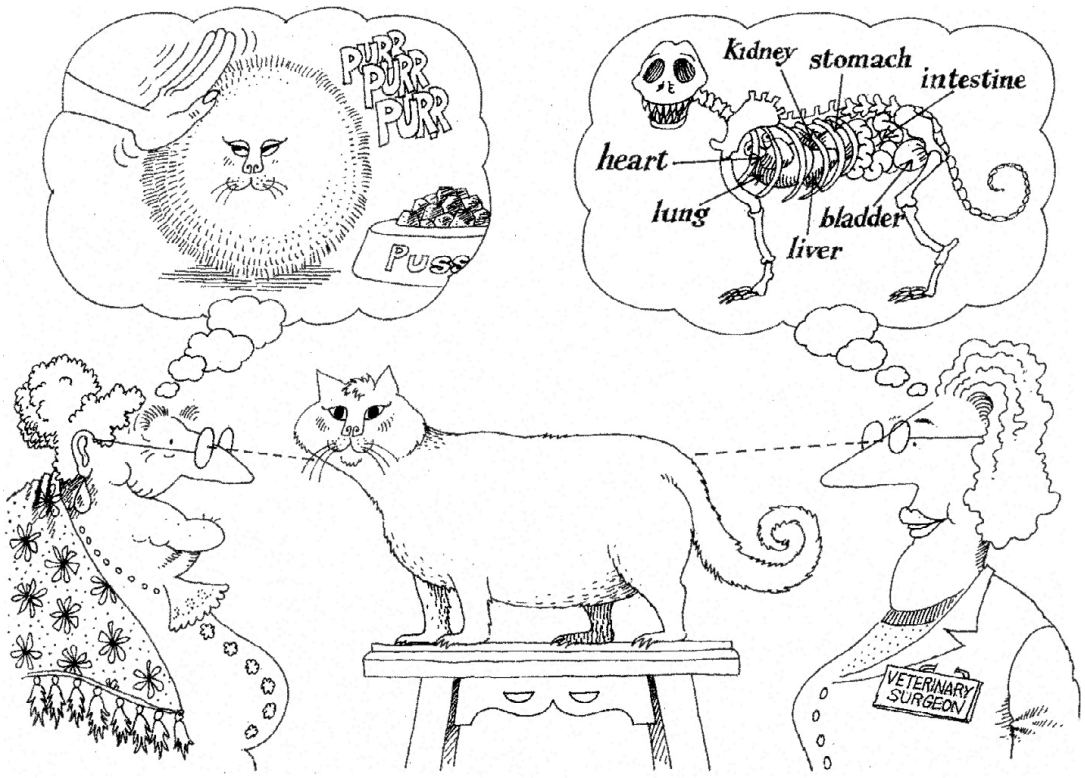
The Meaning of Abstraction

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

to realize it" [44]. Combining these different viewpoints, we define an abstraction as follows:

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

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



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

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

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

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

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

A client is any object that uses the resources of another object (known as the server). We can characterize the behavior of an object by considering the services that it provides to other objects, as well as the op-

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

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

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

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

All abstractions have static as well as dynamic properties. For example, a file object takes up a certain amount of space on a particular memory device; it has a name, and it has contents. These are all static properties. The value of each of these properties is dynamic, relative to the lifetime of the object: A file object may grow or shrink in size, its name may change, its contents may change. In a procedure-oriented style of programming, the activity that changes the dynamic value of objects is the central part of all programs; things happen when subprograms are called and statements are executed. In a rule-oriented style of programming, things happen when new events cause rules to fire, which in turn may trigger other rules, and so on. In an object-oriented style of programming, things happen whenever we operate on an object (i.e., when we send a message to an object). Thus, invoking an operation on an object elicits some reaction from the object. What operations we can meaningfully perform on an object and how that object reacts constitute the entire behavior of the object.

Examples of Abstraction

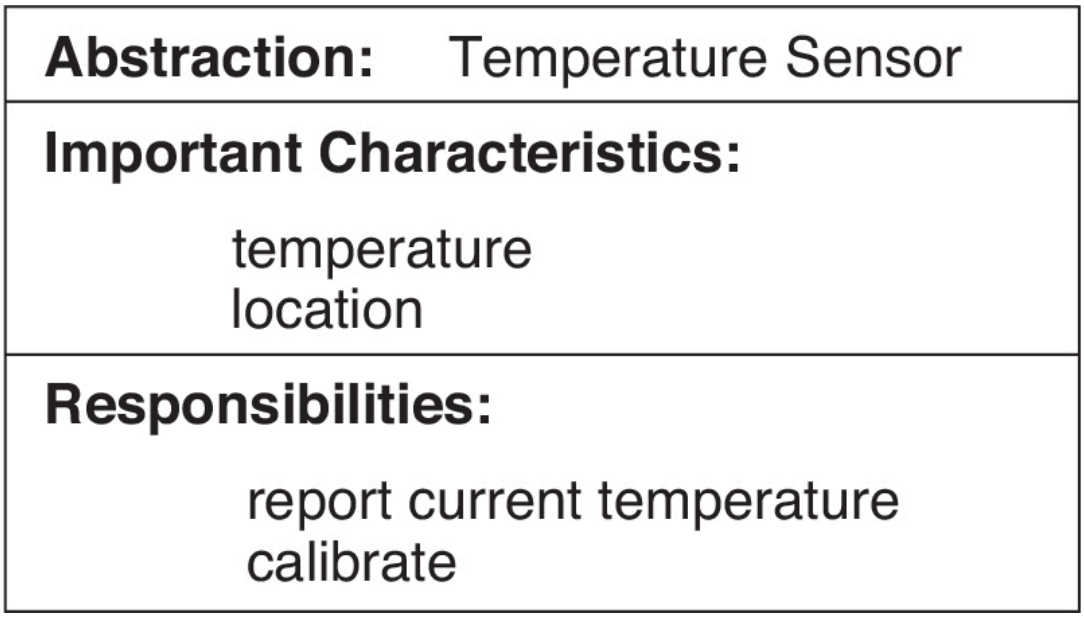
Let's illustrate these concepts with some examples. We defer a complete treatment of how to find the right abstractions for a given problem to **Chapter 4**.

On a hydroponics farm, plants are grown in a nutrient solution, without sand, gravel, or other soils. Maintaining the proper greenhouse environment is a delicate job and depends on the kind of plant being grown and its age. One must control diverse factors such as temperature, humidity, light, pH, and nutrient concentrations. On a large farm, it is not unusual to have an automated system that constantly monitors and adjusts these elements. Simply stated, the purpose of an auto-

mated gardener is to efficiently carry out, with minimal human intervention, growing plans for the healthy production of multiple crops.

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

Figure 2-6 Abstraction of a `Temperature Sensor`



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

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

Figure 2-7 Abstraction of an `Active Temperature Sensor`

Abstraction: Active Temperature Sensor
Important Characteristics: temperature location setpoint
Responsibilities: report current temperature calibrate establish setpoint

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

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

A growing plan is responsible for keeping track of all interesting actions associated with growing a crop, correlated with the times at which those actions should take place. For example, on day 15 in the lifetime of a certain crop, our growing plan might be to maintain a temperature of 78°F for 16 hours, turn on the lights for 14 of these hours, and then drop the temperature to 65°F for the rest of the day. We might also want to add certain extra nutrients in the middle of the day, while still maintaining a slightly acidic pH. From the perspective outside of each growing-plan object, a client must be able to establish

the details of a plan, modify a plan, and inquire about a plan, as shown in **Figure 2-8**. (Note that abstractions are likely to evolve over the lifetime of a project. As details begin to be fleshed out, a responsibility such as "establish plan" could turn into multiple responsibilities, such as "set temperature," "set pH," and so forth. This is to be expected as more knowledge of client requirements is gained, designs mature, and implementation approaches are considered.)

Figure 2-8 Abstraction of a `Growing Plan`

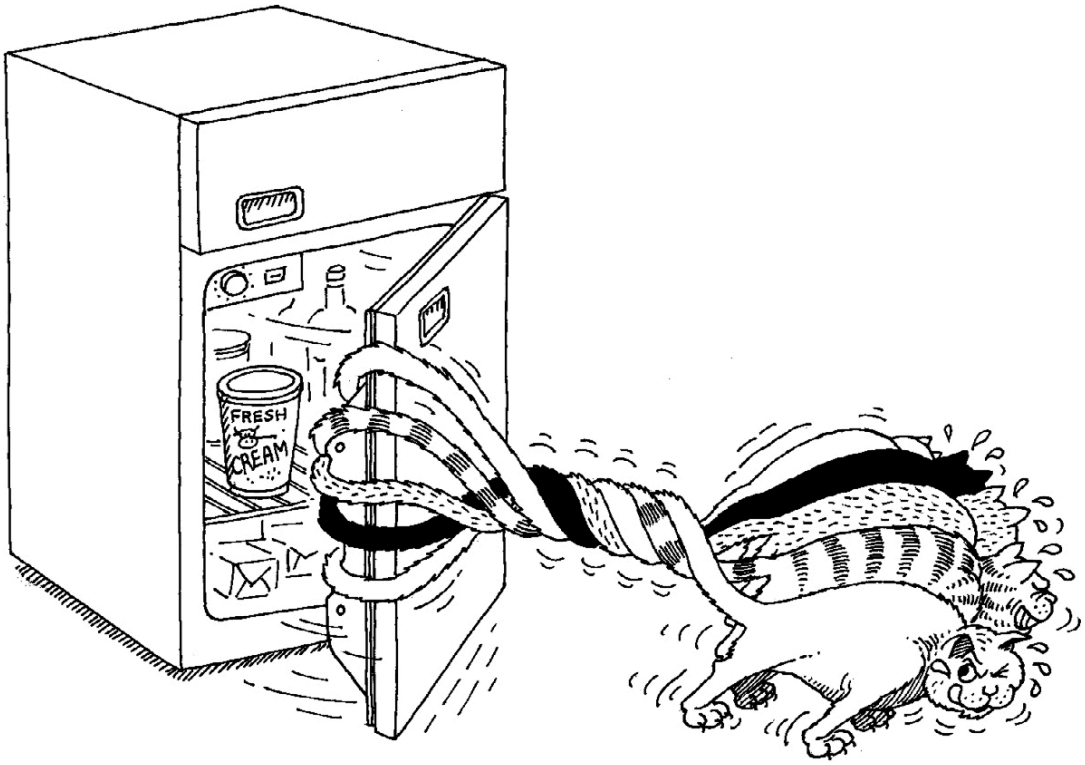
Abstraction: Growing Plan
Important Characteristics: name
Responsibilities: establish plan modify plan clear plan

Related Candidate Abstractions: Crop, Conditions, Plan Controller

Our decision is also that we will not require a growing plan to carry out its plan: We will leave this as the responsibility of a different abstraction (e.g., a `Plan Controller`). In this manner, we create a clear *separation of concerns* among the logically different parts of the system, so as to reduce the conceptual size of each individual abstraction. For example, there might be an object that sits at the boundary of the human/machine interface and translates human input into plans. This is the object that establishes the details of a growing plan, so it must be able to change the state of a `Growing Plan` object. There must also be an object that carries out the growing plan, and it must be able to read the details of a plan for a particular time.

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

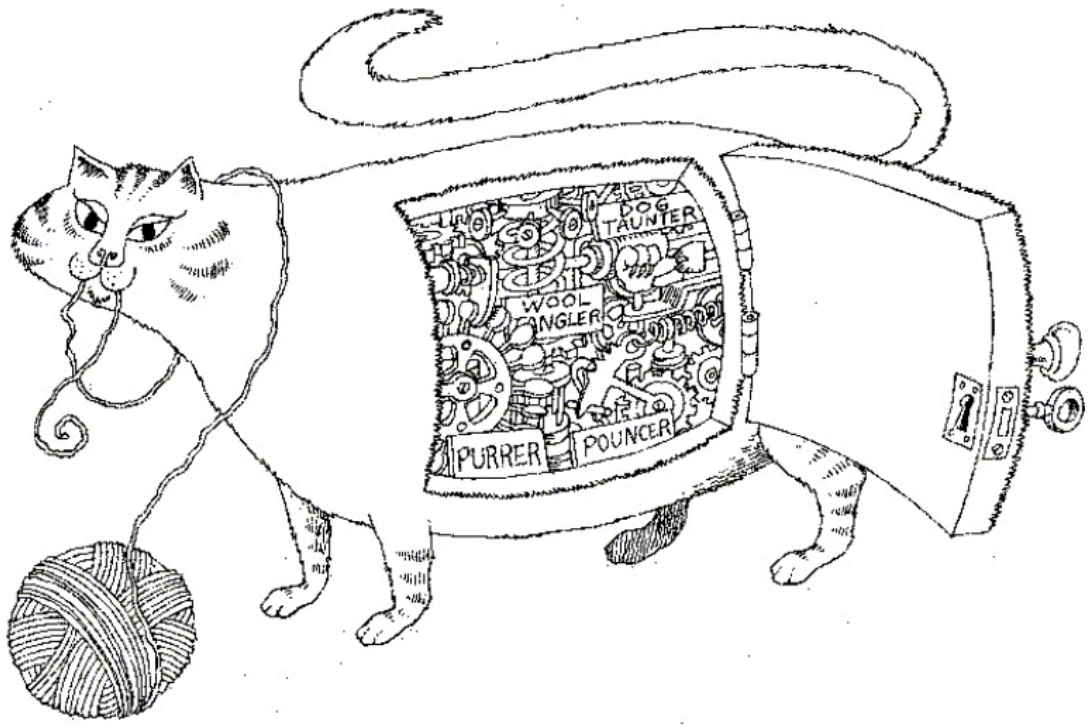
boundaries of each abstraction and thus the responsibilities and protocol of each object.



Objects collaborate with other objects to achieve some behavior.

The Meaning of Encapsulation

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



Encapsulation hides the details of the implementation of an object.

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

Encapsulation provides explicit barriers among different abstractions and thus leads to a clear separation of concerns. For example, consider again the structure of a plant. To understand how photosynthesis works at a high level of abstraction, we can ignore details such as the responsibilities of plant roots or the chemistry of cell walls. Similarly, in designing a database application, it is standard practice to write programs so that they don't care about the physical representation of

data but depend only on a schema that denotes the data's logical view [52]. In both of these cases, objects at one level of abstraction are shielded from implementation details at lower levels of abstraction.

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

To summarize, we define encapsulation as follows:

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

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

Examples of Encapsulation

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

SEPARATION OF CONCERNS

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

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

Figure 2-9 Abstraction of a `Heater`

Abstraction: Heater
Important Characteristics: location status
Responsibilities: turn on turn off provide status

Related Candidate Abstractions: Heater Controller, Temperature Sensor

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

cific heater, followed by another number used to signal turning the heater on.

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

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

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

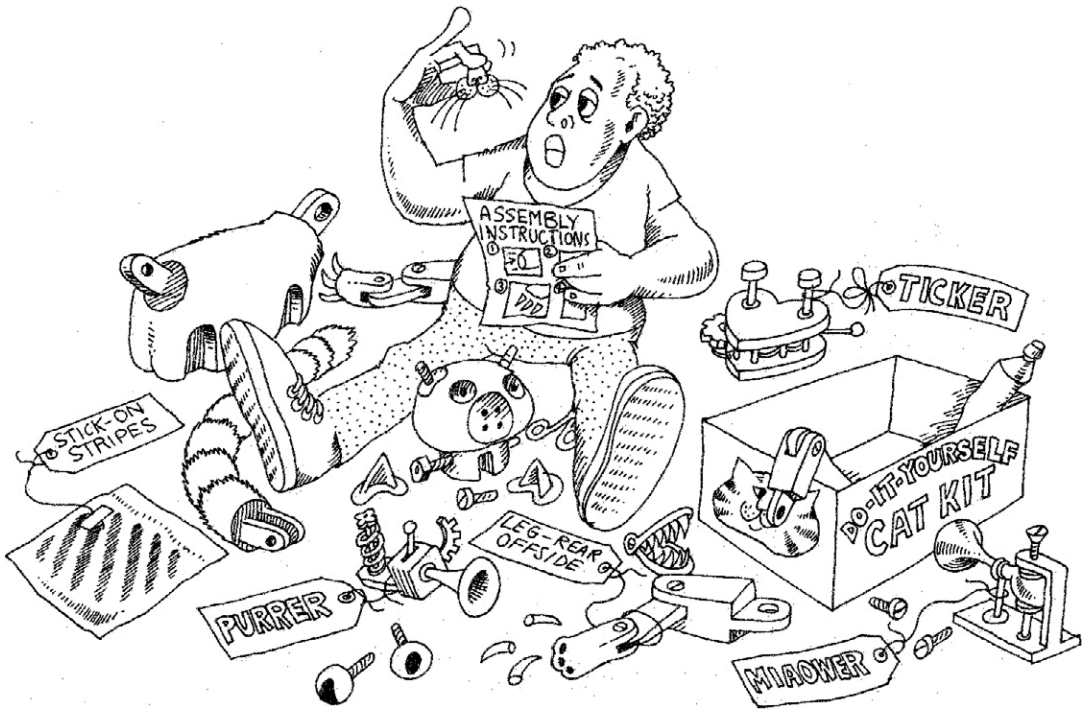
Intelligent encapsulation localizes design decisions that are likely to change. As a system evolves, its developers might discover that, in actual use, certain operations take longer than is acceptable or that some objects consume more space than is available. In such situations, the representation of an object is often changed so that more efficient algorithms can be applied or so that one can optimize for space by calculating rather than storing certain data. This ability to change the

representation of an abstraction without disturbing any of its clients is the essential benefit of encapsulation.

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

The Meaning of Modularity

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



Modularity packages abstractions into discrete units.

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

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

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

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

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

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

The developer must therefore balance two competing technical concerns: the desire to encapsulate abstractions and the need to make certain abstractions visible to other modules. "System details that are likely to change independently should be the secrets of separate modules; the only assumptions that should appear between modules are those that are considered unlikely to change. Every data structure is

private to one module; it may be directly accessed by one or more programs within the module but not by programs outside the module. Any other program that requires information stored in a module's data structures must obtain it by calling module programs" [62]. In other words, strive to build modules that are cohesive (by grouping logically related abstractions) and loosely coupled (by minimizing the dependencies among modules). From this perspective, we may define modularity as follows:

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

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

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

Several competing nontechnical needs may also affect modularization decisions. Typically, work assignments in a development team are given on a module-by-module basis, so the boundaries of modules may be established to minimize the interfaces among different parts of the development organization. Senior designers are usually given respon-

sibility for module interfaces, and more junior developers complete their implementation. On a larger scale, the same situation applies with subcontractor relationships. Abstractions may be packaged so as to quickly stabilize the module interfaces as agreed upon among the various companies. Changing such interfaces usually involves much wailing and gnashing of teeth—not to mention a vast amount of paperwork—so this factor often leads to conservatively designed interfaces. Speaking of paperwork, modules also usually serve as the unit of documentation and configuration management. Having ten modules where one would do sometimes means ten times the paperwork, and so, unfortunately, sometimes the documentation requirements drive the module design decisions (usually in the most negative way). Security may also be an issue. Most code may be considered unclassified, but other code that might be classified secret or higher is best placed in separate modules.

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

Examples of Modularity

Let's look at modularity in the Hydroponics Gardening System. Suppose we decide to use a commercially available workstation where the user can control the system's operation. At this workstation, an operator could create new growing plans, modify old ones, and follow the progress of currently active ones. Since one of our key abstractions here is that of a growing plan, we might therefore create a module whose purpose is to collect all of the classes associated with individual growing plans (e.g., `FruitGrowingPlan`, `GrainGrowingPlan`). The im-

plementations of these `GrowingPlan` classes would appear in the implementation of this module. We might also define a module whose purpose is to collect all of the code associated with all user interface functions.

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

The Meaning of Hierarchy

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

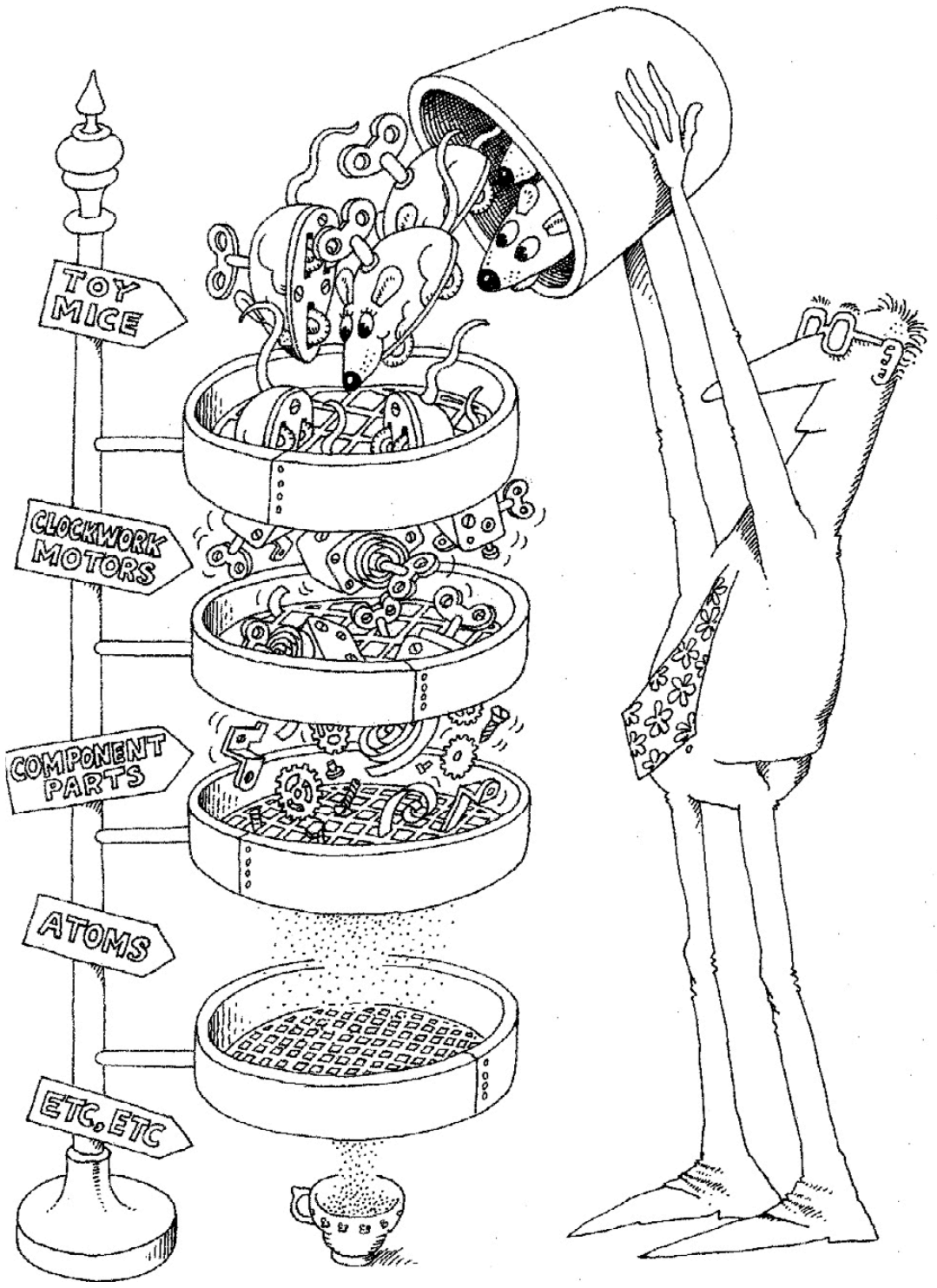
We define hierarchy as follows:

> Hierarchy is a ranking or ordering of abstractions.

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

Examples of Hierarchy: Single Inheritance

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



Abstractions form a hierarchy.

Semantically, inheritance denotes an "is a" relationship. For example, a bear "is a" kind of mammal, a house "is a" kind of tangible asset, and a quick sort "is a" particular kind of sorting algorithm. Inheritance thus implies a generalization/specialization hierarchy, wherein a subclass specializes the more general structure or behavior of its super-

classes. Indeed, this is the litmus test for inheritance: If `B` is not a kind of `A`, then `B` should not inherit from `A`.

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

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

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

define new software in the same way we introduce any concept to a newcomer, by comparing it with something that is already familiar" [64].

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

Examples of Hierarchy: Multiple Inheritance

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

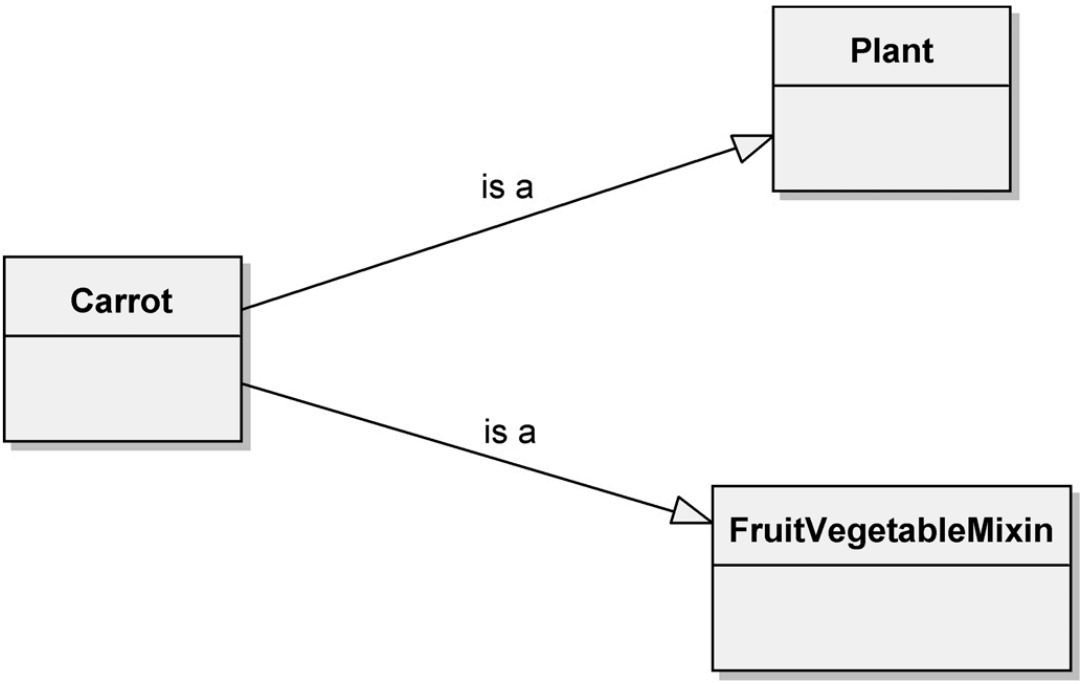
For example, suppose that we choose to define a class representing a kind of plant. Our analysis of the problem domain might suggest that flowering plants and fruits and vegetables have specialized properties that are relevant to our application. For example, given a flowering plant, its expected time to flower and time to seed might be important

to us. Similarly, the time to harvest might be an important part of our abstraction of all fruits and vegetables. One way we could capture our design decisions would be to make two new classes, a `Flower` class and a `FruitVegetable` class, both subclasses of the class `Plant`. However, what if we need to model a plant that both flowered and produced fruit? For example, florists commonly use blossoms from apple, cherry, and plum trees. For this abstraction, we would need to invent a third class, `FlowerFruitVegetable`, that duplicated information from the `Flower` and `FruitVegetable` classes.

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

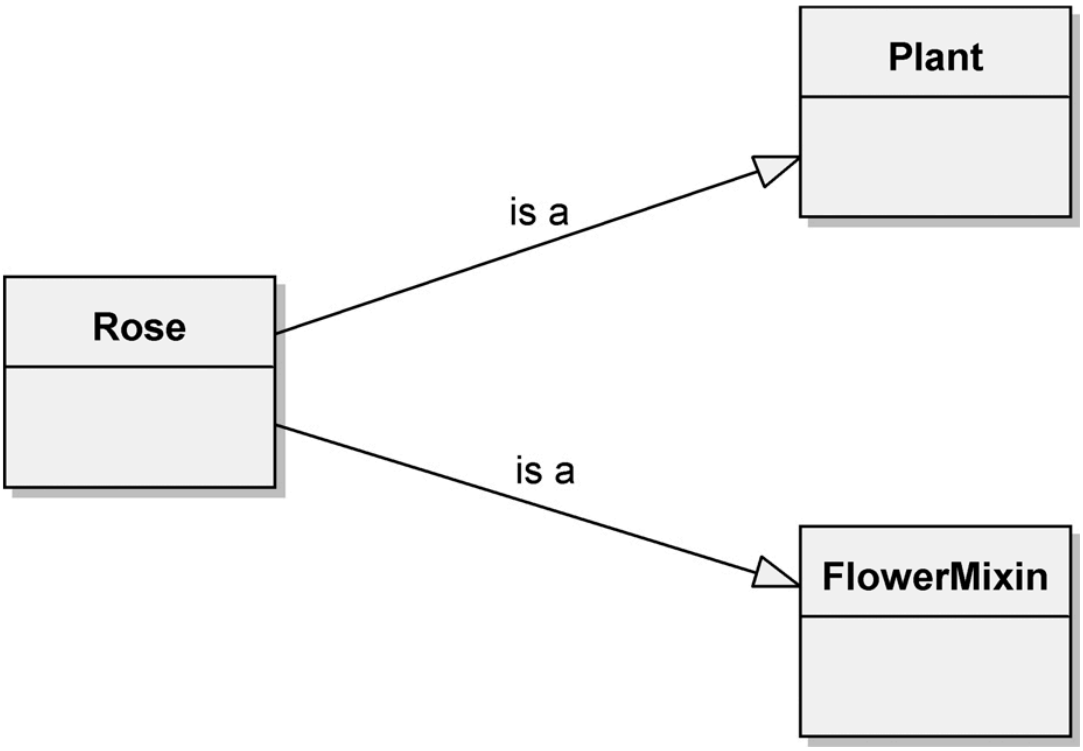
For example, we can define a `Rose` class (see **Figure 2-10**) that inherits from both `Plant` and `FlowerMixin`. Instances of the subclass `Rose` thus include the structure and behavior from the class `Plant` together with the structure and behavior from the class `FlowerMixin`.

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



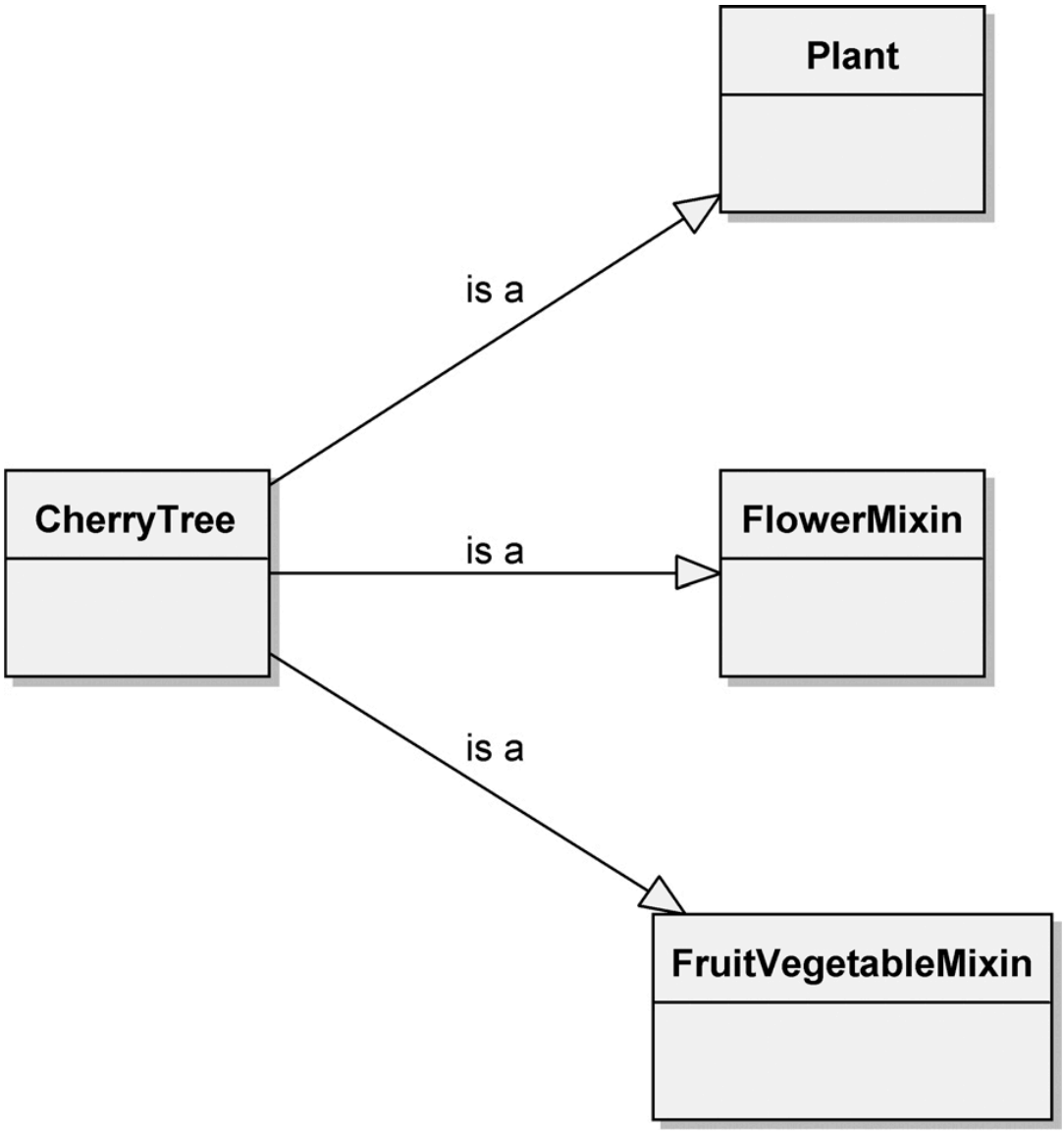
Similarly, a `Carrot` class could be as shown in **Figure 2-11**. In both cases, we form the subclass by inheriting from two superclasses.

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



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

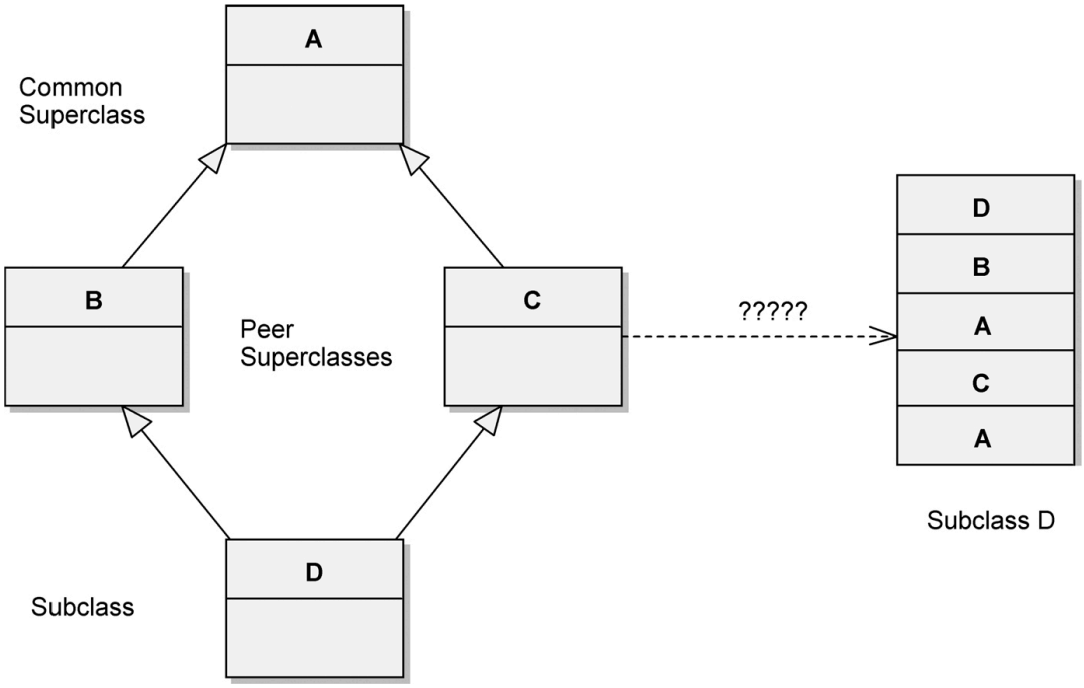
Figure 2-12 The `CherryTree` Class, Which Inherits from Multiple Superclasses



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

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

Figure 2-13 The Repeated Inheritance Problem



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

Examples of Hierarchy: Aggregation

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

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

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

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

we destroy the `Garden` object, we in turn destroy the `GrowingPlan` instance.

The Meaning of Typing

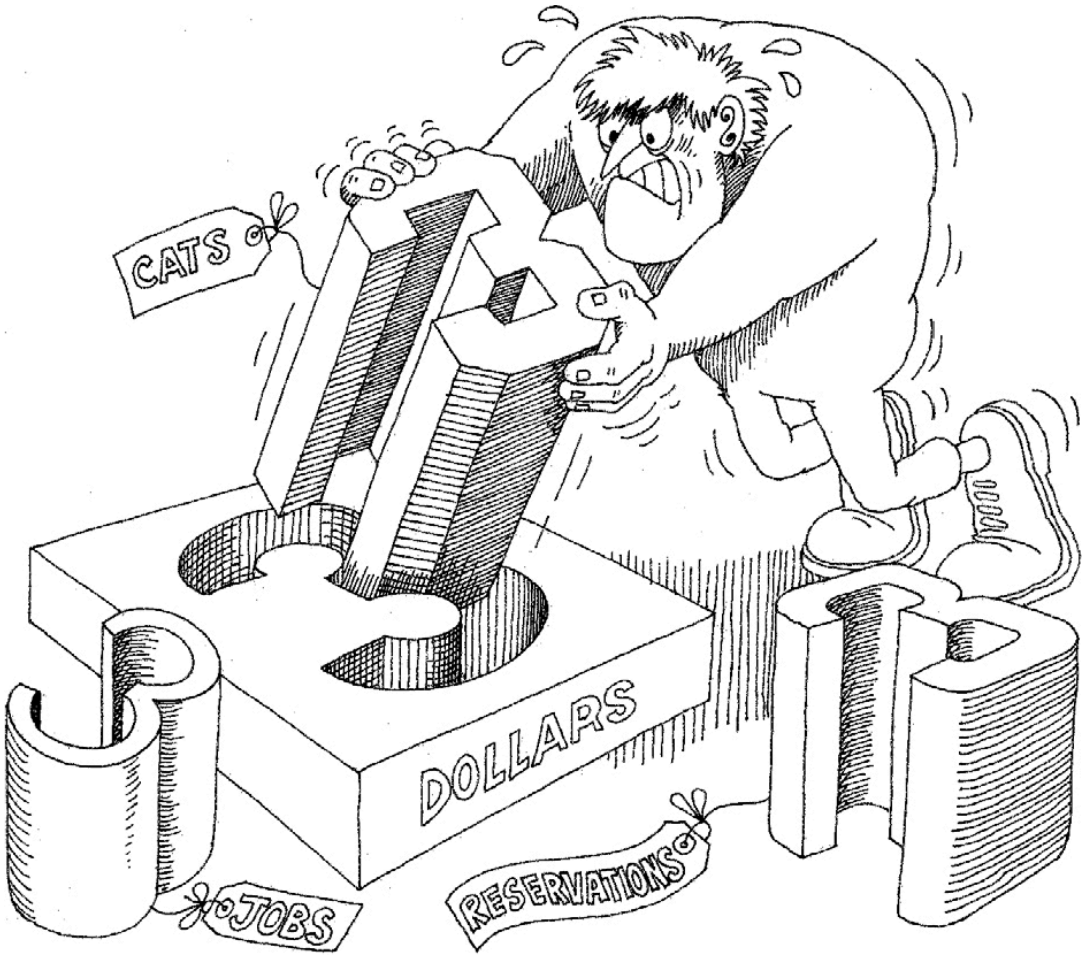
The concept of a type derives primarily from the theories of abstract data types. As Deutsch suggests, "A type is a precise characterization of structural or behavioral properties which a collection of entities all share" [68]. For our purposes, we will use the terms **type** and **class** interchangeably.[2]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

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

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

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

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



Strong typing prevents mixing of abstractions.

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

There are two general solutions to these problems. First, we could use a type-safe container class that manipulates only objects of a specific class. This approach addresses the first problem, wherein objects of different types are incorrectly mingled. Second, we could use some form of runtime type identification; this addresses the second problem of knowing what kind of object you happen to be examining at the moment. In general, however, runtime type identification should be used only when there is a compelling reason because it can represent a

weakening of encapsulation. As we will discuss in the next section, the use of polymorphic operations can often (but not always) mitigate the need for runtime type identification.

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

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

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

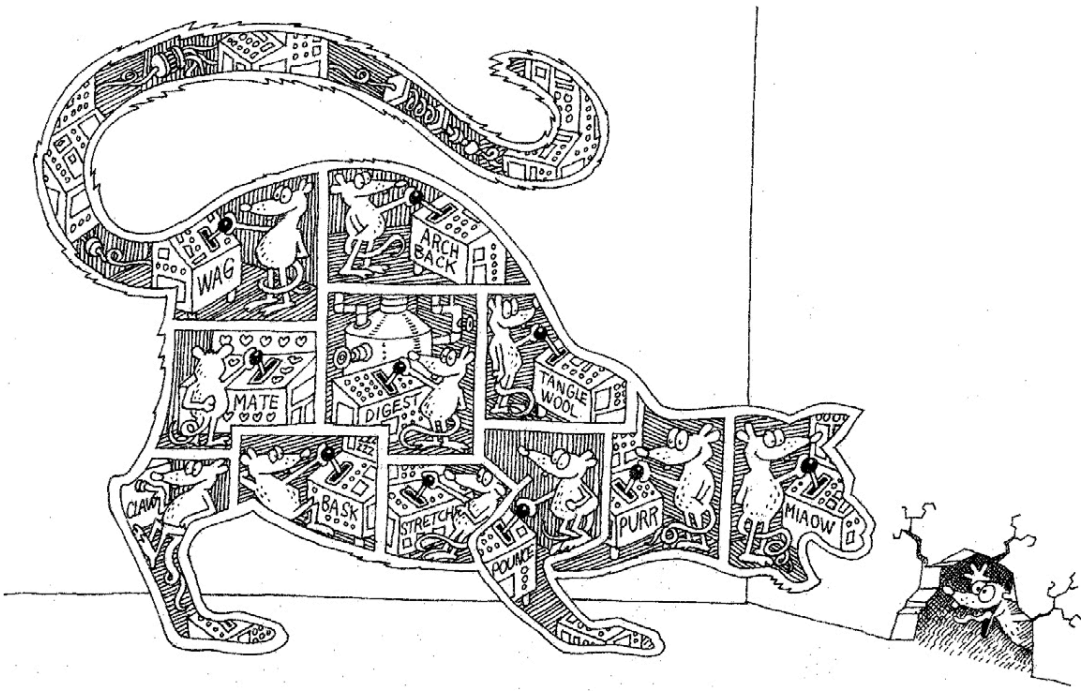
Examples of Typing: Static and Dynamic Typing

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

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

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

The Meaning of Concurrency



Concurrency allows different objects to act at the same time.

For certain kinds of problems, an automated system may have to handle many different events simultaneously. Other problems may in-

volve so much computation that they exceed the capacity of any single processor. In each of these cases, it is natural to consider using a distributed set of computers for the target implementation or to use multitasking. A single process is the root from which independent dynamic action occurs within a system. Every program has at least one thread of control, but a system involving concurrency may have many such threads: some that are transitory and others that last the entire lifetime of the system's execution. Systems executing across multiple CPUs allow for truly concurrent threads of control, whereas systems running on a single CPU can only achieve the illusion of concurrent threads of control, usually by means of some time-slicing algorithm.

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

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

Whereas object-oriented programming focuses on data abstraction, encapsulation, and inheritance, concurrency focuses on process abstraction and synchronization [78]. The object is a concept that unifies

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

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

Examples of Concurrency

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

In general, there are three approaches to concurrency in object-oriented design. First, concurrency is an intrinsic feature of certain programming languages, which provide mechanisms for concurrency and synchronization. In this case, we may create an active object that runs some process concurrently with all other active objects.

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

Third, we may use interrupts to give us the illusion of concurrency. Of course, this requires that we have knowledge of certain low-level

hardware details. For example, in our implementation of the class `ActiveTemperatureSensor`, we might have a hardware timer that periodically interrupts the application, during which time all such sensors read the current temperature and then invoke their callback function as necessary.

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

The Meaning of Persistence

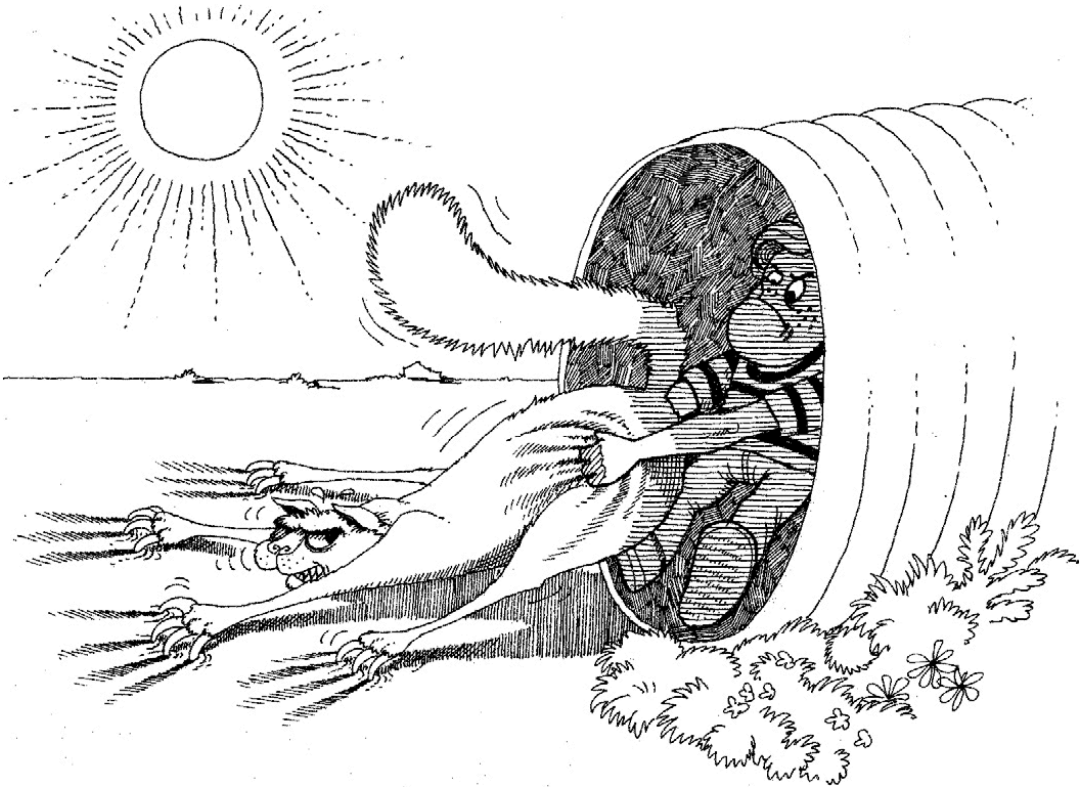
An object in software takes up some amount of space and exists for a particular amount of time. Atkinson et al. suggest that there is a continuum of object existence, ranging from transitory objects that arise within the evaluation of an expression to objects in a database that outlive the execution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

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

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

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

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



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

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

Persistence deals with more than just the lifetime of data. In object-oriented databases, not only does the state of an object persist, but its

class must also transcend any individual program, so that every program interprets this saved state in the same way. This clearly makes it challenging to maintain the integrity of a database as it grows, particularly if we must change the class of an object.

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

To summarize, we define persistence as follows:

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

2.4. Applying the Object Model

As we have shown, the object model is fundamentally different from the models embraced by the more traditional methods of structured analysis, structured design, and structured programming. This does not mean that the object model abandons all of the sound principles and experiences of these older methods. Rather, it introduces several novel elements that build on these earlier models. Thus, the object model offers a number of significant benefits that other models simply do not provide. Most importantly, the use of the object model leads us to construct systems that embody the five attributes of well-structured complex systems noted in **Chapter 1**: hierarchy, relative primitives (i.e., multiple levels of abstraction), separation of concerns, patterns, and stable intermediate forms. In our experience, there are five other

practical benefits to be derived from the application of the object model.

Benefits of the Object Model

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

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

Third, the use of the object model produces systems that are built on stable intermediate forms, which are more resilient to change. This also means that such systems can be allowed to evolve over time,

rather than be abandoned or completely redesigned in response to the first major change in requirements.

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

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

Open Issues

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

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

These issues are the themes of the next five chapters.

Summary

- The maturation of software engineering has led to the development of object-oriented analysis, design, and programming methods, all

of which address the issues of programming-in-the-large.

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